Table of contents

Introduction

Crescendo: A Programmable VST Plugin

Crescendo is a free, programmable VST plugin designed for Windows. It utilizes a built-in programming language, offering a wide array of capabilities for users to design customized VST instruments, audio effects, MIDI effects, and sequencers.

The main objective of Crescendo is to empower users with the flexibility and control to design their own audio tools, as opposed to relying on pre-built plugins. This allows users to craft instruments and effects that are precisely tailored to their artistic vision through the plugin's custom programming language.

Core Concepts

Layers: The Building Blocks of Sound

Crescendo adopts a layered architecture for sound design, a common approach found in many virtual instrument software. Each layer acts as a self-contained processing unit with its own unique set of properties: **Triggers, Sound Generation Instructions, Off Triggers.**

This layered approach offers flexibility in instrument design. By combining multiple layers, each with its own set of triggers, sound generation instructions, and off triggers, users can create complex and nuanced instruments.

The Processing Pipeline

Crescendo handles audio and MIDI data through a clearly defined multi-stage processing pipeline. This pipeline ensures a structured flow of information and allows users to control the sequence of transformations applied to the data. The stages of the pipeline are: **MIDI Processing, Layer Processing, Post-processing**.

The processing pipeline's modular structure allows for flexibility in configuring the plugin's behavior.

VST Variables: User-Controllable Parameters

VST variables (VST VARs), also referred to as VST parameters, are user-defined parameters that can be exposed to the host DAW, providing a means to control aspects of the instrument or effect from the DAW's interface. These variables are often represented as knobs, sliders, or buttons within the plugin's graphical user interface (GUI) or the host DAW's interface, allowing users to adjust them in real time or automate their changes over time.

MIDI CCs and Extended MIDI CCs

MIDI CCs (Control Change messages) are used to send control information from a MIDI controller or sequencer to a VST plugin. Standard MIDI CCs use numbers 0-127, each associated with specific functions like volume, pan, modulation, or expression.

Crescendo extends the standard MIDI CC range with custom CC numbers, often referred to as extended MIDI CCs. These extended CCs offer greater control and automation capabilities. They provide access to parameters like pitch bend, program change, aftertouch, internal variables within the plugin, keyswitch values, and even VST variables.

Expressions: The Foundation of Dynamic Control

Expressions are fundamental to achieving dynamic control in Crescendo, enabling users to go beyond static settings and create instruments that respond to MIDI input, internal modulators, or changes in VST variable values. This dynamic control allows for a wide range of expressive possibilities and nuanced sound design within Crescendo.

Processing Types

Crescendo VST plugin can function as different types of VSTs: **MIDI effect, audio effect, or instrument**. This versatility stems from the structure of the instrument file and the presence of specific instructions within its sections.

MIDI Effects: Shaping MIDI Data

- **Functionality**: A MIDI effect processes incoming MIDI data, modifying or transforming it **before it reaches an instrument or sound generator**. This includes altering note pitches, velocities, or timings, generating arpeggios or chords, quantizing note timings, routing MIDI messages to different channels, and more.
- **Instrument File Structure**: A MIDI effect instrument file typically consists of only a **COMMON** section. This section contains instructions that specifically handle MIDI processing, such as **MIDICH**, **ARPEGGIO**, **CHORD**, **MAP**, **VELOCITY**, **PITCH**, **TIMING**, and **SEQUENCER**.

Audio Effects: Modifying Audio Signals

- **Functionality**: An audio effect processes incoming audio signals, applying transformations to alter their characteristics. Common examples include reverb, delay, chorus, flanger, distortion, equalization, and compression.
- **Instrument File Structure**: An audio effect instrument file requires a **COMMON** section and a **POST** section.
    - The **COMMON** section might contain instructions for handling MIDI input, but these instructions **are not used for audio processing within the POST section**.
    - The **POST** section is where the audio processing instructions are defined, operating on the audio signals present at the input.

Instruments: Generating and Shaping Sound

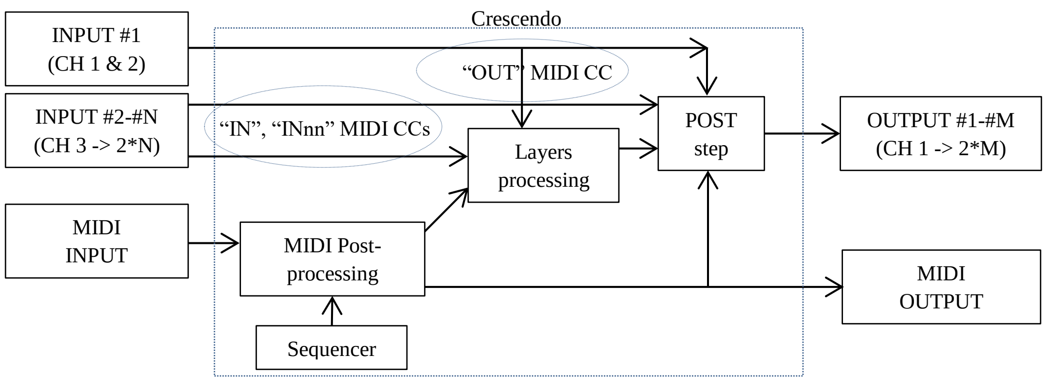
- **Functionality**: An instrument combines MIDI processing, sound generation, and audio processing to create a playable virtual instrument. It responds to MIDI input, generates audio waveforms using oscillators, shapes the sound over time with envelopes, filters the audio signal, and applies effects.
- **Instrument File Structure**: An instrument file needs at least one **LAYER** section in addition to the **COMMON** section. It can also optionally include a **POST** section.
  - The **COMMON** section handles initializations, common instructions, and MIDI processing.
  - The **POST** section allows for global audio processing, applying effects to the combined output of all active layers.
  - The **LAYER** section defines the individual sound-generating units, each with its trigger conditions, sound generation instructions, and output routing.

Multiple I/O options and I/O Diagram

The Crescendo plugin supports multiple dimensions of multiple Input/Output:

- **Multi I/O (Multiple Input/Output):** This moves beyond the limitations of typical single-input, single-output plugins, opening up a world of possibilities for sound design and audio processing. This allows **Sidechaining, Routing to Separate Channels, Multi-Track Routing and more.**
- **Multi Channel:** The MIDI standard allows for up to 16 separate data fluxes in a single MIDI Input, discerned by a number called the channel. Crescendo stores separate values of each MIDI CC and each channel, allowing the correct routing of the parameter values among the channels, including NOTE ON and OFF messages. The cannels can be filtered out or you can allow the triggering of multiple notes from different channels.
- **Multi Instrument:** The MIDI standard includes two MIDI CCs, the program number and the bank number, designed to allow the user to support multiple instruments in real time. Crescendo stores separate values for each channel for program and bank number MIDI CCs, allowing up to 16 separate instruments, swappable in real time. You need 16 separate MIDI controllers to drive them though and an instrument file that includes the appropriate instructions to distinguish in real time the program and bank number.

Visualizing the Flow: Input/Output Diagram



The input/output diagram in the Figure visually represents the various signals and data that the Crescendo VST plugin interacts with. This diagram provides a fundamental understanding of how the plugin handles audio and MIDI information.

Key Elements of the Diagram:

- **Audio Inputs:**
    - **INPUT #1 (CH 1 & 2):** This is the main stereo audio input. Its usage depends on the type of VST:
        - **Effect VST (Crescendo-effect.DLL):** Typically used for the audio signal to be processed.
        - **Instrument VST (Crescendo.DLL):** Can be freely used, often for sidechaining.
    - **INPUT #2-#N (CH 3 -> 2*N):** These are additional stereo audio inputs, available through the `IN` and `INnn` variables. They are primarily used for sidechaining and other auxiliary audio input purposes. If these inputs are not connected, they evaluate to zero within the plugin.
- **Audio Outputs:**
    - **OUTPUT #1 (CH 1 & 2):** The main stereo audio output, associated with the `OUT` variable. It carries the processed audio signal, whether it's the result of layer processing, post-processing, or simply a pass-through of the input signal.
    - **OUTPUT #2-#M (CH 3 -> 2*M):** Additional stereo audio outputs, accessible through the `OUT1` to `OUT99` variables. They enable the plugin to send multiple processed audio signals to different destinations. Unconnected audio outputs are discarded.
- **MIDI Input: MIDI INPUT** receives MIDI messages from the host DAW or up to 16 MIDI controllers. These messages can include note data, control changes, program changes, and other MIDI events.
- **MIDI Output: MIDI OUTPUT** sends MIDI messages generated or modified by the plugin to the host DAW or other connected MIDI devices. All enabled channels are processed and re-transmitted.
- **VST Parameters (VST VARs):** These are the parameters that can be controlled from the host DAW's interface or from Crescendo's GUI. They are represented in the diagram as "MIDI CCs," as they can be accessed with the same functions used for the MIDI CCs.
- **TEMPERAMENT drop down list:** It allows the user to choose among the available temperaments. See TEMPERAMENT instruction below. Normally modifiable only on the interface, it can be optionally linked to some MIDI cc or VST var (See TEMPERAMENTCC below).
- **Optional keyswitches drop down lists:** They allow the user to select pre-programmed options, e.g. playing style. Modifiable in the interface or reacting at some keys on the MIDI keyboard. See KEYSWITCH instruction below.
- Both the VST vars and the drop down lists are accessible inside the instrument file for getting user input, like the other MIDI CCs, with the given MIDI CC ranges.

Signal Flow in the Diagram:

The diagram depicts the following signal flow:

1. **Input:** Audio and MIDI signals enter the plugin through the respective inputs.
2. **MIDI Processing:** Incoming MIDI messages undergo processing, including channel filtering using the `MIDICH` instruction and storage of MIDI CC values. Note messages are

potentially processed by the arpeggiator, chord generator, or sequencer, depending on the instrument's configuration.
3. **MIDI Post-processing:** Processed MIDI note messages may undergo further modifications using instructions like `MAP`, `TIMING`, `PITCH`, and `VELOCITY` . The selected temperament is also applied at this stage.
4. **Layer Processing:** Processed MIDI data triggers the activation of layers, each generating audio signals based on their defined parameters and processing.
5. **Post-processing:** The combined output from all layers, along with any audio inputs, is processed in the **POST** stage using instructions and functions defined in the `POST` section of the instrument file.
6. **Output:** The final audio signals, determined by the values of the `OUTnn` variables, are sent to the corresponding audio outputs. MIDI messages generated or modified by the plugin are sent through the MIDI output.

MIDI Polyphonic Expression (MPE) support:

MPE (MIDI Polyphonic Expression) is a way of using MIDI that allows you to control multiple aspects of each note individually. Instead of applying effects like pitch bend to an entire channel, MPE allows you to bend, modulate, and adjust the pressure of each individual note in a chord. This is achieved by assigning each note to its own MIDI channel, giving it a unique set of control messages.

Multi channel and Multi instrument support, along with some other minor features (per channel polyphony and post-release sample and hold) allow Crescendo to support MIDI Polyphonic Expression (MPE).

A small tutorial consisting in an heavily commented example instrument file supporting most of MPE features is given in a later chapter. Readers are encouraged to read it after having read all this manual to better comprehend all the instructions.

Host Engagement Rules

Here it is a detailed account of how the Crescendo VST plugin interacts with the host DAW (Digital Audio Workstation). These interactions are crucial for the plugin to function correctly and seamlessly integrate into the host environment. Let's break down the key aspects of host engagement rules in Crescendo.

Host Engagement: A Collaborative Relationship

Crescendo, as a VST plugin, relies on the host DAW for various essential functions:

- **Audio and MIDI I/O:** The host DAW handles the audio and MIDI input and output for the plugin, routing signals to and from Crescendo. The host provides the audio stream that Crescendo processes and receives the processed audio back from the plugin. It also manages the MIDI messages that trigger notes and control parameters within Crescendo.
- **Parameter Control and Automation:** The host DAW provides the interface for users to control Crescendo's parameters (VST VARs) and to automate those parameters over time. This control can be through the DAW's mixer, automation lanes, or dedicated plugin windows.

- **Timing and Synchronization:** The host DAW provides the timing information (BPM, time signature, song position) that Crescendo uses for synchronization. This is especially important for the sequencer and any time-based effects or modulations within the plugin.
- **Plugin Management and Configuration:** The host DAW manages the loading, instantiation, and configuration of the Crescendo plugin. It handles the plugin's initialization process, sets up the audio and MIDI connections, and provides access to system resources.

Key Aspects of Host Engagement Rules

There are several important rules and considerations regarding Crescendo's interaction with the host DAW:

- **VST Version Compatibility:** Crescendo is designed to conform to version 2.4 of the VST specification. This ensures compatibility with a wide range of host DAWs that support this VST standard.
- **Platform and Architecture:** The current implementation of Crescendo is for x86-64 Windows systems. This means it's compatible with 64-bit Windows operating systems and requires a host DAW that supports 64-bit VST plugins.
- **Instrument or Effect Configuration:** Crescendo can be configured as either an instrument or an audio effect, depending on the filename of the DLL file (Crescendo.dll for an instrument, Crescendo-effect.DLL for an effect). This distinction is important as it determines how the host DAW handles the plugin's audio routing and MIDI connections.
- **Multi I/O Configuration:** Crescendo supports multiple audio inputs and outputs. The number of inputs and outputs is defined using the `IO` instruction within the instrument file. The host DAW needs to be configured to accommodate these additional I/O channels.
- **Host Temperament Handling:** When using the `HOST` temperament, Crescendo relies on the host DAW to send detune information for tuning adjustments. If the host DAW doesn't support temperaments, the `HOST` temperament behaves like equal temperament.
- **MIDI CC and VST VAR Mapping:** Crescendo allows you to link MIDI CCs to VST variables using the `MIDICC` instruction. This mapping relies on the host DAW to correctly transmit MIDI CC messages to the plugin and to update VST variable values in response to changes from the MIDI controller.
- **DPI Awareness:** Crescendo is designed to be DPI-aware, meaning it automatically adjusts its display based on the screen resolution. However, if the host DAW doesn't properly support DPI awareness, this can lead to scaling issues with the plugin's interface. The `DPIAWARE` instruction can force DPI awareness on the host process, but this might not be compatible with all DAWs and should be used with caution.

A Guide to Audio Tracks, MIDI Tracks, and Return Tracks in Ableton Live and Crescendo

Audio Tracks in Ableton Live and Crescendo

- **Purpose:** Audio tracks in Ableton Live are designed primarily for recording, editing, and processing audio signals.
- **Crescendo Usage:** You can use the Crescendo VST plugin as an audio effect on an audio track. In this scenario, Crescendo acts as a processor for the audio signal passing through the track, applying effects like delay, reverb, or distortion defined in the plugin's instrument file.

- **Input Restrictions:** On an audio track, the first input of a VST (INPUT #1 in Crescendo) is usually reserved for the track's audio data. This limits the options for sidechain routing, as the primary input is occupied by the track's inherent audio.
- **Sidechaining Limitations:** While sidechaining is possible with Crescendo on audio tracks, the limitations on input routing mean you can only sidechain to the plugin's secondary inputs (INPUT #2 and above).

MIDI Tracks in Ableton Live and Crescendo

- **Purpose:** MIDI tracks are used to send MIDI data to instruments, whether they are hardware synthesizers or software VST instruments.
- **Crescendo Usage:** Crescendo can be used on a MIDI track as either an instrument (generating sound from MIDI data) or as a MIDI effect (processing and manipulating MIDI data).
- **Input Flexibility:** MIDI tracks provide more flexibility for sidechaining because all audio inputs of a VST are available. This is because MIDI tracks route external MIDI data to instruments rather than using their primary input for audio data.
- **Crescendo Placement:** When using Crescendo on a MIDI track, the `.dll` version of the plugin should be placed first in the track chain, followed by any other desired effects. If the `-effect.dll` version of Crescendo is used first, it can still function, receiving MIDI messages, but INPUT #1 will be unavailable for sidechaining. Other VSTs in the chain can be only effects, so in the case of Crescendo, the `-effect.dll` version must be used.

Return Tracks in Ableton Live

- **Purpose:** Return tracks function similarly to audio tracks in Ableton Live, with the key difference being that their audio input is fixed and cannot be selected. This means they receive their input from a dedicated send channel, allowing you to create a separate effects chain that can be blended with the main mix.

Limitations and Rules

- **Input/Output Limits:** Ableton Live supports VSTs with a maximum of 4 inputs and 16 outputs. This can restrict complex routing scenarios involving multiple audio sources and destinations.
- **MIDI Redirection Restrictions:** MIDI redirection in Ableton Live requires both the sending and receiving tracks to be MIDI tracks. Detune values associated with temperaments are not preserved during MIDI redirection in Ableton Live 9.7.5, which can affect tuning accuracy.
- **Temperament Support:** Ableton Live 9.7.5 does not have native support for microtuning or temperaments. This means that while Crescendo allows you to use custom temperaments within the plugin itself, those tuning adjustments are not reflected when sending MIDI data to other tracks or instruments within Ableton Live.

Sidechaining

Understanding Sidechaining

Sidechaining is a powerful technique in music production where the audio signal from one source, called the "sidechain input," is used to dynamically control parameters in an effect applied to a separate audio signal, called the "target" signal. A classic example is the "ducking" effect, where the volume of a target track is reduced when the sidechain input signal, often a kick drum, exceeds a certain threshold. This creates a rhythmic pumping effect that gives the kick drum more prominence in the mix.

Sidechaining in Crescendo

Crescendo's Approach to Sidechaining:

- **`INxx` Variables:** Crescendo uses specialized variables, denoted as `INxx`, to represent the values of its various audio inputs. These variables act as conduits for external audio signals routed from other tracks within the host DAW, enabling sidechain functionality.
- **Input Numbering:** The specific `INxx` variable you use corresponds to the input channel you've routed the sidechain signal to. For example, if you route your sidechain to `INPUT #2`, you would use the variable `IN` (or its aliases `IN00` or `IN0`) to access that signal.
- **Flexibility in Processing:** You have the flexibility to utilize `INxx` variables in either the `LAYER` or `POST` sections of your Crescendo instrument file, giving you control over where sidechaining is applied in the signal flow.

Sidechaining with Crescendo in Ableton Live 9.7.5: A Comprehensive Guide

Key Concepts and Limitations

- **Maximum Inputs/Outputs:** Ableton Live 9.7.5 has specific constraints on the number of inputs and outputs a VST plugin can manage. The maximum is **4 inputs and 16 outputs**. This limitation can impact complex sidechaining setups, so it's crucial to be mindful of the number of inputs you're using for sidechaining within Crescendo.
- **Track Compatibility:** The type of track in Ableton (Audio or MIDI) directly influences your sidechaining options. Audio tracks are designed for audio processing, with their **first input (INPUT #1) typically dedicated to audio data** from sources like clips or external inputs. This limits sidechain routing options on audio tracks. In contrast, **MIDI tracks, intended for routing MIDI data to instruments, offer more flexibility for sidechaining** as all their audio inputs are available for routing external audio signals.
- **Crescendo Placement:** The position of the Crescendo plugin within a MIDI track's signal chain is critical. If Crescendo is the **first plugin in the chain, you can sidechain to INPUT #1**. However, if Crescendo is used as an effect further down the chain, **INPUT #1 is not selectable for sidechaining**.

Specific Steps for Sidechaining Setup

1. **Identify Sidechain Source:** Determine the audio track whose signal you want to use as your sidechain input.
2. **Route Sidechain Signal:** Navigate to the I/O section of your sidechain source track in Ableton. Route its output to one of the available audio inputs on the target track where Crescendo is loaded. Typically, you'll use **INPUT #2 or higher** as INPUT #1 might be reserved for the target track's audio.
3. **Crescendo's IN Variable:** In your Crescendo instrument file, use the corresponding `INxx` variable to access the routed sidechain signal. For example, if you routed the sidechain to

INPUT #2, you would use the variable `IN` (or its equivalents `IN00` or `IN0`) to access it within your code.
4. **Implement Sidechain Control:** Apply the sidechain signal (`INxx` variable) to control parameters within your Crescendo code. You can dynamically alter volume, filter cutoff, or other effects using the sidechain input. For instance, you could multiply an oscillator's output by the `IN` variable to achieve a ducking effect.

For additional details, please consult the relevant Ableton KB page: https://help.ableton.com/hc/en-us/articles/209775325-Sidechaining-a-third-party-plug-in

General Sidechaining Implementation

While the specifics vary between DAWs, the core principles are consistent. Refer to your DAW's documentation for compatibility and routing options. Generally:

1. **DAW Compatibility:** Ensure your DAW supports sidechaining and routing audio between tracks as an input.
2. **Routing Mechanisms:** Understand the routing options in your DAW (sends, busses, auxiliary channels) to connect the sidechain source to the target plugin.
3. **Plugin Compatibility:** Make sure the target plugin supports sidechain inputs, typically documented in the plugin's manual. Crescendo does support sidechaining.
4. **Sidechain Input Access:** Once routing is set up, locate and access the sidechain input within the target plugin, which might involve dedicated input channels or internal routing mechanisms.

Outputs Routing

Crescendo is a versatile VST plugin capable of functioning as an instrument, audio effect, and MIDI effect, with robust multi-I/O capabilities. Understanding how to route outputs effectively is crucial for leveraging its full potential in Ableton Live 9.7.5 and other DAWs. This breakdown provides a comprehensive view of output routing in Crescendo, focusing on its interaction with Ableton Live's specific features and limitations.

General Output Routing in Crescendo

- **Multiple Outputs:** Crescendo can handle **multiple stereo audio outputs** simultaneously, allowing you to send different elements of the instrument or effect to distinct destinations for individual processing or mixing.
- **OUT Variables:** The core of Crescendo's output routing lies in the use of **OUT variables** within its programming language. These variables determine which audio signal is sent to a specific output channel.
    - `OUT`, `OUT0`, or `OUT00` represent the **main stereo output**, connected to **OUTPUT #1**. This output initially contains the combined signal from all active layers plus the input signal on **INPUT #1** (if connected). The **POST section** can further manipulate this combined signal.
    - `OUTnn` variables (where `nn` > 0) correspond to **secondary outputs**, connected to **OUTPUT #2** onwards. For example, `OUT1` represents **OUTPUT #2**, `OUT2` represents **OUTPUT #3**, and so on.

- **Initialization of OUT:** In the LAYER sections, `OUTnn` variables are initialized to 0 at each sample. The last instruction writing to an `OUTnn` variable within a layer defines the signal sent to the corresponding output. In the POST section, `OUT` is initialized with the sum of all active layers' outputs plus the signal on INPUT #1, `OUTnn` variables are initialized with the sum of all active layers' corresponding `OUTnn` outputs.
- **Signal Flow:** The final value assigned to an `OUTnn` variable after processing in the POST section (or directly from a LAYER section if no POST section is used) determines the audio signal sent to the corresponding physical output of the plugin.

Output Routing in Ableton Live 9.7.5

- **Ableton's I/O Section:** Ableton Live manages audio routing through the **I/O section** located in each track's view. This section provides controls for selecting input sources, output destinations, and routing modes.
- **Connecting to Other Tracks:** You can route Crescendo's outputs to different tracks within Ableton by selecting the desired destination track and output channel in the **I/O section** of the track where Crescendo is loaded. Ableton provides options for routing **pre-FX, post-FX, or directly from specific outputs** of a VST.
- **Sidechaining and Output Routing:** As discussed in the previous section, sidechaining in Ableton involves routing the output of one track to a specific audio input of a VST on another track. This routing is also managed in the I/O section.

Key Considerations for Output Routing

- **Ableton's Input/Output Limits:** As you noted previously, Ableton Live 9.7.5 restricts VST plugins to a maximum of **4 inputs and 16 outputs**. Be mindful of this when designing complex instruments with multiple outputs.
- **MIDI and Audio Tracks:** In Ableton, MIDI tracks offer more flexibility for sidechaining and input routing. However, when it comes to output routing, both MIDI and audio tracks can route audio signals to other tracks using the I/O section.
- **Signal Flow and Processing:** Understanding the signal flow within Crescendo, from the LAYER sections through the optional POST section, is crucial for correctly routing outputs and applying effects. The order of instructions determines the order of processing and how signals are combined and sent to outputs.

Midi Redirection

MIDI redirection in Ableton Live involves using the MIDI output from one MIDI track as the input for another MIDI track. This enables you to chain MIDI effects, process the same MIDI data with multiple instruments, or control various instruments from a single MIDI source.

General MIDI Redirection in Ableton Live 9.7.5

- **Track Requirements:** Both the source and destination tracks must be MIDI tracks.
- **Origin Track Setup:**
  - You can use a complex plugin chain or a single VST like Crescendo with MIDI processing instructions in its COMMON section.
  - The track can simultaneously output audio and MIDI.
- **Destination Track Setup:**

    - Select the origin track as the MIDI input source in the destination track's I/O section.
    - Choose the desired routing mode (Pre FX, Post FX, or the name of a VST that supports MIDI output, like Crescendo).
    - NOTE: Crescendo apply a prefiltering for channels. Filtered out channels are not retransmitted.
- **For additional details**, please consult the relevant Ableton KB page: https://help.ableton.com/hc/en-us/articles/209070189-Accessing-the-MIDI-output-of-a-VST-plug-in

Detuning Limitations in Ableton Live 9.7.5

Ableton Live 9.7.5 **does not currently support temperaments**, meaning it cannot handle fractional detune values. While Crescendo allows you to apply custom temperaments using the `TEMPERAMENT` instruction, these **fractional detune values are lost during MIDI redirection** in Ableton Live 9.7.5. This means that if your origin track uses a temperament that deviates from standard 12-tone equal temperament, the detuned notes will be rounded to the nearest semitone when received by the destination track.

Impact on MIDI Redirection

This limitation significantly impacts the use of MIDI redirection for microtonal music in Ableton Live 9.7.5. If you're using Crescendo to generate MIDI data with microtonal detuning, the subtle pitch variations will be lost when redirected to another track. The destination instrument will receive only the quantized note values, resulting in a loss of the intended microtonal nuances.

Workarounds and Alternatives

- **Other DAWs:** If you require precise microtonal control during MIDI redirection, you may consider using a DAW that fully supports temperaments and preserves detune values.
- **Internal Routing within Crescendo:** You could create a multi-timbral instrument within a single instance of Crescendo, using different layers for different parts with microtonal detuning. This avoids relying on MIDI redirection between tracks in Ableton Live.

General MIDI Redirection Concepts

MIDI redirection, as a concept, is not limited to Ableton Live. Other DAWs with support for VST plugins and internal routing will likely offer similar capabilities. Key points to keep in mind when working with MIDI redirection in general:

- **DAW Compatibility:** Not all DAWs handle MIDI redirection in the same way. Consult your DAW's documentation for specific instructions and limitations.
- **Temperament Support:** The handling of temperament data during MIDI redirection can vary between DAWs. Some may preserve detune values accurately, while others might not.
- **Latency:** MIDI redirection can introduce latency, especially in complex setups or on systems with limited processing power. Be mindful of potential timing issues.

DAW Interaction issues

Bank Load and Saving

The plugin implements bank load and save. DAWs use these functions for retrieving the parameters and for saving and restoring from/to its internal file format.

The stored information are the instrument full file path, the knob and drop box positions, including VST VARs, keyswitches and the current temperament selected, the current program and bank number selected for each of the 16 MIDI channels and the last information inserted by the end user in the SAMPLEUI controls.

VST VARs, drop box positions and the other data are restored to the saved values and the file is reloaded and recompiled.

NOTE: the VST assumes that the underlying instrument file has not changed since the last bank loading, otherwise you will have UNDEFINED behavior.

Multithreading

Crescendo is **mono thread and thread safe**, meaning that while it operates on a single thread, it is designed to be used safely in multithreaded environments. However, the behavior of Crescendo in a multithreaded context relies heavily on how the host DAW manages plugin instances.

- **Separate Threads per Instance:** The ideal scenario is for the host DAW to use a **separate thread for each instance of the plugin**. This approach allows multiple instances of Crescendo to run concurrently on different processor cores, maximizing performance and preventing conflicts.
  You can load multiple VST plugin instances, each in a different track, or multiple instances in chain in the same track or both. The DAW loads one DLL VST instance and creates multiple VST plugin object instances. Each instance can have its own current selected instrument file and its own current settings.
  Note that Settings.ini is the same for all instances but the instrument file can contain settings instructions that overwrite those in Settings.ini.
  Each VST instance will play with its own options and the DAW will combine the sounds.
- **Testing with Ableton Live:** Ableton Live 9.7.5 handles Crescendo's threading in a way that allows for multicore utilization and prevents performance issues.
- **Potential Issues with Other DAWs:** It's essential to note that other DAWs might not handle threading in the same way. If a DAW attempts to run multiple instances of Crescendo on the same thread, it could lead to performance bottlenecks or unexpected behavior. It would be prudent to check the documentation or support resources for your specific DAW to understand its multithreading behavior with VST plugins.

Additional Notes

- **Input and Output Buffers:** Some DAWs, such as Ableton Live, might use the same memory buffers for both input and output audio data when interacting with VST plugins. This behavior requires careful handling within the plugin's code to prevent data corruption or unexpected results, especially when performing sidechaining or other complex audio routing. Crescendo includes specific code to address this potential issue in Ableton Live.
- **SaviHost and Reaper Compatibility:** SaviHost and Reaper, two VST host applications, uses separate input and output buffers. This simplifies the plugin's interaction, as it doesn't

require special handling to prevent buffer conflicts. Additionally they treat instrument and effect plugins equally regarding sidechain input. This behavior, while potentially useful for some workflows, deviates from the strict VST standard and might not be consistent with other DAWs.

Using Crescendo

Installation and Setup

Overview

Crescendo is a programmable VST plugin.
It is conforming to the version 2.4 of the VST specification.
Currently only x86-64 Windows version is implemented.
It was tested in Ableton Live 9.7.5, LMMS 1.2.2, Reaper 7.19 and SaviHost64.

Obtaining the DLL Files

To use the Crescendo VST plugin, you first need to obtain its DLL (Dynamic Link Library) files. The plugin package comes in a compressed format like `.zip` or `.7z`. You can obtain these files from a trusted source, for example the KVR Audio website. The package should contain two DLL files:

- **Crescendo.dll:** This file is primarily intended to be used as a VST instrument.
- **Crescendo-effect.dll:** This file is intended to be used as a VST audio effect.

The package may also include:

- Some PDF manuals (E.g. this file).
- Example instrument files (.txt).

Placing the DLL Files

Once you have the DLL files, you need to place them in your DAW's VST (Virtual Studio Technology) plugin folder. The location of this folder varies depending on your DAW and operating system. You can usually find this information in your DAW's documentation or preferences.

To install the Crescendo plugin:

1. **Identify your DAW's VST folder.**
2. **Copy both DLL files ("Crescendo.dll" and "Crescendo-effect.dll") into the VST folder.**
3. **Restart your DAW** (if it was open during the copying process) so that it can recognize the new plugin.

After restarting, you should be able to find Crescendo in the list of available VST instruments and effects within your DAW.

Distinguishing Between Crescendo.dll and Crescendo-effect.dll

Crescendo.dll and **Crescendo-effect.dll** are two separate files that constitute the Crescendo VST plugin. These files contain identical code and functionality, but they are differentiated solely by a setting within the VST standard that classifies plugins as either instruments or effects.

The need for two DLLs arises from limitations among different DAWs.

- **Fixed Classification:** Most DAWs do not allow for dynamic reclassification of a plugin as an instrument or an effect after it has been loaded.
- **Inconsistent DAW Behavior:** Different DAWs exhibit varying degrees of strictness in adhering to this classification. Some DAWs, such as Audacity, only support VST effects, while others, like Ableton Live, strictly enforce the instrument/effect distinction. Reaper, however, is more flexible and allows both DLLs to be used interchangeably.

DAW-Specific Treatment and Compatibility Considerations

- **Audacity:** Audacity only supports VST effects. Therefore, only "Crescendo-effect.dll" will load and function correctly in Audacity.
- **Ableton Live:** Ableton Live rigorously follows the VST standard's classification. "Crescendo.dll" must be used for instrument functionality, while "Crescendo-effect.dll" must be used for audio effects.
- **Reaper:** Reaper is more flexible and treats both DLLs interchangeably, allowing them to be used as either instruments or effects.
- **Other DAWs:** The behavior of other DAWs might vary. It is advisable to consult the documentation or support resources for your specific DAW to determine how it handles these two files.

Functionality and Intended Roles

Despite the enforced distinction, the two files are identical in actual content, byte for byte, only the name is different. If you have only one file, just copy it and rename both with the names given above, then copy them in the VST folder of your DAW.

- **Crescendo.dll** is primarily intended as a VST instrument. This means it is designed to generate audio output based on MIDI input.
- **Crescendo-effect.dll** is designed as an audio effect. It focuses on processing incoming audio signals to modify their characteristics.

It's important to understand that the categorization is more about compatibility with different DAWs than about limiting functionality.

- **Theoretical Capabilities:** Even though "Crescendo.dll" is primarily an instrument, it can theoretically process audio input through sidechaining mechanisms. Similarly, "Crescendo-effect.dll," despite being an effect, can receive and respond to MIDI data, but this might be limited in certain DAWs.
- **MIDI Message Routing:** Most DAWs will not route NOTE ON/OFF MIDI messages to the -effect version, because it is expected to be an audio effect. You might be able to hear the audio effect on the input audio, but playing MIDI keys might not trigger any sound.

Technical details: The `isSynth` Method

This method is crucial for establishing how a DAW categorizes and handles the plugin, impacting features like audio routing, MIDI connections, and sidechaining capabilities.

Defining the Plugin Type

The `isSynth` method is a communication mechanism between the VST plugin and the host DAW. This method essentially informs the DAW whether the plugin should be treated as a synthesizer (an instrument that generates sound) or as an effect (a processor that modifies existing audio).

- **Technical Implementation:** DAWs typically implement a method named `isSynth(bool)`, which the VST plugin calls to declare its type. This call usually happens once, during the plugin's initialization.
- **Crescendo's Approach:** Crescendo, by default, determines its type based on the filename of the DLL:
    - `Crescendo.dll`: This version calls `isSynth(true)`, identifying itself as an instrument.
    - `Crescendo-effect.dll`: This version calls `isSynth(false)`, identifying itself as an effect.

Limitations and Inconsistent DAW Behavior

DAWs can vary in how strictly they adhere to the `isSynth` declaration and how consistently they implement this functionality. Some key limitations and challenges include:

- **Caching of `isSynth` Call:** Some DAWs, like Ableton Live, cache the initial `isSynth` call and ignore subsequent attempts to change the plugin's type. This means the initial declaration is crucial and cannot be modified later.
- **Inflexible DAWs:** Certain DAWs may require a certain type of VST plugin, like Audacity, needing a specific `isSynth` declaration.

It is recommended that users consult their DAW's documentation and experiment to determine the best approach for their specific setup and workflow.

Initialization

The Two Stages of Initialization via Settings.ini

Crescendo utilizes a "Settings.ini" file located in the `<My Documents>\Crescendo` directory to manage global settings. This file is processed in two distinct stages:

Stage 1: Initial Plugin Loading

- **Crescendo folder Location and Creation:** When Crescendo loads, it searches for the Crescendo folder (`<My Documents>\Crescendo`). If it does not exist, Crescendo creates it.
- **Settings.ini Location and Creation:** When Crescendo loads, it searches for the "Settings.ini" file. If it's not found, Crescendo creates it with default values and some predefined temperaments.

- **Simplified Parser:** A streamlined parser processes only four specific instructions from the "Settings.ini" file during this stage:
    - **VSTVARS:** This instruction defines the initial number of VST variables (also called VST parameters or VST vars) that are exposed to the host DAW.
    - **INTERFACE:** This instruction sets the initial size of the plugin's graphical user interface (GUI) window in pixels.
    - **DPIAWARE**: This instruction can force DPI awareness on the host process, but this might not be compatible with all DAWs and should be used with caution.
    - **IO:** This instruction configures the number of audio inputs and outputs that the plugin will use.
- **Early Execution:** These four instructions are executed at the beginning of the plugin's operation due to limitations imposed by some early versions of the VST standard and certain DAWs.

- **FFMPEG executable Location**:
    - During initialization, after processing the "Settings.ini" file, Crescendo attempts to **locate the FFMPEG executable ("FFMPEG.EXE") on the user's system**. It searches for the executable in several predefined locations within the `<My Documents>\Crescendo` directory:

        - `<My Documents>\Crescendo\`
        - `<My Documents>\Crescendo\FFMpeg\`
        - `<My Documents>\Crescendo\FFMpeg\x64`
        - `<My Documents>\Crescendo\FFMpeg\bin`
        - `<My Documents>\Crescendo\FFMpeg\x86`

    If `FFMPEG.EXE` is found in any of these locations, Crescendo **stores the full file path** for later use. This path storage allows the plugin to readily access FFMPEG whenever it encounters an unsupported audio file format.

    If `FFMPEG.EXE` is not found in any of these locations, Crescendo will continue to run normally, but will fail to load unsupported files, leaving the sample slot in the previous state and putting an error in the log window.

    You can install the single static linked file (80+ MB in size) or the standard installation.

    Importance of FFMPEG Initialization Step: This initialization step of detecting FFMPEG and storing its path is crucial for several reasons:

    - **Enhanced Compatibility:** It allows Crescendo to work with a diverse range of audio sources, accommodating user workflows that might involve various media file formats.
    - **Streamlined User Experience:** Users do not need to manually convert unsupported files before loading them into Crescendo. The plugin handles this conversion automatically and transparently.
    - **Efficiency:** The initialization step is performed only once when the plugin loads. Subsequent use of FFMPEG does not require repeating this search, optimizing the file loading process.

Stage 2: Instrument File Loading

- **Full Parser:** Every time a new instrument file is loaded, the "Settings.ini" file is loaded and processed again, but this time using the full parser, which can handle all instructions.
- **Global Settings:** The settings in "Settings.ini" act as global settings, establishing a baseline configuration for all instrument files.
- **Overrides:** Declarations within individual instrument files can override the global settings defined in "Settings.ini." This provides flexibility for instrument-specific configurations.

Importance of Settings.ini

The "Settings.ini" file plays a vital role in ensuring compatibility across different DAWs and maintaining consistent behavior across instruments. It is recommended to **include the IO instruction in "Settings.ini"** to guarantee consistent audio routing, especially for DAWs that don't allow modifying audio inputs and outputs at runtime.

File Management

Main Directory: `<My Documents>\Crescendo`

The primary directory for Crescendo is located at `<My Documents>\Crescendo`. This directory is created automatically if it does not exist when the plugin is loaded. It serves as the central hub for various subdirectories and files related to the plugin's functionality.

Instrument Files: Instrument and Sample Subdirectories

Instrument files, which define how audio is processed, including layer specifications, sound generation, and post-processing.

All instrument files should reside in <My Documents>\Crescendo, but full or relative path for the files are supported. All the samples paths are relative to the instrument file path and can be an absolute path (not recommended for portability).

When a file is loaded with the "Browse" or “Reload” buttons, it is loaded and compiled.
The program name of the Crescendo VST is changed to the file name (without the path) to better identify it.
The knob positions and the drop box position are set to 0 or the initial value given in the instrument file. A signal is sent to the host DAW as if the user modified all the knobs: this triggers in host DAWs the pausing/cancelling of parameter automations.

Note that if the instrument file or the Settings.ini are modified e.g. in a text editor, they should be reloaded for the modifies to take effect.

Sample Files

The paths to sample files, referenced in the instrument files using the `SAMPLE` instruction, are relative to the location of the instrument file. Absolute paths are permitted but not recommended for portability reasons. This relative path structure ensures that instruments and their associated samples can be easily moved or shared without breaking the file links.

Subdirectories for Organization

There is not a predefined subdirectory structure for instruments and samples, but it's a good practice to create subdirectories within the main Crescendo directory to organize files based on instrument type, project, or any other preferred categorization. For example, you might have subdirectories like "Pianos," "Drums," "Synths," or "Orchestral" to group instruments of similar types. Similarly, you could create subdirectories for specific projects or collaborations.

Cache Directory: `<My Documents>\Crescendo\Cache`

Crescendo utilizes a dedicated cache directory, located at `<My Documents>\Crescendo\Cache`, to store audio files that have been converted by FFMPEG. This directory plays a crucial role in optimizing performance and minimizing redundant conversions.

Caching Mechanism:

When an unsupported audio file is loaded, Crescendo uses FFMPEG to convert it to a supported format and stores the converted file in the cache directory. The file name is decorated with the file size in bytes and the last modify date/time, so if you load an exact copy of the file from some other folder (e.g. a backup copy), with the same file name, file size and last modify time, the cached version will be used. No content checking is performed because is too costly.

Imported Directory: `<My Documents>\Crescendo\Imported`

Crescendo's "SF2 Bulk Import" feature, which converts SoundFont 2.04 files, creates a dedicated "Imported" directory within the main Crescendo directory. This directory houses subdirectories and files related to imported SoundFonts.

Structure Within "Imported":

- **SoundFont Subdirectories:** For each imported SoundFont, a subdirectory is created within "Imported," named after the SoundFont file (without the .sf2 extension). This subdirectory contains the converted data for that specific SoundFont.
- **"samples" Subdirectory:** Within each SoundFont subdirectory, a "samples" subdirectory is created to store the extracted samples in WAVEFILE format.
- **Preset Files:** Each preset within the SoundFont is converted into a separate text file (.txt) and stored in the SoundFont subdirectory.
- **"00000_ALL_INSTRUMENTS.txt" File:** This file contains all instruments from the SoundFont, along with controls for selecting the current instrument.
- **"00000_ALL_INSTRUMENTS_16.txt" File:** This file contains all instruments from the SoundFont, along with controls for selecting the current instrument for all 16 channels.

Supported File Formats

Here it is an overview of the various file formats supported by Crescendo, encompassing instrument definitions, audio samples, multimedia files, tuning systems, sequencer data, and SoundFont libraries.

Instrument Files: Defining Crescendo's Instruments and Effects

Crescendo's instruments and effects are defined using plain text files, both ANSI and UTF-8 with or without BOM. Other UTF encodings are not supported. There is not a required file extension, but the `.txt` extension allow the user to use their preferred text editor.

Content and Structure:

These instrument files contain code written in Crescendo's dedicated programming language, outlining the instrument's or effect's behavior, parameters, and sound generation processes. The language uses a structured approach with keywords, instructions, and expressions to define various aspects.

Location:

Instrument files are typically stored in the main Crescendo directory located at `<My Documents>\Crescendo`. However, the plugin supports loading files from any location using absolute or relative file paths.

Audio Files: Direct Support and FFMPEG Integration

Crescendo directly supports certain audio file formats, while leveraging the FFMPEG library to expand its compatibility to a wider range of media files.

Directly Supported Formats:

- **WAV:** Uncompressed PCM WAV files in various bit depths (8, 16, 24, 32-bit integer, and 32 or 64-bit IEEE floating point) and channel configurations (mono and stereo). Both big and little endian encodings are supported.
- **AIFF:** Uncompressed PCM AIFF files with up to 32-bit integer sample depth, in both big and little endian encodings.
- **AIFF-C:** Uncompressed PCM AIFF-C files with up to 32-bit integer and 32 or 64-bit IEEE floating point sample depth, supporting both big and little endian encodings.

FFMPEG Integration:

For audio and video file formats that Crescendo doesn't natively support, it utilizes the FFMPEG library, if installed on the user's system. FFMPEG allows the plugin to extract the first audio track from various media file formats, including video. This extracted audio is cached for efficient use.

Potential Issues:

Even with FFMPEG, Crescendo might encounter issues converting some files, like encrypted audio files, such as Ableton Live's AIFF-C files.

SoundFont Files: Importing Existing Sound Libraries

Crescendo supports importing SoundFont 2.04 files (`.sf2` extension), allowing users to integrate pre-existing sound libraries into the Crescendo environment.

Import Process:

The "SF2 Bulk Import" button in the Crescendo interface triggers the conversion process. It creates an "Imported" subdirectory within the main Crescendo directory and a subdirectory named after the SoundFont file. This SoundFont subdirectory stores:

- **Extracted Samples:** The samples from the SoundFont are extracted and saved in WAVEFILE format (16-bit or 24-bit) within a "samples" subdirectory.
- **Preset Files:** Each SoundFont preset is converted into a separate text file (`.txt`) containing the code to emulate the preset in Crescendo.
- **ALL_INSTRUMENTS Files:** A file named "00000_ALL_INSTRUMENTS.txt" contains all instruments from the SoundFont, with controls for selecting the current instrument using MIDI Program or Bank controls or dedicated knobs. In this file all 16 channels can be used and if you use the program and bank MIDI CC and the pedals, 16 separate settings are recognized. The file's knobs are linked to the channel 0. To be able to automate all 16 channels with your DAW, you must use "00000_ALL_INSTRUMENTS_16.txt". Be aware that this file has up to 69 VST VARS and some knobs may not be usable by your DAW (e.g. Ableton has a limit of 64 VST VARS).

Tuning Files: Exploring Alternative Tuning Systems

Crescendo supports various file formats related to microtonal music and custom tuning systems, providing flexibility beyond the standard 12-tone equal temperament.

SCALA Files:

SCALA files (`.scl` extension) define microtonal scales by specifying the intervals that make up the scale. Crescendo uses these intervals to calculate the tuning of each note.

TUN Files:

Crescendo supports AnaMark tuning files (`.tun` extension) in versions 0 and 1. These files define musical tunings by specifying base frequencies and deviations from equal temperament for various keys.

KBM Files:

KBM files (`.kbm` extension), associated with the KBM microtonal keyboard layout, can work in conjunction with SCALA files. They provide information about keyboard remapping and specific tuning settings, potentially modifying how the temperament from the SCALA file is mapped to the keyboard.

Sequencer Files: Storing and Sharing MIDI Sequences

The sequencer within Crescendo can store and load sequences of MIDI notes using TAPE files (`.tap` extension). These files utilize the CSV (comma-separated value) format, making them easily editable and shareable.

TAPE File Content:

Each row in a .tap file represents a single note on the sequencer's TAPE. The row contains five comma-separated floating-point numbers:

1. Trigger time in seconds from the start of the sequence (0.0 for the first note).
2. Release time in seconds from the start.
3. Note pitch in semitones (e.g., 69.0 for A3).
4. Velocity (0.0 to 127.0).
5. Off velocity (0.0 to 127.0).

Time Rescaling:

When saving TAPE data, the timing information is rescaled as if the sequence is playing at a tempo of 120 BPM and a time signature of 4/4. This normalization ensures consistency in timing representation. During loading, the timing is adjusted to match the current BPM and time signature.

Drag and Drop Support in Crescendo

Crescendo offers a convenient drag and drop feature that allows users to load various file types directly into the plugin's interface. This functionality streamlines the workflow, making it easier to integrate different elements into instruments and effects.

Audio Files: Swapping Samples with Ease

Users can drag and drop any media file onto the `SAMPLEUI` elements in the interface. If the file format is not directly supported by Crescendo, it automatically utilizes the FFMPEG library (if installed) to extract the audio. This makes it straightforward to swap samples without manually browsing through file directories.

Tuning System Files: Applying Custom Temperaments

To apply custom temperaments, users can drag and drop SCALA (`.scl`) and TUN (`.tun`) files directly onto the Crescendo interface. The plugin will then load the tuning system defined in the file and make it the active temperament.

Instrument and SoundFont Files: Direct Loading

For quickly loading instruments and sounds, users can drag and drop instrument files (`.txt`) and SoundFont files (`.sf2`) directly onto the Crescendo interface. This bypasses the need to use the "Browse" button, making it easier to switch between instruments or experiment with different sounds.

Enhanced User Experience

The drag and drop functionality in Crescendo significantly enhances the user experience by:

- **Speeding up Workflow:** It eliminates the need for manual file browsing and selection, allowing users to quickly load and apply different elements.
- **Improving Intuitiveness:** It makes the interface more interactive and user-friendly, especially for tasks like sample swapping and temperament loading.
- **Encouraging Experimentation:** The ease of loading different files encourages users to explore various sound design possibilities.

It's important to note that the drag and drop functionality requires that the Crescendo GUI is visible. This requires the host DAW's support for this feature.

Instrument Files

Instrument File Syntax in Crescendo

Here it is a detailed explanation of the syntax used in Crescendo instrument files. These files, typically with a `.txt` extension, utilize a structured programming language to define the behavior and characteristics of instruments and audio effects.

Encoding: ANSI and UTF-8 Support

Crescendo supports both ANSI and UTF-8 encodings for instrument files, accommodating a broader range of characters, including those from languages requiring extended character sets. However, identifiers (variable, keyword, label and function names) must still use ANSI-compatible characters. Strings and comments within the file can utilize extended characters, offering flexibility for documentation and code clarity.

Byte Order Mark (BOM) Handling: Crescendo ignores the BOM, if present in the file.

Other UTF encodings: Crescendo does not support other UTF encodings.

Comments: Annotating and Explaining Code

Comments are an essential part of any programming language, and Crescendo offers multiple ways to include them in instrument files.

- **Line Comments:** Any line starting with a non-alphabetic character (excluding whitespaces) is considered a comment. This includes symbols like `#`, `//`, `!`, `@`, `%`, and numerical characters. These comments are ignored by the script interpreter.
- **Trailing Comments:** Developers can add comments to the end of a line using C++, Matlab or Bash-style syntax (`//`, `%` or `#`) . Example: `Foo = BAR // Comment`.
- **Multiline Comments:** While there's no dedicated syntax for multiline comments, you can achieve this by starting each line with a comment character.

Whitespace: Enhancing Readability

Whitespace characters, like spaces and tabs, are handled flexibly in Crescendo instrument files. Leading and trailing spaces are ignored, allowing for indentation. Multiple consecutive spaces, tabs, or separators are treated as a single element. This flexibility promotes clean and readable code formatting.

Multiline Instructions: Using the Slash Character

Crescendo allows instructions to span multiple lines for better readability. To achieve this, simply terminate intermediate rows with a single slash (`/`). You can add multiple trailing characters with ASCII codes less than 33 after the slash, and these will be ignored.

Important Considerations:

- The current row length limit is fixed at 8192 characters.
- Each row can contain at most one instruction.
- **Raw Text Merging:** When splitting instructions across multiple lines, be mindful that the merging happens on the raw text level before processing. For instance, a comment ending with a slash can unintentionally merge with the next line. While it's technically possible to split keywords, numbers, and strings across lines, it's generally not advisable for code readability.

Crescendo's Instrument File Structure: A Hierarchical Approach

The INCLUDE Instruction: Integrating External Files

A key element of this structure is the `INCLUDE` instruction, which allows you to incorporate the contents of other text files into your main instrument file. This feature promotes modularity and code reuse, making it easier to manage complex instruments and maintain consistency across multiple projects.

Syntax and Usage

The syntax for the `INCLUDE` instruction is:

```
INCLUDE "filename"
```

- `"filename"` represents the name of the file you want to include. This can be either a relative path (relative to the location of the main instrument file) or an absolute path.

Location and Restrictions

- The included files should ideally reside in the `<My Documents>\Crescendo` directory, but relative and full paths are supported, offering flexibility in file organization.
- There is a limit of 20 on the nesting depth of included files to prevent recursive calls and potential infinite loops.

Benefits of Using INCLUDE

- **Modularity:** Break down complex instrument definitions into smaller, more manageable components that can be reused across multiple projects.
- **Code Reuse:** Create libraries of common settings, functions, or sound definitions that can be easily incorporated into different instruments.
- **Consistency:** Ensure that certain settings or configurations are applied consistently across multiple instruments by including a common settings file.
- **Organization:** Improve the readability and maintainability of your code by separating different aspects of the instrument definition into separate files.

Example Scenario

Imagine you have a set of common filter definitions that you want to use in multiple instruments. You could create a file named `filters.txt` containing those definitions and then use the `INCLUDE` instruction in your main instrument file to import them:

```
// Main Instrument File
INCLUDE "filters.txt"

// Rest of the instrument definition
```

Key Considerations

- **Order of Processing:** Instructions in included files are processed as if they were part of the main instrument file at the point of inclusion.
- **Variable Scope:** Variables defined in included files are typically accessible within the scope of the main instrument file, allowing for data sharing and communication between different components.

Overall File Structure

A typical Crescendo instrument file, after merging any included files, might have the following structure:

```
// File Start
// Common Section Rows (Global settings, MIDI processing, etc.)
...
POST     // Optional (Post-processing section)
// Post Section Expressions
...
LAYER    // First Layer Delimiter (Optional)
// Layer 1 Declaration Rows
...
LAYER
// Layer 2 Declaration Rows
...
LAYER
// Layer n Declaration Rows
...
// End File
```

This structure, combined with the `INCLUDE` instruction, allows for a hierarchical and modular approach to defining instruments and audio effects in Crescendo.

Key Syntax Elements

Here it is the core syntax elements that make up Crescendo's instrument file language.

- **Instructions:** Instructions are commands or statements that tell Crescendo what to do. They typically start with a keyword, followed by parameters separated by commas or semicolons. Examples of instructions include `SAMPLE`, `VSTVARS,` and `POLY`.
- **Expressions:** Expressions combine constants, variables, operators, keywords, and functions to produce a single value. They are used to control various aspects of sound generation and processing.
- **Variables:** Variables store values that can be accessed and manipulated within the instrument file. There are different types of variables:
- **Assignments:** Assignments are the core instructions for sound production and processing. They are in the form of `<variable> = <expression>`, optionally preceded by the HOLD or LAST keyword or an EXECIF clause, all on the same line. The `<variable>` can be any user defined variable or any output variable (`OUTnn` or `SENDS<i>`). VST variables,

Keywords, input variables and MIDI CCs cannot be modified in this way. The `<expression>` can be any valid Crescendo expression.
- **The IF ... GOTO/EXIT statement**.
- **A LABEL declaration** for IF ... GOTO targets.
- **The POST keyword** to separate the COMMON section from the optional POST processing section.
- **The LAYER keyword** to separate layers and the first layer from the COMMON or POST section.

Variables in Crescendo

Variable Names

Variable names in Crescendo must adhere to specific rules:

- **Start with a Letter:** The first character of a variable name must be a letter.
- **Length Limit:** Variable names cannot exceed 31 characters.
- **Allowed Characters:** Variable names can include letters (A-Z), numbers (0-9), and underscores (_).
- **Keyword conflicts**: Using reserved keywords or function names on the left of an equal sign is an error: it will be signaled in the log and the whole line will be ignored.
- **Case Insensitivity:** Crescendo treats variable names as case-insensitive. For example, `OscFreq`, `oscFreq`, and `OSCFREQ` all refer to the same variable.

Variable Usage

Variables play a crucial role in Crescendo's programming language. You can use variables to:

- **Store Values:** Variables act as containers for holding values that can be accessed and manipulated throughout the instrument file.
- **Automate Parameters:** You can assign expressions to variables, allowing you to control parameters dynamically based on MIDI CCs, VST variables, or other factors.
- **Perform Calculations:** You can use variables in mathematical and logical expressions to perform calculations, modify signals, and create intricate audio behaviors.
- **Store Intermediate Results:** Variables can hold temporary results of calculations, making your code more readable and efficient.

Types of Variables

Crescendo utilizes different types of variables:

- **User Variables:** Standard user defined variables, for storing intermediate values, much like a conventional programming language.
- **VST Variables:** These are user-defined parameters exposed to the host DAW's interface. You declare them using the `VSTVARS` instruction to specify the total number of VST variables and the `VSTVAR` instruction to define individual VST variables.
- **Internal Variables:** Crescendo relies on numerous internal variables to track various aspects of the instrument's state and behavior. These include:

    - **MIDI CC Values:** These are accessed using the `MCC`, `MCC2` and `MCC3` functions, representing the values of standard and extended MIDI CC messages received by the plugin.
    - **Keyswitch Values:** Accessed through extended MIDI CC numbers, keyswitch values correspond to the states of keyswitches defined using the `KEYSWITCH` instruction.
    - **Input and Output Levels:** Variables like `INnn` represent audio input levels from various input channels, while `OUTnn` determine the audio output levels for respective output channels.

Scope and Accessibility

The scope of a variable determines where it can be accessed and modified.

- **COMMON Section:** Variables declared in the COMMON section are global, accessible from both the POST section and all LAYER sections.
- **POST Section:** Variables declared within the POST section are local to that section and cannot be accessed from LAYER sections.
- **LAYER Sections:** Variables declared within a LAYER section are local to that layer and cannot be accessed from other LAYER sections or the POST section.

It's important to note that expressions and assignments within the COMMON section are typically "replicated" in the POST and LAYER sections when the variable is used. This means that the calculations defined in the COMMON section are essentially imported into the relevant sections where the variable is referenced.

Variable Declaration and Initialization

There are no explicit variable declaration instructions in Crescendo. You implicitly declare a variable by using it on the left side of an assignment operator (`=`).

For example:

```
OscFreq = 440
```

This line implicitly declares a variable named `OscFreq` and assigns it an initial value of `440`.

Further considerations on Variables

Each parameter of an oscillator, an envelope, a filter or other functions can be a constant or an expression or a plain variable.
Constants and plain variables don't need to be calculated, so they do not consume CPU resources (variables consumed CPU resources only when they were first calculated).
Other complex expressions need result slots and operation slots, so for clarity a variable can be used for temporary storage and referenced in the parameter definition.
In this case no penalty is incurred, because in any case the compiler would have produced the same code.
The advantage is that if assigned to a variable, the expression can be reused without recalculate it, but common subexpression optimization is not performed by the current implementation.

Example:
```
FOO=FILT(0;BAR;<complex expression1>;<complex expression2>)
```

is equivalent to the clearer version below:

```
VARfreq=<complex expression1>
VARres=<complex expression2>
FOO=FILT(0;BAR;VARfreq;VARres)
```

Declaring a variable in the common section does not automatically imply that it is always calculated.
If a layer or the post section is not using it, the compiler does not let it execute for that particular layer or section.
So common subexpression or filters, oscillators, envelopes are better put in the header, among other global things.
Exception at this algorithm is given in the IF ... GOTO section below.

In the POST section are imported only COMMON instructions that DO NOT write the OUTnn variables.
Variables that use and process OUTnn variables ARE imported, but obviously the OUTnn value is different within a POST section.
To avoid confusion, just avoid putting expressions using the OUTnn variables on the right in the COMMON section.

Expressions in Crescendo

Expressions are fundamental to the plugin's functionality, enabling users to define complex relationships between parameters, control signals, and audio processing operations.

Composition and Functionality

Expressions in Crescendo are formed by combining various elements:

- **Constants:** Fixed numerical values, always represented as 32-bit floating-point numbers. For instance, `10`, `0.5`, and `440` are examples of constants.
- **Variables:** Identifiers representing values that can change during the execution of the instrument file. These can include user-defined variables, internal variables like `KEY`, `ONVEL`, and `OUT`, as well as MIDI CC values accessed through `MCC`, `MCC2` and `MCC3` functions.
- **Keywords:** Reserved words with specific meanings and functionalities within the language. Examples include `KEY`, `ONVEL`, `OUT`, `SENDS<i>`.
- **Operators:** Symbols used for mathematical operations. Crescendo supports standard mathematical operators: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `^` (exponentiation).
- **Functions:** Built-in operations that perform specific tasks. These include functions for oscillator generation (`OSCG`), envelope creation (`ENV`), filter application (`FILT`), delay effects (`DELAYx`), and various mathematical and transcendental operations.

Expressions can be used for a variety of purposes:

- **Automating Parameters:** By assigning an expression to a variable, you can dynamically control a parameter based on MIDI CC values, VST variables, or other factors. This enables real-time manipulation and modulation. You can directly put the expression in place of the parameter, but using a variable can increase code readability and do not incur in performance penality. Also the variable value can be reused without any cost and is the best solution to reuse common subexpression as the current implementation does not perform this optimization.
- Calculate an intermediate result to be stored in a variable and used later, possibly multiple times.
- Accumulate the signal in one of the SENDS<i> channel (only in the LAYERs).
- Calculate the final sound output of the LAYER (or the whole VST plugin if it is in the POST section).
- **Signal Processing:** Expressions can combine, filter, and process audio signals, allowing you to create a wide range of sound effects.
- **Conditional Logic:** The `EXECIF` keyword provided conditional execution, making it possible to create dynamic behavior based on specific conditions. It requires an assignment to be conditionally executed.

Using Expressions in conditional Logic: The `IF...GOTO/EXIT` construct supports, variables, keywords or a constant for the comparison. To compare one or two expressions, first assign them to some dummy variables.

Example Expressions

Here are some examples of expressions and their functionality:

1. **`OscFreq * 2^(PitchBend/12)`:** This expression calculates the oscillator frequency, dynamically adjusting it based on the current pitch bend value.
2. **`Gain * ENV(0.1, 0.2, 0.8, 0.5)`:** This expression multiplies the gain by the output of an envelope with specific attack, decay, sustain, and release times.
3. **`OUT = OSCG("Sgv00PS", 1, Freq, Phase, 0.1, 0.2, 0.8, 0.5)`:** This expression assigns the output of a slotted oscillator to the `OUT` variable, using specific parameters for frequency, phase, and envelope.

Key Considerations

When working with expressions, keep in mind:

- Expressions can be used to modify or create a variable on the left of an equal sign, much like a normal programming language.
- The expression is evaluated for each sample (if not prefixed with the HOLD keyword).
- **Order of Operations:** Mathematical operators follow standard precedence rules. Use parentheses to control the order of evaluation.
- **Variable Scope:** The scope of variables determines where they can be accessed and modified. Variables declared in the `COMMON` section are global, while those declared within `POST` or `LAYER` sections are local to those sections.
- **Dead Values:** The compiler optimizes code by not allowing the execution of instructions that result in "dead values," meaning values that do not contribute to the final output. This can happen when a global expression, that is automatically imported in all layers and in the POST step, is not actually used. Also expressions after the last OUT assignment are ignored, since do not contribute to any output.

Automation data as Keywords or MIDI CCs

In Crescendo all non user variable data is accessible with keywords or MIDI CC number. User variable data is accessible only with the variable name.

Crescendo utilizes several internal variables represented by keywords. These keywords grant direct access to essential information and parameters within the plugin. You can always access them also with their MIDI CC number. Some examples:

- **`KEY` or `KEYI`:** Represents the MIDI note number of the currently played note, without temperament correction. It's an integer value between 0 and 127.
- **`KEYF`:** Represents the actual note being played, including temperament correction. It can have a fractional part, indicating cents of detuning from equal temperament.
- **`ONVEL`:** Represents the note-on velocity of the triggering note, ranging from 0 to 127.
- **`OFFVEL`:** Represents the note-off velocity, also ranging from 0 to 127. It might be unavailable in some cases, defaulting to the `ONVEL` value.
- **`PROGRAM`:** Holds the currently selected MIDI program number, an integer between 0 and 127.
- **`BANK`:** Contains the currently selected MIDI bank number, an integer between 0 and 16383.
- **`SAMPLERATE` or `SAMPLEFREQ`:** Represents the current sample rate, which can be useful in custom filter designs.
- **`BPM`:** Holds the current beats per minute value.
- **`NUM` and `DEN`:** Represent the numerator and denominator of the current time signature.
- **`WHEEL` or `PBEND`:** Represents the current pitch bend value, a floating-point number between -1 and +1.
- **`TEMPERAMENT`:** Indicates the temperament currently in effect.
- **`IN`:** Represents the audio signal coming from VST input #2 (stereo), often used for sidechaining.
- **`IN0-IN99`:** Represent audio signals from additional VST inputs (#3 to #101, stereo).
- **`OUT` and `OUT0-OUT99`:** Represent the output signals of the instrument. `OUT` corresponds to output #1, while `OUT1` to `OUT99` correspond to outputs #2 to #100.
- **`SENDS1-SENDS4`:** These variables are used to pass audio signals from layers to the POST processing stage.

The MIDI CC number space is divided into blocks. These are the main blocks available:

- **Standard MIDI CCs (0-127):** These represent typical MIDI control messages, like modulation wheel, volume, pan, and expression.
- **Extended MIDI CCs (128-199):** Crescendo extends the standard range to include parameters like pitch bend, program change, aftertouch, and various internal variables.
- **Polifonic aftertouch (200-327):** Aftertouch for each key of the MIDI keyboard.
- **Keyswitch Values (400-527):** Each keyswitch is assigned an extended MIDI CC number, allowing for keyswitch-based triggering and control.
- **VST Variables (600-727):** Every VST variable, exposed as a parameter on the plugin's interface, has a corresponding extended MIDI CC number for access and manipulation within the instrument file.

For the full list of MIDI CC keywords and numbering see the dedicated chapter.

Multi channel and layer considerations

MIDI CCs and other parameters can have different values in different contexts.

The first discriminant is the default channel associated to a LAYER or the POST step.

This channel is used to access the right slot when accessing multichannel MIDI CCs.

There is also per LAYER data, see below.

When a LAYER is triggered, the channel number from which the NOTE ON (and NOTE OFF) message arrives is assigned to it. This channel is used to determine from which channel the NOTE OFF message must arrive for the trigger of the release stage and from which channel all triggers, EXECIF and MIDI CC values should be taken.

For more control there is the MCC3 instruction to access every channel of every MIDI CC.

POST step: It is global by design. By default the channel chosen for multichannel data is the channel 0, but there is the POSTCH instruction to set a different default channel for the POST step. You can always use MCC3 to access a single MIDI CC from a specific channel.
If each channel should have separate effects, you should use different VST instances and split the channels with MIDICH instruction (see below).

In the DAW project file the last program and bank for each channel is saved. For Crescendo versions before the 1.0.135 only the channel 0 values are stored. Just save again the DAW project to resolve the issue.

The textual program and bank number and names, enabled with some values of the HIDEUI option are relative to the channel 0. To show all channels program and bank information, use the option 10 of HIDEUI (or in some cases the automatic options, see HIDEUI for details).

Key considerations:

- There are local, per channel and global MIDI CCs (see below).
- Most global automations have the same formula for all LAYERs, but the value of the per channel or local MIDI CC is different, so it is the result.
- Constants and global MIDI CCs have obviously the same values for all contexts.

Accessing MIDI CCs and more: The `MCC`, `MCC2` and `MCC3` Keywords

Given a MIDI CC number, the user can access its value with three functions:

- The `MCC` function: It retrieves the smoothed raw, unscaled value of the specified control parameter. This function is the fastest to retrieve a MIDI CC value as usually does not consume any CPU time, since it is translated to a MIDI CC number in the operation using it.
  - **Syntax:** `MCC(<num>)`
  - **Parameter:** `<num>` is a numeric constant representing the MIDI CC number or control parameter identifier.
- The `MCC2` function: It provides a more refined approach, offering scaling and smoothing for continuous controllers. The output is scaled to a range of 0 to 1, making it ideal for modulation purposes. Moreover the MIDI CC number is automatable with an expression.
  - **Syntax:** `MCC2(<expression>)`

    - **Parameter:** `<expression>` can be a numeric constant or a more complex expression, allowing for dynamic specification of the control parameter.
- The `MCC3` function: It is similar to `MCC2`, but provides another parameter to access a custom MIDI channel. The MIDI CC number and channel are automatable with an expression.
    - **Syntax:** `MCC3(<expression>,<channel>)`
    - **Parameters:**
        - `<expression>` can be a numeric constant or a more complex expression, allowing for dynamic specification of the control parameter.
        - `<channel>` can be a numeric constant or a more complex expression, allowing for dynamic specification of the channel number.

MIDI CCs classes

Crescendo is multichannel capable, so there is the need to subdivide the data available in some classes:

1) **MIDI CC with different value per Layer (POST step excluded)**: each layer has its own value, different from other layers. `MCC, MCC2, MCC3` and Keywords access the same value in a given layer, but possibly different than that in other layers. In `MCC3` the channel parameter is thus ignored. Most of these MIDI CC are not available in the POST step. Exception are the `OUTnn` that are treated differently.
   They are: `KEY/KEYI, KEYF, ONVEL, OFFVEL, RANDOM, GAIN, FREQ, TIME, DEFAULTFC, DURATION, GAIN0, FREQ0, OUTnn`.
2) **MIDI CC with different value per channel**: each channel has its value. `MCC`, `MCC2` and Keywords access to the value of the default channel (see above for a definition of default channel). `MCC3` allows to specify a specific channel.
   They are: Standard MIDI CC (0-127), `POLYAFT` (200-327), `AFTERTOUCH`, `PROGRAM`, `WHEEL/PBEND`, `BANK`, User defined MIDI CCs (160-199).
3) **Global MIDI CC**: each layer (POST step included) accesses always the same value, so `MCC, MCC2, MCC3` and Keywords give the same result in each layer.
   They are: KEYSWITCHES (400-527), VST VARS (600-727), `INnn, SENDS, BPM, NUM, DEN, TEMPERAMENT, SAMPLERATE/SAMPLEFREQ`.
4) **The OUTnn keywords** are treated differently in LAYERs and POST step (see below).
5) **User Variables**: the values are local to each LAYER and the POST step. If the expression is defined in the COMMON step, this can be shared by the LAYERs and the POST step that use that variable, but the vales used to evaluate the expression (in particular Keywords and MIDI CCs) will be the ones relative to the entity that is using the expression (see points 1 and 2 above).

The LABEL Instruction and Its Use in Crescendo's IF...GOTO/EXIT Construct

The `LABEL` instruction, combined with the `IF...GOTO/EXIT` statement, empowers developers to introduce conditional branching and loops, adding dynamism and responsiveness to virtual instruments and audio effects.

Defining Jump Points with `LABEL`

The `LABEL` instruction acts as a marker within the code, establishing a named location to which the execution flow can be redirected using the `GOTO` component of the `IF...GOTO/EXIT` statement.

Syntax and Placement:

- The `LABEL` instruction must occupy a single row in the instrument file.
- It takes the following form: `LABEL <label_name>`
- It precedes the expression or `IF...GOTO/EXIT` statement to which it's associated.

Example:

```
LABEL SquareWave
// Code for generating a square wave oscillator
```

In this example, the `LABEL` instruction with the name "SquareWave" marks the location of the code block responsible for generating a square wave oscillator.

Characteristics of Labels

There are specific rules governing the definition and usage of labels:

- **Case Insensitivity:** Labels are case-insensitive, meaning "SquareWave" and "squarewave" refer to the same location.
- **Character Constraints:** Labels can be up to 31 characters long and can consist of alphanumeric characters (A-Z, 0-9) and underscores. Importantly, labels can start with a number, unlike variable names. This allows for BASIC/FORTRAN-style numeric labels.
- **Scope:** The scope of labels is localized to either a `LAYER` or `POST` section. This means a label defined within a `LAYER` is not accessible from the `POST` section or another `LAYER`. However, labels declared in the `COMMON` section are inherited by both `POST` and `LAYER` sections.
- **Redefinition Errors:** Attempting to redefine a label within the same scope will result in an error, ensuring clarity and preventing unintended jumps.

The IF...GOTO/EXIT Statement: Conditional Branching

The `IF...GOTO/EXIT` statement evaluates a condition and, based on its truth value, either redirects the execution flow using `GOTO` or exits the current section using `EXIT`.

Syntax:

- `IF <op1> <comp> <op2> GOTO <label_name>`
- `IF <op1> <comp> <op2> EXIT`

Operands and Comparisons:

- `<op1>` and `<op2>` represent the values being compared. They can be constants, variables, keywords, or the `MCC(#)` function to access MIDI CC values. However, complex expressions are not allowed, including `MCC2` and `MCC3`; use a dummy variable if needed.
- `<comp>` stands for the comparison operator: `=`, `>`, `<`, `>=`, `<=`, `<>`, or `!=`.

Functionality:

- If the condition evaluates to true, the `GOTO` statement directs the execution flow to the instruction following the specified `LABEL`.
- If the condition is false, the execution proceeds to the next instruction in the sequence.
- The `EXIT` statement, when the condition is true, terminates the execution of the current `LAYER` or `POST` section.

Example:

```
IF MCC(64) > 63 GOTO SquareWave
// Code for a sine wave oscillator
LABEL SquareWave
// Code for a square wave oscillator
```

This code snippet checks if MIDI CC 64 (often associated with the sustain pedal) is greater than 63. If true, it jumps to the "SquareWave" label, selecting the square wave oscillator. If false, it continues with the sine wave oscillator.

Considerations When Using IF...GOTO/EXIT

- **Single Condition:** The `IF...GOTO/EXIT` statement handles only one condition at a time, lacking support for logical operators (`AND`, `OR`) to combine multiple conditions.
- **Memory Elements:** Skipping or re-executing instructions that involve memory elements (like delays or filters) can produce unexpected behavior.

EXECIF: Conditional Execution at Trigger Time

The **EXECIF** construct in Crescendo's programming language offers a powerful mechanism for **selective execution of code** based on conditions evaluated at the moment a layer is triggered, also known as **trigger time**. This allows for dynamic behavior and nuanced sound design, enabling you to create instruments that adapt to different playing styles and MIDI input. Unlike the **IF...GOTO/EXIT** statement, which performs conditional checks on a sample-by-sample basis, **EXECIF** makes decisions only once, when a note is triggered, making it more efficient for certain tasks.

Syntax and Variations

The **EXECIF** construct has three primary forms, as detailed in source:

1. Condition Based on MIDI CC Value:

```
EXECIF <mcc>,<min>,<max> <var> = <expression>
```

This form executes the assignment **`<var> = <expression>`** only if the MIDI CC number specified by **`<mcc>`** has a value between **`<min>`** and **`<max>`** (inclusive) **at the trigger time**.

2. Condition Based on Key and Velocity:

```
EXECIF <minkey>,<maxkey>,<minvel>,<maxvel> <var> = <expression>
```

This form executes the assignment if the triggered note's **key number** falls between **`<minkey>`** and **`<maxkey>`** and its **velocity** is between **`<minvel>`** and **`<maxvel>`**.

3. Combined Condition:

```
EXECIF <minkey>,<maxkey>,<minvel>,<maxvel>,<mcc>,<min>,<max> <var> =
<expression>
```

This form combines the previous two, executing the assignment only if **all the specified conditions** are met at trigger time.

Functionality and Applications

The **EXECIF** construct excels in situations where you need to:

- **Select Samples:** Choose different audio samples based on the triggered note's pitch, velocity, or the state of a particular MIDI CC.
- **Adjust Gain:** Modify the gain of a layer depending on the trigger conditions, allowing for dynamic volume control.
- **Apply Effects:** Trigger specific effects only when certain conditions are met, creating more expressive and interactive instruments.
- **Combine Similar Layers:** Streamline code by merging layers that differ only in subtle nuances, using **EXECIF** to select the appropriate instructions at runtime.

Key Considerations

- **Undefined Variables:** If an **EXECIF** condition is not met, the associated variable remains undefined. You need to handle such cases carefully to avoid unexpected behavior.
- **Efficiency: EXECIF** improves performance by avoiding continuous sample-by-sample checks, making it more efficient than **IF...GOTO/EXIT** for trigger-time decisions.

Examples

Here are some illustrative examples of how **EXECIF** can be used:

```
// Selecting different samples based on key range
EXECIF 36, 47, 0, 127 Sample = 1
EXECIF 48, 60, 0, 127 Sample = 2
EXECIF 61, 71, 0, 127 Sample = 3
```

This code snippet selects different samples (1, 2, or 3) based on the pitch of the triggered note.

```
// Adjusting gain based on MIDI CC value
EXECIF 1, 0, 63 Gain = Gain * 0.8
```

If MIDI CC 1 is between 0 and 63 at trigger time, the gain is reduced by 20%.

The COMMON Section

The **COMMON** section acts as the **global control center** for your Crescendo instrument. It's the **first and mandatory section** in any instrument file, and it defines the overall behavior and characteristics of your instrument or audio effect. Think of it as laying the groundwork for everything that follows.

Key Roles of the COMMON Section:

- **Global Definitions and Initializations:** This section is where you **declare essential elements** that are used throughout your instrument. These declarations might include:
  - **`SAMPLE`:** Declares sample slots, specifying the file path, root note, and other properties for the audio waveforms used by your oscillators.
  - **`VSTVARS`:** Declares the number of VST variables (parameters) exposed to your DAW, enabling control over the instrument from the DAW's interface.
  - **`INTERFACE`:** Specifies the dimensions of the instrument's user interface.
- **MIDI Processing Instructions:** The **COMMON** section houses key instructions related to how incoming MIDI data is handled before it reaches the sound-generating layers. This includes:
  - **`MIDICH`:** Filters incoming MIDI messages based on their channel, allowing you to process only the desired MIDI channels.
  - **`MIDICC`:** Initializes MIDI CC values and optionally links them to VST variables, mapping controller movements to parameter changes.
  - **`SEQUENCER`:** Activates and configures Crescendo's built-in sequencer, enabling the generation of MIDI note data based on patterns and sequences.
  - **`MAP`, `VELOCITY`, `PITCH`, `TIMING`:** These instructions provide fine-grained control over the transformation and manipulation of MIDI note data, such as remapping notes, adjusting velocities, transposing pitches, and altering note timing.
  - **`TEMPERAMENTCC`:** Links a MIDI CC to the temperament selection, allowing dynamic control over the instrument's tuning system.
- **Global Expressions and Calculations:** You can define expressions and calculations in the **COMMON** section that are used throughout your instrument. This is useful for:
  - **Shared Settings:** Establishing default values for parameters that might be used by multiple layers.
  - **Complex Automations:** Creating intricate modulation patterns or control signals that influence multiple parts of the instrument.
- **Inclusion of External Files:** The **COMMON** section can include external text files (using the `INCLUDE` instruction), enabling you to organize your instrument code into modular components or reuse common settings across multiple instruments.

Delimiter: The **COMMON** section ends when the **`POST`** or **`LAYER`** keyword is encountered.

Initialization Instructions in Crescendo

- **INTERFACE**: This instruction defines the initial size of the plugin's graphical user interface (GUI) window in pixels. The dimensions specified assume a default font size of 16 DLPs (8 points) and a standard monitor DPI of 96. Crescendo automatically scales these values if the actual font size or DPI is different. This instruction can be placed in the `Settings.ini` file or within the instrument file.
- **QUALITY**: This instruction controls the quality of the sample interpolation used in Crescendo. It allows adjustment of parameters like the number of samples in the interpolation window, window width, interpolation type, and oversampling factor. This instruction is usually placed in the `Settings.ini` file but can also be specified in the instrument file.
- **VSTVARS**: This instruction declares the number of VST variables (parameters) that will be exposed to the host DAW. Valid values range from 1 to 128. The default value is 20. This instruction is crucial for defining how many control parameters the user can access in the DAW's interface.
  - It's important to note that some DAWs might have limitations on the number of VST parameters they support. For example, Ableton Live has a maximum of 64

    parameters. While higher-numbered VSTVARS can still be used within the instrument file, they might not be accessible for automation or manual adjustment in the DAW GUI.

- **IO**: This instruction configures the number of audio inputs and outputs for the VST plugin. The minimum number of inputs is 2, and the maximum is 101. The minimum number of outputs is 1, and the maximum is 100. The default values are 2 inputs and 1 output. It is recommended to include the IO instruction in the `Settings.ini` file for optimal compatibility across different DAWs, especially those that might not allow changes to these parameters once a plugin is loaded.
- **DPIAWARE**: This instruction forces DPI awareness in the host process if the provided flag is not zero. This action is performed only when the VST plugin is loaded. However, using this instruction might not be compatible with all DAWs and should be used cautiously.

Sample Slots: The Core of Audio Data Storage

Crescendo employs the concept of "sample slots," numbered containers in memory, to store audio data that will be used to produce sound. These slots are dynamically allocated, meaning they are created and assigned numbers as needed based on the highest slot number referenced in any of the relevant instructions (`SAMPLE`, `SAMPLEUI`, `SAMPLES`, or `RENDER`).

Think of them as shelves in a library, each holding a different sound. You can fill these shelves either by bringing in sounds from external files or by creating your own sounds directly within Crescendo.

`SAMPLE`: Defining Single Waveforms

The `SAMPLE` instruction serves as the primary means of filling these sample slots. It offers two distinct approaches:

1. **Loading from an External Audio File:** This is the most common way to use `SAMPLE`. You specify the path to an audio file on your computer and Crescendo loads that audio data into the designated sample slot.
    - Beyond just loading the file, you have granular control over how the audio is handled:
        - **Normalization Mode:** This determines how the audio's volume is adjusted, ensuring it doesn't clip or sound too quiet.
        - **Center Note:** This crucial parameter tells Crescendo what musical note the sample corresponds to, ensuring it plays back at the correct pitch.
        - **Looping:** You can define whether the sample should play once (one-shot) or loop continuously, making it suitable for sustained sounds.
        - **Release Samples:** Some instruments have distinct sounds for the sustain and release portions of a note. The `SAMPLE` instruction allows you to define a separate release portion within a single audio file, simplifying sample management.
2. **Synthesizing Waveforms from Harmonic Data:** `SAMPLE` also empowers you to create sounds directly within Crescendo without relying on external files. You do this by defining a set of harmonics.
    - A **harmonic** is a frequency that is a multiple of a fundamental frequency. By combining harmonics with different amplitudes and phases, you can create a wide variety of waveforms.
    - The `SAMPLE` instruction allows you to specify up to 32 harmonics, each with its own:

    - **Type:** You can choose from basic waveforms like sine, square, sawtooth, triangle, or noise.
    - **Frequency:** This determines the pitch of each harmonic.
    - **Amplitude:** This controls the volume of each harmonic.
    - **Phase:** This affects the timing of each harmonic within the waveform.

Example:

```
SAMPLE 200, "samples\Piano_C4.wav", 4, 60
```

This instruction would load the audio file "Piano_C4.wav" into sample slot 200. The normalization mode is set to 4, and the center note is 60 (which corresponds to middle C).

`SAMPLES`: Combining Multiple Samples

The `SAMPLES` instruction is used to create a new sample by mixing multiple audio files together. This is particularly useful for creating:

- **Complex Instruments:** Imagine a piano sound that combines samples from different velocity layers to capture the instrument's dynamic range.
- **Drum Kits:** You could use `SAMPLES` to combine individual drum hits into a single sample, allowing you to trigger an entire drum kit from a single sample slot.

Key Features:

- **Mixing:** `SAMPLES` loads the specified audio files, resamples them to a common sample rate and center note, and then mixes them together into a new sample.
- **Scaling and Panning:** You have individual control over the volume (scaling) and stereo position (panning) of each sample within the mix.
- **Normalization:** You can normalize the final combined sample to prevent clipping and ensure a consistent volume.

Example:

```
SAMPLES 100, 2, 2, 44100, 60, 4, "samples\Kick.wav", 1, 1, 36,
"samples\Snare.wav", 0.8, 0.8, 38
```

This instruction creates a stereo sample in slot 100 by mixing a kick drum sample ("Kick.wav", centered at note 36) and a snare drum sample ("Snare.wav", centered at note 38). The final sample has a duration of 2 seconds, a sample rate of 44100 Hz, and is normalized using mode 4.

`RENDER`: Dynamic Sample Generation and Processing

The `RENDER` instruction takes sample manipulation a step further. Instead of simply loading or mixing existing samples, it dynamically generates a new sample by processing one or more pre-defined sample slots. This opens up a wide range of creative possibilities.

Key Capabilities:

- **Combining and Resampling:** Similar to `SAMPLES`, `RENDER` can combine multiple sample slots into a single output sample. However, it offers more control over resampling, allowing you to precisely adjust the pitch and playback speed of each input sample.
- **Envelope Application:** `RENDER` allows you to apply an ADSR envelope to each input sample, shaping its volume over time within the final rendered sample. This is incredibly valuable for creating dynamic sounds, such as drum hits with custom attack and decay characteristics.
- **Layering and Complex Sound Design:** The combination of sample mixing, resampling, and envelope control makes `RENDER` ideal for creating intricate sounds. You can use it to layer sounds, create complex textures, or even emulate the behavior of physical instruments with evolving timbres.

Example:

```
RENDER 300, 1, 2, 44100, 60, 1, 200, 1, 1, 60, 0.01, 0.2, 0.8, 0.5, 0.3, 201,
0.5, 0.5, 64, 0.05, 0.1, 0, 0.8, 0.2
```

This instruction generates a one-second stereo sample in slot 300 by combining and processing the contents of sample slots 200 and 201. Each input sample is given its own scaling, center note, and ADSR envelope, allowing for precise control over the final sound.

Prefiltering Sample Slots in Crescendo

Here it is how you can prefilter sample slots in Crescendo, allowing for fixed filtering operations on samples to be performed during the sample loading process, as opposed to during real-time audio processing. This can potentially save valuable CPU resources, especially when dealing with filters that would normally be applied in real-time for every note played.

`FILTER, FILTERSEMI, EQ1DB`, `EQ3D` and `WIDEPAN`: Prefixes for Sample Pre-filtering

These instructions act as prefixes to the `SAMPLE`, `SAMPLES`, or `RENDER` instructions. They enable you to apply a filter to the audio data within the sample slot **before** it undergoes normalization. This means the filtering becomes an inherent part of the sample data itself, eliminating the need to apply the same filter in real-time using the corresponding filtering instruction.

Key Points:

- **Prefix Application:** They must be placed **immediately before** the `SAMPLE`, `SAMPLES`, or `RENDER` instruction to which they apply.
- **Sample Data Only:** These prefixes only affect **sampled data**, not synthesized waveforms created using harmonic data.
- **Limited Filter Types:** The types of filters you can apply using these prefixes are restricted to the same set available corresponding instructions.

Considerations and Benefits:

- **Fixed Filtering:** Prefiltering is most effective for filters with fixed parameters that don't need to change during playback. If you need dynamic filter control, use the `FILT` instruction instead.

- **Offline Processing:** The filtering happens during the sample loading stage, so it doesn't impact real-time performance.
- **CPU Savings:** By offloading the filtering to the sample loading phase, you can potentially reduce CPU usage during playback, especially if your instruments use many layers or complex filtering.

Combining Prefilters

You can combine multiple `FILTER`, `FILTERSEMI`, `EQ1DB`, and `EQ3DB` prefixes to create more complex filtering chains. Each additional prefix will apply another filter stage to the sample data. The order of the prefixes determines the order in which the filters are applied.

For example, you could create a 5-band equalizer by using five consecutive `EQ1DB` prefixes before the `SAMPLE` instruction. You can also combine these with the `WIDEPAN` prefix to adjust the stereo width and panning of the sample.

Example:

```
FILTER 0, 100, 0.7 FILTER 1, 10000, 0.3 SAMPLE 200, "samples\NoisyPad.wav", 1,
60
```

In this example, a low-pass filter and a high-pass filter are applied to the "NoisyPad.wav" sample before it's loaded into slot 200, effectively creating a band-pass filter.

Enabling Interactive Sample Swapping: The `SAMPLEUI` Instruction

Sample Slots and Dynamic Sound Switching

Crescendo utilizes numbered "sample slots" in memory to store the audio data used by its oscillators. The `SAMPLE` instruction is used to populate these slots, either with audio loaded from external files or with waveforms synthesized from harmonic data. By changing the sample slot number referenced by an oscillator (e.g., using the `OSCG` function), you effectively change the sound being produced.

The `SAMPLEUI` instruction builds upon this concept by providing a visual and interactive mechanism for users to control this sample switching process.

Implementing `SAMPLEUI`: Creating User-Accessible Sample Browsers

The `SAMPLEUI` instruction creates a dedicated user interface (UI) element within your instrument's interface, typically resembling a small file browser. This UI element allows users to easily browse their file system and select a new audio sample to load into a specific sample slot.

Syntax:

```
SAMPLEUI <Sample_slot>; <X_pos>; <Y_pos>; “<Text>”
```

- **`<Sample_slot>`:** This parameter is crucial. It specifies the numerical ID of the sample slot that will be controlled by this particular `SAMPLEUI` element. Assigning multiple `SAMPLEUI` instructions to the same sample slot can lead to unpredictable behavior, so ensure that each `SAMPLEUI` element controls a unique sample slot.

- **`<X_pos>` and `<Y_pos>`:** These parameters determine the position of the `SAMPLEUI` element within your instrument's interface. They specify the coordinates of the upper left corner of the UI element, allowing you to arrange multiple `SAMPLEUI` elements (if needed) within the available interface space.
- **`"<Text>"`:** This optional parameter allows you to provide a short descriptive text that will appear in the header of the `SAMPLEUI` element. This text serves as a label, helping users understand the purpose or category of sounds that can be loaded using this particular `SAMPLEUI` instance.

Functionality:

Once implemented, the `SAMPLEUI` element typically provides a "Browse" button, allowing users to navigate their file system and select an audio file. Users can also often drag and drop compatible audio files directly onto the `SAMPLEUI` area. This user-friendly approach empowers them to experiment with different sounds and tailor the instrument to their specific needs, enhancing the overall user experience.

Runtime Behavior and Saving User Preferences

When a user interacts with a `SAMPLEUI` element and selects a new sample:

1. **Loading and Updating:** Crescendo loads the chosen audio file into memory and updates the contents of the associated sample slot, effectively replacing any previous audio data stored there.
2. **Bypassing Pre-processing:** It's important to note that if you have any pre-processing steps (like filtering or normalization) defined for the sample slot in your instrument file, these are bypassed when loading a sample through the `SAMPLEUI` interface. The new sample is loaded in its raw, unprocessed form.
3. **Saving User Settings:** Crescendo takes care of saving the last selected file path and settings for each `SAMPLEUI` element. This information is stored as part of the host DAW's project file. When the project is reloaded, the VST instrument automatically restores the user's chosen samples and settings, ensuring their customizations are preserved.

Practical Applications: Enhancing User Experience and Creative Possibilities

The `SAMPLEUI` instruction opens up a wide range of possibilities:

- **Multi-Sampled Instruments:** You can create instruments where users can swap out samples for different velocity layers, articulations, or round-robin variations, providing a much richer and more expressive playing experience.
- **User-Customizable Drum Kits:** Imagine a drum kit instrument where users can build their own kits by loading their favorite kick, snare, hi-hat, and other percussion samples using intuitive `SAMPLEUI` controls.
- **Sound Design Exploration:** For instruments designed for sound design or experimentation, `SAMPLEUI` elements become powerful tools, allowing users to explore a vast range of sounds and discover unique sonic combinations within your instrument.

Example:

```
SAMPLEUI 200; 10; 10; "Kick Drum"
SAMPLEUI 201; 10; 60; "Snare Drum"
```

In this example, two `SAMPLEUI` elements are created. The first controls sample slot 200 and is labeled "Kick Drum," while the second controls slot 201 and is labeled "Snare Drum." Users can then use these elements to load their own kick and snare samples into the instrument.

Understanding Crescendo's `SAMPLEOFF` Instruction

Here it is a comprehensive overview of the `SAMPLEOFF` instruction in Crescendo. This instruction plays a crucial role in managing and selecting samples, especially in multi-sampled instruments, by introducing the concept of an "offset" to the sample slot numbering system.

Sample Slots and the Need for Offsets

Crescendo uses numbered "sample slots" to store the audio waveforms used by its oscillators. The `SAMPLE` instruction is used to load these waveforms into specific slots, either from external audio files or by synthesizing them from harmonic data. Oscillators, like the `OSCG` function, reference these slots by their number to produce sound.

In simple instruments, directly specifying the sample slot number in the `OSCG` function might suffice. However, as instrument complexity increases, especially with multi-sampled instruments where different notes or velocity ranges use different samples, this approach becomes less manageable and less flexible.

The `SAMPLEOFF` instruction addresses this challenge. It allows you to define an offset value that is added to the sample slot number used by oscillators. This offset effectively shifts the sample selection, enabling you to organize your samples more logically and create instruments with multiple sample banks that can be easily switched or automated.

`SAMPLEOFF` Placement and Scope

The location of the `SAMPLEOFF` instruction within your Crescendo instrument file determines its scope of influence:

- **`COMMON` Section:** When placed in the `COMMON` section, `SAMPLEOFF` sets a global default sample offset for the entire instrument. This offset is applied to all layers unless overridden by a `SAMPLEOFF` instruction within a specific layer.
- **`POST` Section or `LAYER`:** If you place `SAMPLEOFF` within a `POST` section or a specific `LAYER`, its effect is limited to that particular section or layer. This localized scope allows for fine-grained control over sample selection within different parts of your instrument.

Reserved Slots: Ensuring Consistency

Crescendo reserves the first 200 sample slots (0-199) and these are **not affected** by the `SAMPLEOFF` offset. This design decision ensures that you have dedicated slots for storing samples that need to remain at a fixed position regardless of the offset. You might use these slots for constant waveforms, such as sine waves used for modulation, or for samples that play a critical role across multiple layers.

Illustrative Examples: Multi-Sampled and Multi-Banked Instruments

Here there are two practical examples demonstrating the power and flexibility of `SAMPLEOFF` in constructing sophisticated instruments:

1. Multi-Sampled Instrument (Note and Velocity-Based Sample Switching)

```
// Define samples for different note and velocity ranges
SAMPLE 200, "samples\C3V0_50.wav", 4, 60     // Sample for C3 velocity 0-50
SAMPLE 201, "samples\C3V51_100.wav", 4, 60    // Sample for C3 velocity 51-100
// ... and so on for other note and velocity ranges

// Define a common instruction, replicated in all LAYERS
OUT = OSCG("Sgf000S", 200, Attack, Decay, Sustain, Release)

// Define layers with specific velocity ranges and sample offsets
LAYER
NOTEON 60, 60, 0, 50       // Note C3, velocity 0 to 50
SAMPLEOFF 0                // Sample offset 0, selecting sample slot 200
LAYER
NOTEON 60, 60, 51, 100      // Note C3, velocity 51 to 100
SAMPLEOFF 1                // Sample offset 1, selecting sample slot 201
// … and so on…
```

In this example:

- Different samples are loaded into slots 200, 201, etc., representing various note and velocity ranges (here is shown only the note C3 (MIDI note 60)).
- The common `OUT` instruction in the `COMMON` section sets up the oscillator (`OSCG`) to use sample slot 200 as the default.
- Within each `LAYER`, the `NOTEON` instruction defines the note and velocity range, and the `SAMPLEOFF` instruction adjusts the offset to select the appropriate sample for that note and velocity range. For example, in the first layer, `SAMPLEOFF 0` ensures that the oscillator uses slot 200 (200 + 0), while in the second layer, `SAMPLEOFF 1` shifts the selection to slot 201 (200 + 1).

2. Multi-Sample and Multi-Banked Instrument (Keyswitch-Controlled Banks)

```
// Keyswitch setup (on key 0)
KEYSWITCH "Style", 1, 0, 0, 0, 200, 202, "Legato", "Staccato", "Pizzicato"

// ... Samples for different notes, velocities, and styles (Legato, Staccato,
Pizzicato)

// Common oscillator using keyswitch value as base sample slot
OUT = OSCG("Sgf000S", MCC(400), Attack, Decay, Sustain, Release)

// Layers with keyswitch-based selection and velocity-based offsets
LAYER
NOTEON 60, 60, 0, 50       // Note C3, velocity 0 to 50
SAMPLEOFF 0                // Sample offset 0, based on Keyswitch selection
LAYER
NOTEON 60, 60, 51, 100      // Note C3, velocity 51 to 100
SAMPLEOFF 3                // Sample offset 3, based on Keyswitch selection
// … and so on…
```

This example introduces keyswitch control:

- A keyswitch is defined using the `KEYSWITCH` instruction, allowing you to select between "Legato," "Staccato," and "Pizzicato" styles by pressing different keys (here, controlled by MIDI note C0 or extended MIDI CC 400).
- The `OUT` instruction now uses `MCC(400)` (the value of the keyswitch) as the base sample slot number.
- Each `LAYER` uses `SAMPLEOFF` to offset the selection within the chosen bank. For instance, if the keyswitch selects the "Legato" bank (starting at sample slot 200), the first layer would use slot 200, the second layer would use slot 203, and so on.

These examples highlight how `SAMPLEOFF`, combined with other Crescendo instructions, provides a powerful and elegant solution for creating instruments with intricate sample mapping and switching mechanisms. By understanding `SAMPLEOFF`, you can unlock a new level of flexibility and control in your sound design process within Crescendo.

MIDI Processing Stage

Here it is a comprehensive explanation of Crescendo's MIDI Processing Stage, a critical component in its signal flow that handles incoming MIDI data and prepares it for subsequent audio generation. This stage encompasses several key processes, including filtering, CC handling, note processing, and post-processing modifications, all of which contribute to the plugin's flexibility and control over MIDI data.

Initial Filtering: Selectively Processing MIDI Channels

Crescendo allows users to filter incoming MIDI data based on its channel, ensuring only relevant messages are processed. The **`MIDICH` instruction** defines the acceptable MIDI channel range, specified as `<min>;<max>`. This filtering occurs early in the MIDI Processing Stage, preventing unnecessary calculations or actions based on irrelevant MIDI data. By default, all channels (0-15) are processed if no `MIDICH` instruction is present.

MIDI CC Value Handling: Storage, Smoothing, and Linking

Crescendo diligently handles MIDI CC (Control Change) messages by:

1. **Storing Latest Values:** The plugin stores the most recently received value for each MIDI CC and for each channel, allowing for later access and utilization in various processing stages.
2. **Smoothing Continuous CCs:** To avoid abrupt parameter changes and potential audio artifacts like clicks or pops, smoothing is applied to continuous MIDI CCs representing parameters like volume, pan, or modulation. This smoothing operates on the stored MIDI CC values.
3. **Linking to VST VARs and Temperaments:**
   - **`MIDICC` Instruction:** This instruction establishes a connection between a MIDI CC of a specific channel and a VST VAR (parameter), enabling real-time control of the VST VAR using a MIDI controller or sequencer. Importantly, this linking preserves the VST VAR's scale, ensuring accurate mapping between the MIDI CC value and the parameter value.
   - **`TEMPERAMENTCC` Instruction:** This instruction links the Temperament drop-down list to a specific MIDI CC of a specific channel, allowing dynamic changes to the tuning

system using a MIDI controller. This opens up creative possibilities for real-time shifts in temperament during a performance.

Keyswitches: Expanding Expressive Control

Keyswitches are a powerful tool for triggering different functionalities or settings within an instrument. In Crescendo, the `KEYSWITCH` instruction, declared in the `COMMON` section, allows you to assign specific keys on your MIDI keyboard to control various aspects of the instrument. This might include activating different layers, changing articulations, selecting samples, or controlling effects. You must choose which channel changes the Keyswitch. Since the channels are 16 and the Keyswitches are at most 8, not all MIDI keyboards may host Keyswitches. Keyswitches values are global though.

The `KEYSWITCH` instruction offers several types of keyswitches:

- **Multiple Keys Keyswitch:** Defines a range of keys where each key within the range corresponds to a distinct setting. Pressing a key activates its associated setting while deactivating others. For instance, you could use a range of keys to select between different playing techniques or articulations, such as legato, staccato, or pizzicato in a string instrument.
- **One Key Keyswitch:** Uses a single key to cycle through a set of values. Each press of the key increments the value, looping back to the minimum value after reaching the maximum. This type is ideal for sequentially selecting different instrument articulations or switching between predefined parameter settings.
- **Two Keys Keyswitch:** Utilizes two keys to increment and decrement a value within a defined range. Pressing one key increases the value, while the other decreases it, cycling through the range. This type is suitable for smoothly adjusting parameters like vibrato depth or filter cutoff frequency.

Polyphony, Glide, and Portamento: Shaping Note Behavior

Crescendo offers various options for shaping the behavior of notes, influencing how they are triggered, sustained, and transitioned between. These options are primarily controlled by the `POLY` instruction.

Polyphony Control:

The `POLY` instruction determines the instrument's polyphony, which refers to the number of notes that can be played simultaneously. This setting impacts how the instrument responds to incoming note messages. The polyphony can be set as global or as per channel.
There are the NORMAL and RETRIG/LEGATO mode, with optional basic gliding effect.

Glide and Portamento:

Crescendo supports both glide and portamento, techniques for creating smooth pitch transitions between notes. The `POLY` instruction allows you to specify the glide time, which determines the duration of the pitch transition.

- **Glide:** Creates a smooth pitch transition between consecutive notes, typically used for expressive playing techniques.

- **Portamento:** Applies a pitch slide from the current note to the next note, often used for creating expressive slides or connecting notes in a melodic phrase.

Pedal and Sostenuto: Emulating Sustain Pedals

Crescendo includes the `PEDAL` and `SOSTENUTO` instructions to simulate the behavior of sustain pedals commonly found on acoustic pianos. These instructions link specific MIDI CC values to sustain functionalities, allowing for nuanced control over note sustain and release.

PEDAL:

The `PEDAL` instruction mimics a standard sustain pedal. When the associated MIDI CC is activated (typically MIDI CC 64), all subsequently played notes are sustained, overriding their natural decay and release characteristics. This effect continues until the MIDI CC is deactivated.

SOSTENUTO:

The `SOSTENUTO` instruction emulates a sostenuto pedal, which selectively sustains notes that were held down at the moment the pedal was activated. Notes played after the pedal is activated are not sustained. This effect remains active as long as the associated MIDI CC is on.

Key Range Specification:

Both the `PEDAL` and `SOSTENUTO` instructions allow you to define a specific key range to which the sustain effect will be applied. This enables you to create instruments with localized sustain zones, mimicking the behavior of some instruments or enabling more creative control over sustain behavior.

Applications Beyond Piano Emulation:

While primarily used for replicating piano pedal behavior, the `PEDAL` and `SOSTENUTO` instructions can be creatively applied to other instrument designs, because they substantially swap release time and decay time. E.g. if the sustain is not 0 you can select amomg two release times effectively implementing a dampen pedal.

Note ON/OFF Message Processing: A Branching Path

The handling of NOTE ON/OFF messages in Crescendo depends on the **state of the internal sequencer**, creating two distinct processing paths:

Sequencer Disabled:

Arpeggiator and Chord Generation:

  - If the **`ARPEGGIO` or `CHORD` instructions** are present, the incoming NOTE ON/OFF messages can trigger these components.
  - The arpeggiator expands single notes into melodic patterns, while the chord generator creates chords from single notes.
  - Both components offer control over parameters like pitch, velocity, duration, and timing of the generated notes.

Sequencer Enabled:

The sequencer provides five distinct operational modes, each offering a different level of control and interaction between the sequencer, the user, and the TAPE:

1. **NORMAL:** The sequencer is off, allowing for TAPE editing or resetting. Incoming MIDI notes pass through unmodified.
2. **PAUSE:** The sequencer is frozen, preserving the current state of the TAPE and passing through incoming MIDI notes.
3. **SEQ. ONLY:** The sequencer plays back notes exclusively from the TAPE, ignoring any live input.
4. **LIVE:** The sequencer plays notes from the TAPE while also allowing live MIDI input to be played simultaneously.
5. **MERGE:** The sequencer plays notes from the TAPE and records incoming MIDI input onto the TAPE, integrating live performance into the sequence.

The Sequencer and the TAPE: Crafting Complex MIDI Patterns

Crescendo's unique feature is the programmable MIDI sequencer, a feature that distinguishes it from traditional step sequencers. This sequencer utilizes a virtual "TAPE" to store and manipulate note data, offering a flexible and powerful environment for crafting intricate musical phrases and patterns.

The TAPE: A Dynamic Container for MIDI Notes

Imagine the TAPE as a virtual tape recorder, capable of storing and manipulating MIDI note information. The TAPE holds a chronologically ordered sequence of notes, each defined by its trigger time, release time, pitch, velocity, and off velocity.

Key TAPE Positions:

- **Cursor:** Indicates the next note scheduled for playback. It moves along the TAPE as the sequencer progresses.
- **End:** Marks the boundary of the currently populated section of the TAPE. New notes generated by the sequencer's program are added after this point.

Dynamic TAPE Population:

The TAPE is not statically defined; it's populated and modified dynamically at runtime. This dynamic behavior is influenced by several factors, including:

- **Initial ARPEGGIO:** If an `ARPEGGIO` instruction is present, the TAPE is initialized with the notes generated by this arpeggio, providing a starting point for the sequencer.
- **Live Input (MERGE Mode):** In the `MERGE` operational mode, the sequencer can record incoming MIDI notes played by the user directly onto the TAPE, enabling real-time interaction and improvisation.
- **Program Output:** The sequencer's internal programming language can generate new notes or modify existing ones on the TAPE, influencing the sequencer's output.

Saving and Loading TAPE Data:

The TAPE's contents can be saved to and loaded from .tap files, which are comma-separated value (CSV) text files. These files store the note data, preserving the timing, pitch, and velocity information for each note in the sequence.

TAPE CSV Structure:

Each row in the .tap file represents a single note on the TAPE. The five comma-separated values in each row correspond to:

1. **Trigger Time:** The time, in seconds from the start of the sequence, when the note should be triggered.
2. **Release Time:** The time, in seconds from the start, when the note should be released.
3. **Pitch:** The note's musical pitch, expressed in semitones (e.g., 69.0 represents A3).
4. **Velocity:** The note's attack velocity (volume), ranging from 0.0 to 127.0.
5. **Off Velocity:** The note's release velocity, also ranging from 0.0 to 127.0.

BPM and Time Signature in TAPE Files:

When saving a TAPE file, the timing data is rescaled as if the sequence is playing at 120 BPM with a 4/4 time signature. This normalization ensures consistency when loading TAPE files into projects with different tempo and time signature settings. Upon loading a TAPE file, the timing data is adjusted to match the current BPM and time signature, preserving the original rhythmic feel of the sequence.

MIDI Post-processing Steps: Shaping Note Data

After processing through the arpeggiator, chord generator, or sequencer, the MIDI note messages undergo further refinement through the MIDI Post-processing steps:

Note Message Modifications:

- **`MAP`:** Constrains incoming note pitches to a specified scale, ensuring generated notes stay within the desired musical context. This is especially useful when working with arpeggiators or chords to control harmonic content.
- **`TIMING`:** Adjusts note timing by adding delays or quantizing note onsets to specific rhythmic grids. This allows for fine-tuning of rhythmic feel and groove.
- **`PITCH`:** Transposes note pitches up or down, enabling global or key-range specific pitch adjustments without affecting tempo.
- **`VELOCITY`:** Modifies note velocities based on various factors like incoming velocity, MIDI CC values, or predefined curves. This provides control over dynamics and expressiveness.

Temperament Step: fine tuning of note pitch

The last step in the MIDI processing pipeline involves the application of a temperament.

Microtuning and Temperaments

In music, a **temperament** defines how the intervals within a musical scale are tuned. The most prevalent temperament in Western music is **equal temperament**, where all 12 semitones within an octave are equally spaced. While this system allows for playing in any key without noticeable dissonance, it comes at the expense of perfectly tuned intervals, which rely on precise mathematical ratios.

Alternative temperaments, like just intonation or meantone temperament, aim to achieve purer intervals but often introduce dissonance in certain keys. For example, **just intonation**, built on simple integer ratios, creates intervals that sound more consonant but might sound dissonant when transposed to certain keys.

Crescendo's Temperament System

Crescendo's temperament system is exceptionally flexible. It allows you to:

- Use default temperaments.
- Define **custom temperaments**.
- Import temperaments from external files.
- Switch temperaments dynamically.

These features open up a world of possibilities for exploring non-standard tuning systems. The only limitation is that there is a shared temperament among all 16 channels. To use different temperaments for different instruments you must use separate Crescendo instances.

Default Temperaments: HOST and EQUAL

Crescendo provides two default temperaments:

- **HOST:** This temperament depends on your DAW's capability to handle detune optional parameter in Note ON messages for tuning adjustments. If your DAW supports temperaments, Crescendo will use the detune information to tune notes accordingly. If not, `HOST` will function like equal temperament.
- **EQUAL:** This represents the standard 12-tone equal temperament.

Defining Custom Temperaments

The `TEMPERAMENT` instruction is the key to defining custom temperaments in Crescendo. You can specify the tuning using several methods:

1. **Cents:** Define how much each semitone deviates from the root note in cents. For example, to sharpen C# by 10 cents, you'd set the C# value to 110.
2. **Cents Deviations:** Define how much each semitone deviates from equal temperament in cents. For example, to sharpen C# by 10 cents, you'd set the deviation for C# to 10.
3. **Number or Name:** You can select a previously defined temperament using its numerical index or name.
4. **Importing from SCALA and TUN Files:** You can import temperaments from **SCALA (.scl)** files, which define microtonal scales, or **AnaMark tuning (.tun)** files.
5. **GUI Selection:** The VST's interface has a dropdown list where you can choose the active temperament.

Importing Temperaments Using SCALA, TUN, and KBM Files

Crescendo supports importing temperaments from specialized file formats, expanding its microtonal capabilities:

- **SCALA (.scl) Files:** Define microtonal scales by specifying the frequency ratios between notes. Crescendo uses these ratios to calculate the tuning of each note.
- **TUN (.tun) Files:** AnaMark tuning files that define base frequencies and deviations from equal temperament for various keys.
- **KBM (.kbm) Files:** Often used with SCALA files, these files define keyboard mappings. They specify how the notes in the SCALA file are mapped to keys on a keyboard, enabling non-standard keyboard layouts.

Dynamic Temperament Switching

Dynamic temperament switching lets you change temperaments in real-time during performance. You can trigger these changes via:

- MIDI CC messages
- VST parameters
- Keyswitches

See TEMPERAMENTCC instruction for details.

MIDI Output:

The final MIDI messages, potentially modified by the arpeggiator, chord generator, sequencer, post-processing steps, and temperament mapping, are sent to both the **next processing stage (Layer Processing)** and the **MIDI output**. This allows for the processed MIDI data to trigger sounds within Crescendo and be sent to other VST instruments or tracks in the DAW.

The channel information is preserved but if the MIDICH instruction is used, not all channels may be routed to the output.

Miscellaneous instructions of the COMMON Section:

`LAST`: Applying Instructions at the Tail End of Each Layer

The `LAST` prefix is a powerful tool for ensuring specific instructions are executed at the very end of each `LAYER` section, regardless of their position in the `COMMON` section.

How `LAST` Works:

1. **Saving for Later:** When Crescendo encounters an instruction prefixed with `LAST` in the `COMMON` section, it stores that instruction in a temporary file without immediate processing.
2. **Appending to Layers:** During the compilation of each `LAYER` section, Crescendo retrieves all instructions saved with the `LAST` prefix and appends them to the end of that layer's code. Note that this doesn't affect the `POST` section.
3. **Delayed Execution:** This ensures that instructions marked with `LAST` are executed only after all other instructions within that specific `LAYER` have been processed.

Benefits and Use Cases:

- **Standardized Elements with Variations:** You can define common instructions in the `COMMON` section with `LAST` to create a baseline for all layers while still allowing for layer-specific modifications. For instance, you could set default filter settings with `LAST` that apply to all layers.
- **Overrides and Refinements:** `LAST` allows you to place instructions in the `COMMON` section that can override or refine settings defined earlier within individual layers. This is helpful for ensuring consistency or applying final adjustments.
- **Organization and Code Clarity:** Using `LAST` can improve code organization and readability by grouping related instructions that should be applied consistently at the end of each layer, even if those instructions logically belong within the `COMMON` section.

Key Points:

- Syntax checking for `LAST`-prefixed instructions occurs within each `LAYER` section, not in the `COMMON` section where they are declared. This means any variables used in a `LAST` instruction must be defined within the scope of the `LAYER` where it's executed.
- `LAST` can be combined with the `HOLD` and `EXECIF` instructions for more complex scenarios.

`FEND`: Global Overrides and Final Say

The `FEND` prefix serves a similar purpose to `LAST` but with a global scope. Instructions marked with `FEND` are executed at the very end of the entire instrument file compilation, potentially overriding settings defined anywhere else in the file, including within the `COMMON` section itself.

How `FEND` Works:

1. **Saving to the End:** Similar to `LAST`, instructions prefixed with `FEND` in the `COMMON` section are saved in a temporary file and not immediately processed.
2. **Appending After Compilation:** Crescendo appends all `FEND`-marked instructions to the end of the compiled code after processing the entire instrument file, ensuring they are executed last.

Purpose and Benefits:

- **Forcing Specific Settings:** `FEND` is useful for enforcing specific global configurations that should not be altered by any other settings in the file. This could include enabling debug mode (`DEBUG` instruction), setting the interface size (`INTERFACE`), or defining the silence threshold (`SILTH`).
- **Overriding Inherited Settings:** When using `INCLUDE` to import settings from external files, `FEND` can override inherited settings that might conflict with the desired configuration of the current instrument.

Example:

```
// In the COMMON section:
DEBUG 0      // Disables debug mode initially
FEND DEBUG 1  // Forces debug mode to be enabled, overriding the previous
setting
```

Limitations:

- Syntax checking for `FEND` instructions occurs at the end of the file compilation. Errors might not be detected until the entire file has been processed.
- `FEND` can only be used with instructions that have a global scope, such as `QUALITY`, `DEBUG`, `INTERFACE`, `UIMOD`, `HIDEUI`, and `SETFONT`. It cannot be used with instructions within `LAYER` or `POST` sections.

`CHOKE`: Replicating Acoustic Instrument Behavior

The **CHOKE** instruction, replicates the behavior of certain acoustic instruments, such as a piano, where striking a key that is already held down produces a distinct sound. When a note is played, Crescendo checks for another note of the same pitch that is currently active. If one is found, the **CHOKE** instruction forces a release of the earlier note, silencing it. This replicates the way a piano hammer hitting an already vibrating string mutes the previous sound.

CHOKE takes one parameter, **`<time>`**, which determines the duration of the forced release in seconds. A value of 0, which is the default, disables the choking effect. The **CHOKE** instruction must be placed within the **COMMON** section of the instrument file to ensure it applies to all layers.

`SILTH`: Defining the Silence Threshold

The **SILTH** instruction in Crescendo's programming language establishes the **amplitude threshold below which a sound is considered silent**, effectively controlling when a layer instance is deactivated and its resources are released. This threshold is crucial for managing polyphony and preventing the accumulation of inaudible sounds that could unnecessarily consume processing power.

The **SILTH** instruction takes one parameter, **`<number>`**, which represents the amplitude threshold as a **floating-point value** ranging from 0 to 1. A lower value corresponds to a quieter threshold, allowing sounds to decay further before being considered silent. Conversely, a higher value represents a louder threshold, leading to earlier deactivation of layer instances.

The **default value for SILTH is 1e-7, equivalent to -140 dB**. This value is typically sufficient for most instruments. However, you can adjust it based on the specific characteristics of your instrument or to achieve particular effects. For instance, you might set a higher **SILTH** value for percussive instruments with rapid decays or a lower value for instruments with long, subtle tails.

Placement:

The **SILTH** instruction can be placed either in the **COMMON** section, affecting the entire instrument, or within a specific **LAYER** section, applying only to that layer. If **SILTH** is not explicitly defined in a layer, it inherits the value from the **COMMON** section.

Example:

```
SILTH 0.0001
```

This sets the silence threshold to 0.0001 (-80 dB), meaning that sounds with amplitudes below this value will be considered silent and the associated layer instance will be deactivated.

Interaction with Off Triggers:

The **SILTH** value plays a role in the behavior of off triggers, particularly **OFFMCCT**, which initiates the release phase of a layer based on changes in MIDI CC values. When the intensity of a layer falls below the **SILTH** threshold during the release phase, the layer instance is completely deactivated.

FEND Instruction for Override:

You can use the **FEND** instruction in the `Settings.ini` file to ensure a specific **SILTH** setting is not overwritten. The **FEND** instruction queues instructions to be processed after the instrument file has been parsed, allowing you to override settings defined within the instrument file itself. This can be useful for enforcing consistent silence threshold behavior across multiple instruments or for testing purposes.

For instance, to guarantee a silence threshold of -100 dB regardless of the settings within an instrument file, you would add the following line to the `Settings.ini` file:

```
FEND SILTH 0.001
```

This will override any **SILTH** instructions found in the instrument file and enforce the -100 dB threshold.

The POST Section

The **POST** section is **optional**. If present, it defines the **final stage of audio processing** after the individual layers have generated their sounds. It operates on the combined output of all active layers and any audio inputs the instrument might have.

Input Variables: A Variety of Signals to Process

The Post-processing Stage has access to a diverse range of input variables, representing audio signals from different sources:

- **OUT and OUTnn Variables:** These variables carry the audio output signals:
  - `OUT` (or its aliases `OUT0` and `OUT00`): This variable holds the combined audio output from all active layers, blended with the audio signal present on `INPUT #1`.
  - `OUTnn` (where `nn > 0`): These variables represent the audio signals specifically routed to the secondary outputs of the plugin, numbered from `OUTPUT #2` onwards. For example, `OUT1` corresponds to `OUTPUT #2`, `OUT2` corresponds to `OUTPUT #3`, and so on.
- **IN and INnn Variables:** These variables provide access to the plugin's audio inputs, allowing for sidechaining and other multi-input processing techniques:
  - `IN` (or its aliases `IN0` and `IN00`): This variable represents the signal received on `INPUT #2`, often used for sidechaining where the amplitude or dynamics of an external audio source control parameters within the instrument.
  - `INnn` (where `nn > 0`): These variables correspond to the signals received on the subsequent inputs, numbered from `INPUT #3` onwards.
- **SENDS Variables:** These variables act as dedicated channels for layers to send specific audio signals directly to the Post-processing Stage, providing greater control over audio routing:

    - `SENDS<num>` (where `<num>` is 1, 2, 3, or 4): Each layer can modify the value of a `SENDS<i>` variable using an assignment like `SENDS<num>=<expr>`, where `<expr>` represents an audio signal or expression.

Output Variables: Defining the Final Audio Output

- **OUT and OUTnn Variables as Output Destinations:** The `OUT` and `OUTnn` variables, initially carrying the audio signals described above, are also the destinations for the processed audio signals.
- **Signal Manipulation Determines Output:** The Post-processing Stage manipulates these variables using expressions and instructions, ultimately determining the final audio output of the plugin.
- **Last Assignment Wins:** The final value assigned to each `OUTnn` variable after the execution of the last instruction in the POST section is the signal that is sent to the corresponding output channel of the plugin.

SENDS Variables: Targeted Routing from Layers

- **Layer-Specific Signals:** The `SENDS<i>` variables are initialized to zero at the beginning of the Post-processing Stage and are modified within individual layer definitions using assignments.
- **Passing Audio to the POST Section:** This mechanism allows layers to send specific portions of their output to the Post-processing Stage for dedicated processing.
- **Discarding Unused Sends:** If a `SENDS<i>` variable is not used within the instructions of the POST section, its value is discarded, meaning the audio signal sent from the layer through that `SENDS<i>` variable will not be included in the final output.

Signal Manipulation and Routing: Shaping the Sound

- **Comprehensive Processing Tools:** The Post-processing Stage has access to a wide range of functions and instructions similar to those available in the LAYER sections, enabling users to manipulate audio signals in various ways.
- **Key Processing Techniques:** Common operations performed in the Post-processing Stage include:
    - **Mixing and Blending:** Combining the audio signals from multiple layers, inputs, and `SENDS<i>` channels using arithmetic operations like addition, subtraction, and multiplication.
    - **Effects Processing:** Applying global effects like reverb, delay, chorus, flanger, distortion, equalization, compression, and more, using specialized functions like `REVERB0`, `SIGM`, `DELAY`, and various filtering functions.
    - **Signal Routing:** Directing specific audio signals to desired output channels by manipulating the `OUTnn` variables.

Final Output: The Culmination of Processing

- **Determined by OUTnn Values:** The final audio output of the Crescendo plugin is determined by the values assigned to the `OUTnn` variables at the end of the Post-processing Stage.

- **Unconnected Outputs Remain Silent:** Audio signals routed to unconnected output channels (`OUTnn` variables corresponding to unused outputs) are discarded.
- **Multi-Channel Output:** By manipulating the `OUTnn` variables, users can create multi-channel audio output, sending different processed signals to separate destinations within the host DAW.

Key Points and Considerations

- **Optional but Powerful:** The Post-processing Stage is an optional step, but it offers significant flexibility and control over the final sound of the instrument or audio effect.
- **No Note-Specific Processing:** The Post-processing Stage does not process NOTE ON or NOTE OFF messages. This means it cannot apply effects or manipulations that are tied to individual notes.
- **Envelopes Not Applicable:** Envelopes, which are used to shape sound over time in response to note events, are not available in the Post-processing Stage.
- **Global Effects and Refinement:** This stage is primarily used for applying effects that impact the entire output of the instrument, as well as for final mixing, routing, and refinement before the audio leaves the plugin.

Delimiter: The **POST** section ends when the **`LAYER`** keyword (signifying the start of a new layer) is encountered or when the instrument file ends.

The LAYER Sections

LAYER sections are the **heart of your instrument** where the actual sound generation and processing take place. Each **LAYER** section defines a **distinct, self-contained sound-generating unit**. An instrument can have **multiple LAYER** sections, allowing for the creation of complex, multi-layered sounds.

Key Roles of a `LAYER` Section:

- **Triggers: Defining When the Layer is Activated:** Each **LAYER** section starts by defining its **triggers**. Triggers are conditions that determine when the layer becomes active. They can be based on:
    - **`NOTEON`:** Specific MIDI notes or note ranges.
    - **Velocity Ranges:** The incoming note's velocity (how hard a key is pressed).
    - **`KEYSWITCH`:** Dedicated keys used to activate or deactivate layers or change their behavior.
    - **MIDI CC Values:** The values of specific MIDI controllers.
    - **VST Variable Values:** The values of the instrument's VST parameters.
- **Sound Generation and Processing:** Once triggered, a layer defines how it produces and processes sound. This is done through a series of instructions and expressions that might include:
    - **`OSCG` and Related Functions:** Generating audio waveforms using various oscillator types (sine, sawtooth, square, triangle, noise, samples, etc.).
    - **`ENV` and Related Functions:** Shaping the sound's amplitude or other parameters over time using envelopes (attack, decay, sustain, release).
    - **`FILT` and Related Functions:** Modifying the frequency content of the sound using various filter types (low-pass, high-pass, band-pass, etc.).

    - **Other Functions:** Applying built-in or user-defined audio effects through the use of the vast set of linear and non linear functions available.
- **Output Routing:** Each layer defines where its output goes. This is done using assignments to variables like **`OUT`**, **`OUTnn`**, and **`SENDS<i>`**.
    - **`OUT` or `OUT0`:** Sends the layer's output to the main output bus, where it's combined with the output from other layers.
    - **`OUTnn` (where `nn` is a number):** Routes the layer's output to specific output channels, enabling multi-channel audio configurations.
    - **`SENDS<num>=<expr>`:** Routes a specific signal or expression from the layer to the **POST** section using a dedicated send channel, allowing for specialized processing within the **POST** stage.
- **Off Triggers: Defining When the Layer Releases:** In addition to on triggers, layers can define **off triggers**, which determine when the layer enters its release phase and eventually becomes inactive. Off triggers can be based on similar conditions as on triggers, such as:
    - **`NOTEOFF`:** When the corresponding MIDI note is released. This trigger is always active and does not need to be specified.
    - **Keyswitch Changes:** When a keyswitch is deactivated.
    - **MIDI CC or VST Variable Changes:** When controller or parameter values cross certain thresholds.
    - **Time-Based Conditions:** After a specified duration (using the **`DURATION`** keyword).
- **Forced Release:** Off triggers can also include a **forced release parameter**, which overrides the normal release time of envelopes and other parameters, enabling quick and abrupt sound termination under specific conditions.

Delimiter: Each **LAYER** section ends when another **`LAYER`** keyword is encountered or when the instrument file ends.

Trigger Evaluation: Determining Layer Activation

- **The Foundation of Sound Generation:** Triggers within a layer definition determine when and how that layer becomes active and starts producing sound.
- **Triggers are Conditional Statements:** These conditions can be based on various factors, including note pitch, velocity, keyswitches, MIDI CC values, and VST variable values.
- **A Variety of Trigger Types:** Crescendo offers several trigger instructions to define activation conditions, each sensitive to different control signals:
    - `ONNOTEON`: Triggers based on note pitch and velocity range. This is the default trigger and is assumed if no other trigger instructions are present.
    - `ONMCCT`: Triggers based on MIDI CC values, including extended MIDI CC, keyswitches or VST VAR (parameter).
    - `ONFIRST` and `ONLEGATO`: Creates an exclusive triggering system where one layer is active when no notes are held down and another layer takes over when notes are already active.
    - `ONRROBIN`: Distributes notes across multiple layers in a round robin fashion.
    - `ONCHANNEL`: Triggers based on the MIDI channel of the incoming note.
- **Trigger Logic: "AND" for ON Triggers:** If multiple ON trigger conditions are defined for a layer, **all of them must be met simultaneously** for the layer to be triggered.

Example:

Let's consider a string instrument emulation with separate layers for pizzicato and arco (bowed) playing styles:

- **Pizzicato Layer:**
    - `ONNOTEON 48, 72, 0, 63` (Trigger for notes C3 to C5 with velocities from 0 to 63)
    - `ONMCCT 401, 1, 1` (Trigger if keyswitch 1 is activated)
- **Arco Layer:**
    - `ONNOTEON 48, 72, 0, 127` (Trigger for notes C3 to C5 with any velocity)
    - `ONMCCT 402, 1, 1` (Trigger if keyswitch 2 is activated)

In this scenario, to trigger the **pizzicato layer**, both the note pitch/velocity condition and the keyswitch condition must be met. For the **arco layer**, only the keyswitch condition is required.

The `XFADE` Instruction: Seamless Transitions Between Layers

The `XFADE` instruction facilitates smooth transitions between layers, allowing you to create crossfades controlled by a designated MIDI CC value. The `XFADE` instruction offers two types of crossfades:

- **IN Crossfade:** The layer's volume gradually increases as the associated MIDI CC value rises.
- **OUT Crossfade:** The layer's volume gradually decreases as the associated MIDI CC value rises.

If the conditions specified in the instruction result in a volume of zero, the layer is not triggered.

You can fine-tune the crossfade behavior by adjusting the following parameters:

- **MIDI CC Number:** You can select any assignable MIDI CC to control the crossfade, providing flexibility in mapping crossfades to different controllers.
- **Minimum and Maximum Values:** The `min` and `max` parameters determine the range of the MIDI CC values that will affect the crossfade.
- **Power:** The `power` parameter allows you to shape the crossfade curve, controlling the rate at which the volume changes in relation to the MIDI CC value.

This feature is implemented to accommodate the habits of classic layered instrument developers. Crescendo supports more than one oscillator per LAYER, so this feature can be emulated with an expression including all the sound sources.

Sound Generation and Evolution: The Heart of the Layer

- **Triggered Layers Come to Life:** Once a layer's trigger conditions are satisfied, its sound generation and processing instructions come into play.
- **A Diverse Set of Sound Design Tools:** Crescendo's programming language provides a rich set of functions and instructions to shape the sound, including:
    - **Oscillators:** These components generate the fundamental audio signal, offering a variety of options, including:
        - Synthesized waveforms (sine, sawtooth, square, triangle, noise)
        - Sampled waveforms (using the `SAMPLE` instruction to load audio files)
        - Sophisticated oscillator functions like `OSCG`, `SUPERSAW`, `SINC`, `WAVETABLE`, `GRAINSYNTH`, and `WAVESCAN`

- **Envelopes:** These shape the temporal evolution of various parameters like amplitude, frequency, or filter cutoff using functions like `ENV`, `ENV1`, `ENVCURVE`, and `ENV2`.
  - **Filters:** These modify the frequency content of the audio signal, offering various filter types like low-pass, high-pass, band-pass, and notch, with adjustable parameters like cutoff frequency and resonance.
  - **Effects:** A wide array of audio effects, such as reverb, delay, chorus, flanger, phaser, distortion, and pitch shifting, can be implemented using specific instructions and functions.
  - **Mathematical Operations and Functions:** A full set of mathematical operators and functions are available to manipulate audio signals and create complex processing chains.
- **Signal Flow and Processing Order:** The order in which the sound generation and processing instructions are written within the LAYER section determines the signal flow, similar to how audio effects are chained in a DAW.

Sampled Waveforms: Capturing Real-World Sounds

Crescendo embraces the use of **audio samples as the foundation for sound creation**, mirroring the approach of many samplers and virtual instruments. Users can load audio files, such as **WAV, AIFF, and AIFF-C**, directly into the plugin using the **`SAMPLE` instruction**. If FFMPEG is installed, then most media files, including videos, can be selected.

- **Flexibility in Sample Types:** Crescendo handles a variety of sample types:
  - **Single-Cycle Waveforms:** These are short, repeating waveforms that represent a single cycle of a sound. They are often used as the basis for subtractive synthesis, where filters and envelopes shape the sound over time.
  - **Longer Recordings:** These can be recordings of acoustic instruments, vocal phrases, environmental sounds, or any other audio material. They can be used to recreate realistic instrument sounds or to create unique sonic textures.
- **Looping Options:**
  - The `SAMPLE` instruction supports various looping modes, enabling users to control how the sample is played back.
  - This allows for the creation of sustained sounds from shorter samples or for creating interesting rhythmic variations.
- **Separate Release Sample Handling:** Crescendo also supports separate release samples, allowing for more realistic articulation and decay behavior.

Synthesized Waveforms: Mathematically Generated Sounds

Crescendo provides a range of **built-in synthesizer waveforms**, generated mathematically or algorithmically:

- **Classic Waveform Options:** These include the fundamental waveforms commonly found in synthesizers:
  - **Sine:** Smooth, rounded waveform, often used as the basis for other sounds.
  - **Sawtooth:** Bright, buzzy waveform, often used for leads and basses.
  - **Square:** Hollow, punchy waveform, often used for basses and percussion.
  - **Triangle:** A softer, less harmonically rich waveform than the sawtooth, often used for pads and mellow sounds.

    - **Noise:** Random, aperiodic signal, often used for percussive effects or as a modulation source.
- **Waveform Smoothness Options:** Crescendo allows for adjusting the smoothness of square and sawtooth waveforms, reducing high-frequency content to mitigate potential artifacts.
- **Unison Detuning:** For richer and thicker synthesized sounds, Crescendo supports unison detuning. Users can create multiple instances of the oscillator, even with different waveforms for each harmonic, slightly detuning their frequencies to simulate the imperfections of analog circuits or to create wide, chorused sounds.

Specialized Oscillator Functions: Beyond the Basics

Beyond the `OSCG` function, which handles general oscillator duties, Crescendo offers a range of **specialized oscillator functions**:

- **`SUPERSAW`, `SUPERSAW0`, `SUPERSAW1`:** These functions generate stereo sawtooth waveforms with multiple harmonics, ideal for creating the lush, detuned sounds popularized by Roland's JP-8000 synthesizer.
- **`SINC`:** This function produces a stereo sinc waveform, characterized by its unique spectral properties.
- **`WAVETABLE`:** This function uses a sample as a lookup table for wavetable synthesis. By cycling through different portions of the sample, users can create evolving timbres and unique sonic textures.
- **`GRAINSYNTH`:** This function performs granular synthesis, a technique where sound is created by manipulating small fragments of audio samples called "grains." By controlling parameters like grain size, density, and pitch, users can create textures ranging from subtle shimmer to dense clouds of sound.
- **`WAVESCAN`:** This function performs wavescanning, another granular synthesis technique that involves reading through a sample at a variable speed. This creates evolving textures and sonic movements based on the sample's content.

Combining Sampled and Synthesized Sources: A Hybrid Approach

Crescendo's flexibility allows users to **combine sampled and synthesized waveforms** within a single instrument or effect. This hybrid approach opens up a world of creative possibilities:

- **Layering Techniques:** Users can layer multiple oscillators, each using either a sampled waveform or a synthesized waveform, to create rich and complex sounds. For example, a piano instrument could blend sampled recordings of individual notes with synthesized elements like hammer noise or string resonance to enhance realism.
- **Modulation and Crossfades:** By using envelopes, LFOs, and other modulation sources, users can dynamically blend between different sound sources. This allows for evolving timbres and textures, such as gradually morphing between a sampled vocal phrase and a synthesized pad sound.

OSCG: A Single Function for Two Roles

There is a key design principle within Crescendo: the unification of standard oscillators (for audible sound) and LFOs (for modulation) under a single function, `OSCG`. This unification is facilitated by the **AUTOPHASE** feature, enabling a single function to serve both roles.

- The `OSCG` function stands as the core oscillator in Crescendo, capable of generating a diverse array of waveforms – from sine, sawtooth, square, and triangle to noise and sample-based waveforms. This versatility allows it to handle both audio-rate sound production and low-frequency modulation.
- Crescendo deliberately blurs the line between standard oscillators and LFOs, utilizing frequency and the AUTOPHASE option as the primary differentiators. This design choice allows any function generating a periodic signal to operate in either capacity, including the use of sampled waveforms for LFOs.

Frequency and AUTOPHASE: Key Differentiators

- **Frequency:** The frequency parameter within the `OSCG` function determines whether the output is perceived as audible sound or an LFO:
    - **Standard Oscillators:** For audible sounds, the frequency is typically set within the human hearing range (approximately 20 Hz to 20 kHz).
    - **LFOs:** To create modulation signals, the frequency is set below the audible range, typically less than 20 Hz.
- **AUTOPHASE:** This feature, managed through the `OSCG` function's format string, plays a crucial role in emulating the behavior of continuously running oscillators found in many analog synthesizers.
    - **Standard Oscillators:** When generating audible sound, AUTOPHASE is generally enabled. This simulates the random phase offset of an oscillator that is constantly running, even when no notes are being played. This offset is calculated for each harmonic at trigger time, introducing subtle variations that result in a richer, more organic sound.
    - **LFOs:** For LFO applications, AUTOPHASE is typically disabled to ensure a consistent starting phase for predictable modulation patterns. This means that the LFO will start at a defined phase, usually zero, with each cycle.
    - **Sampled Oscillators:** The AUTOPHASE feature is not applicable and therefore disabled when using sampled waveforms.

Two examples showcasing the adaptation of the `OSCG` function for distinct roles:

- **Standard Oscillator with AUTOPHASE:** `OUT = OSCG("s1vL000", 440)` – defines a sine wave oscillator with a frequency of 440 Hz and AUTOPHASE enabled, simulating a continuously running physical oscillator.
- **LFO with AUTOPHASE Disabled:** `FreqMod = OSCG("s1v0000", 5)` – creates a sine wave LFO with a frequency of 5 Hz and AUTOPHASE disabled, ensuring a consistent starting phase of zero for each cycle. This LFO can be used to modulate the frequency of another oscillator, creating a vibrato effect.

A Look at Default Parameters and Their Modulation

Crescendo offers a range of default parameters and associated modulation options designed to streamline sound design and provide users with a set of pre-configured tools for shaping sound. These parameters, accessible within the instrument file's COMMON or LAYER sections, act as starting points that can be modified, overridden, or linked to various MIDI controllers for dynamic manipulation.

- **`BASEF`:** Sets the base frequency used in calculating the default oscillator frequency (`FREQ`) and filter cutoff frequency (`DEFAULTFC`). By default, `BASEF` is set to 440 Hz. It can be modified using the `BASEF` instruction within the COMMON section or overridden within a specific LAYER.
- **`BASEG`:** Defines the base gain for oscillators, influencing the calculation of the default gain (`GAIN`). Its default value is 1.0. Like `BASEF`, `BASEG` can be adjusted in the COMMON section using the `BASEG` instruction or altered within individual LAYERs.
- **`VELTRACK`:** Determines the influence of note-on velocity on the default gain (`GAIN`) calculation. The higher the `VELTRACK` value, the more pronounced the effect of velocity on the oscillator's gain. It defaults to 2.0 and can be adjusted using the `VELTRACK` instruction.
- **`KEYTRACK`:** Controls how the pitch of an oscillator responds to changes in the MIDI note number. A `KEYTRACK` value of 1.0 results in a standard 12-tone equal temperament, where each note is a semitone higher than the previous one. Values other than 1.0 can create alternative tunings and microtonal scales. The default value is 1.0.
- **`FREQ`:** Represents the default oscillator frequency calculated based on several parameters, including `BASEF`, `KEYTRACK`, and the played MIDI note. It's accessible as an extended MIDI CC and can be used directly within expressions or as a modulation target.
- **`GAIN`:** Represents the default oscillator gain, influenced by parameters such as `BASEG`, `ONVEL` (note-on velocity), and `VELTRACK`. Similar to `FREQ`, it's accessible as an extended MIDI CC and can be used in expressions or modulated.
- **`DEFAULTFC`:** Represents the default cutoff frequency for the `FILT0` function. Users can base it on a fixed value or link it to the calculated frequency (`FREQ`), enabling modulation and envelope control similar to `FREQ` and `GAIN`.

Default Modulation

Crescendo provides a rich set of options for modulating these default parameters, allowing users to create expressive and evolving sounds.

- **`*MOD` Instructions:** These instructions allow for linear modulation of parameters like `FREQ`, `GAIN`, and `DEFAULTFC` using MIDI CCs. Users can specify the desired MIDI CC number and the modulation amount in cents for frequency and filter cutoff or dB for gain. For instance, `FREQMOD` can link a specific MIDI CC to the oscillator's frequency, enabling real-time pitch bending.
- **`*ENV` Instructions:** These instructions apply multiplicative envelopes to the default parameters, shaping their evolution over time. Users can control attack, decay, sustain, release, and other envelope stages for nuanced dynamic control. For example, `GAINENV` applies a gain envelope to the default gain, controlling the volume's rise and fall over a note's duration.
- **`*LFO` Instructions:** These instructions apply LFOs to parameters, introducing periodic modulations for effects like vibrato and tremolo. Users can customize the LFO's waveform, frequency, amount, and other parameters. For instance, `FREQLFO` applies an LFO to the frequency, creating a vibrato effect.
- **`COMPLEXMOD` Instruction:** This instruction allows for more intricate modulation scenarios by multiplying up to three MIDI CC values and applying the result to various parameters.

Global and Local Modulations

- **Global Scope:** By placing modulation instructions in the COMMON section, users apply them to all layers within the instrument, establishing a consistent modulation behavior.
- **Local Scope:** For layer-specific control, these instructions can be placed within individual LAYER sections, overriding any global settings for that layer. This enables tailored modulation settings for each sound-generating unit within the instrument.

Interaction and Merging

When both global and local modulation instructions for the same parameter and MIDI CC are present, Crescendo intelligently merges them. The resulting modulation combines both the global and local settings, providing a nuanced approach to parameter control.

Note that the final formula encounters further differentiation due to multichannel: even if the formula is the same, for instance the GAIN parameter can be different between two layers if they are relative to notes coming from different channels, because the values of the MIDI CC used to calculate the value may be different between the two channels.

Considerations and Limitations

- **Predefined Envelopes within Oscillators:** The `OSCG` oscillator function can use these predefined envelopes, but they come with limitations. They are restricted to one per layer per parameter and offer limited control over envelope shapes. For more complex or multi-oscillator scenarios, separate `ENVx` functions or the built-in envelope options through its `OSCG` format string (gain only) are required.
- **`POST` Section Limitations:** The default parameters `FREQ`, `GAIN`, `DEFAULTFC`, and their associated modulation options are not directly usable in the POST section. This is because the `POST` section operates on the mixed audio output from all layers and does not have access to note-level information, which these parameters rely upon.

Expressions: The Heart of Dynamic Control

Expressions, the core of Crescendo's programming language, form the foundation for automation and modulation within the plugin. They are mathematical and logical constructs that can be used to manipulate nearly every parameter within the instrument, creating complex and evolving soundscapes.

- **Composition:** Expressions are built using a combination of constants, variables, MIDI CCs, keywords, operators, and functions, allowing you to perform calculations, manipulate signals, and control various aspects of sound generation and processing.
- **Unlimited Automation:** Expressions enable you to automate almost any parameter, creating sounds that respond to MIDI input, internal modulators, or VST variables. You could, for example, automate an oscillator's frequency based on the velocity of incoming MIDI notes, or control the cutoff frequency of a filter over time using an envelope.
- **Versatility in Sound Design:** Expressions empower you to implement a vast range of audio effects. You can even emulate the behavior of physical synthesizers by recreating techniques like FM or AM modulation.
- **Conditional Logic:** Crescendo provides conditional execution constructs, like IF...GOTO/EXIT and EXECIF, which add further flexibility. You can make decisions based on specific conditions, leading to more dynamic and responsive instruments.
- Note that the final formula encounters further differentiation due to multichannel: even if the formula is the same, the value can be different between two layers if they are relative to

notes coming from different channels, because the values of the MIDI CC used to calculate the value may be different between the two channels.

Modulation Sources: Shaping the Sound

Crescendo offers a diverse set of modulation sources for shaping and animating your sounds:

- **LFOs (Low-Frequency Oscillators):** LFOs provide periodic signals that can modulate parameters like frequency, amplitude, or filter cutoff, creating effects such as vibrato, tremolo, and wah. Crescendo uniquely unifies LFOs and oscillators, meaning any function that generates a periodic signal, including sampled waveforms, can serve as an LFO.
- **Envelopes:** Envelopes automate parameter changes over time, controlling the attack, decay, sustain, and release phases of a sound, influencing its dynamics and evolution.
- **MIDI CCs (Control Changes):** MIDI CCs provide a standard way to control parameters in real-time using external MIDI controllers.
- **VST Variables (VST VARs):** These are user-defined parameters that can be controlled from the host DAW's interface or through Crescendo's GUI. VST Variables offer another layer of control over settings like filter cutoff frequencies and modulation amounts.
- **Sidechaining:** By supporting multiple inputs, Crescendo brings external audio signals into its processing pipeline. The `INxx` variables, representing input signals, can be utilized within expressions, allowing for sidechain modulation techniques.

Advanced Modulation Techniques

Crescendo offers advanced functions for sophisticated modulation scenarios:

- **`MOD2` Function:** This function introduces multiple Look-Up Tables (LUTs) for non-linear transformations, dynamic amplitude scaling, and a final transformation stage. It offers a level of flexibility comparable to SoundFont 2.04 modulation and can be used to create complex and evolving modulations.

Filters and Equalizers: Shaping the Frequency Spectrum

Crescendo offers various filter types and equalizers, giving you precise control over the frequency content of audio signals:

- **`FILT` Function:** This versatile function provides a general-purpose filter with adjustable cutoff frequency (`Fc`), resonance (`res`), and filter type. You can select from low-pass, high-pass, band-pass, notch, and all-pass filter types, enabling a wide range of tonal shaping possibilities.
- **`FILT0` Function:** Similar to `FILT`, `FILT0` simplifies filter implementation by automatically deriving the cutoff frequency from the `DEFAULTFC` MIDI CC value. This allows for convenient control of the cutoff frequency using a MIDI controller.
- **`EQ3DB` and `EQ1DB` Functions:** Crescendo provides both 3-band and 1-band equalizers with adjustable frequency ranges, gains, and filter slopes. These functions enable you to boost or attenuate specific frequency bands, shaping the overall tonal balance of the sound.
- **Pre-Filtering for Samples:** You can pre-filter sample data before normalization or equalization using the `FILTER` and `FILTERSEMI` prefixes in the `SAMPLE` instruction. This applies fixed filters to the samples, reducing real-time processing requirements.

Effects and Signal Processing: Enhancing and Transforming Sound

Crescendo features a collection of built-in effects and signal processing functions, expanding the possibilities for sound manipulation:

- **Delay Functions:** Crescendo offers both interpolating and non-interpolating delay functions, catering to different needs and processing requirements. These functions are suitable for creating delays, chorus, flanger, and other time-based effects.
- **Reverb Functions:** You can simulate reverberation using Crescendo's predefined reverb functions. These functions utilize non-interpolating delays to create realistic or stylized reverberant spaces with adjustable parameters like room size, feedback coefficients, and stereo width.
- **Pitch Shifting:** The `PSHIFT` and `PSHIFT2` functions provide real-time pitch shifting capabilities, allowing you to alter the pitch of an audio signal without affecting its tempo. This enables creative pitch manipulation and live performance effects.
- **Noise Gates and Compression:** Crescendo includes functions for noise gating and compression, giving you control over the dynamic range of audio signals. Noise gates effectively silence quiet passages and reduce background noise, while compressors can make quieter parts louder and louder parts quieter, resulting in a more balanced and controlled sound.
- **Comb Filters:** Comb filters create a series of peaks and notches in the frequency spectrum, often used to create flanging or phasing effects.

These functions are often applied in the POST step, but are available also in the LAYER section for special effects.

Envelopes: Shaping Sound Over Time

Envelopes play a crucial role in shaping how sound evolves over time by controlling parameters like amplitude, frequency, or filter cutoff. Crescendo offers various envelope functions:

- **`ENV` Function:** This function creates versatile linear or exponential envelopes with customizable levels and hold times. You can define the envelope's curve with great flexibility, including looping, triggering (ignoring note off), and beat-synced retriggering options.
- **`ENVCURVE` Function:** For more complex envelope shapes beyond simple ADSR models, you can use the `ENVCURVE` function. It defines custom envelopes using lookup tables (`LUT`s), offering precise control over hold times, attack, decay, and release stages.
- **`ENV0`, `ENV1`, and `ENV2` Functions:** These functions provide simpler ADSR envelope types with varying levels of complexity. `ENV0` generates a standard ADSR envelope, `ENV1` adds a delay and a hold time, and `ENV2` introduces a second decay stage.

Distortion Functions: Adding Grit and Character

Crescendo's distortion functions introduce harmonic and non-linear characteristics to audio signals, adding warmth, grit, or aggressive timbres:

- **`SAT` and `SAT2` Functions:** These functions provide saturation effects, clipping the signal at specific thresholds. `SAT` saturates between 0 and 1, while `SAT2` saturates between -1 and 1.
- **`SIGM` and `SIGMDW` Functions:** These functions implement sigmoid saturation, introducing a smooth and controllable type of distortion. `SIGMDW` also incorporates a dry/wet control for blending the distorted signal with the original.
- **`POWABS` Function:** The `POWABS` function allows for symmetric distortion based on a power function, enabling you to shape the distortion curve according to the desired harmonic content.
- **`CURVE` Function:** The `CURVE` function provides a general-purpose lookup table (`LUT`) mechanism for implementing non-linear transformations, enabling you to create custom distortion curves for unique sonic effects.

Transcendental Functions: Mathematical Flexibility

Crescendo supports standard transcendental functions, expanding the possibilities for complex calculations and signal processing within expressions:

- **`LOG` and `EXP` Functions:** These functions implement the natural logarithm and exponential functions, allowing for logarithmic scaling and exponential growth within expressions.
- **Trigonometric Functions (`SIN`, `COS`, `TAN`, `ASIN`, `ACOS`, `ATAN`):** These functions provide standard trigonometric operations, enabling the creation of periodic signals and implementing various signal processing techniques.

Nonlinear Functions: Shaping Signals Creatively

Crescendo's nonlinear functions offer additional tools for shaping signals and introducing unique characteristics:

- **`ABS` and `SIGN` Functions:** The `ABS` function calculates the absolute value of an expression, useful for extracting amplitude information or creating symmetric distortions. The `SIGN` function returns the sign of an expression, enabling the creation of various nonlinear effects.
- **`COPYSIGN` Function:** This function copies the sign of one signal to another, allowing for more controlled manipulation of signal polarity.
- **`ROUND`, `FLOOR`, `CEIL`, `FRAC` Functions:** These functions perform various rounding operations on input signals, which can be useful for creating stepped or quantized effects.

The `PREV` Function: Unlocking Feedback Systems and Digital Filters

The `PREV` function retrieves the value of a variable from the previous sample, enabling the creation of feedback loops and the implementation of digital filters. This function expands Crescendo's signal processing capabilities, allowing you to:

- **Simulate Feedback Systems:** `PREV` allows you to create feedback systems where the output of a process is fed back into its input, potentially with modifications or delays. This enables the realization of effects like echo, chorus, and flanger.
- **Implement Digital Filters:** You can utilize the `PREV` function to construct custom Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters. FIR filters rely on past

input samples to calculate the output, while IIR filters also incorporate past output samples, creating more complex filter responses.

- **Build State Machines:** By storing state information in variables and updating them based on previous values, you can use `PREV` to create state machines within Crescendo. These state machines can control various aspects of the instrument's behavior, such as switching between different sound generation modes or implementing sophisticated modulation schemes.

The `SHOLD` and `HOLD` Keywords: Managing Variable Evaluation

Both the `SHOLD` function and the `HOLD` keyword influence the evaluation and storage of variable values within Crescendo, but they differ in their specific behavior:

- **`SHOLD` Function: Sample and Hold**
    - **Functionality:** The `SHOLD` function samples the value of an expression at specific intervals and holds that value until the next sampling point. You can control the sampling interval using a time parameter specified in beats. It can also hold the sample after release.
    - **Applications:** `SHOLD` is useful for quantizing or smoothing continuously changing values, creating stepped modulation effects, or simulating sample-and-hold circuits found in some analog synthesizers. Hold after release can be used for MPE instruments.
- **`HOLD` Keyword: Trigger-Time Evaluation**
    - **Functionality:** The `HOLD` keyword ensures that an expression is only evaluated at the moment a note is triggered. The calculated value is then held constant for the duration of the note.
    - **Applications:** `HOLD` is ideal for optimizing performance by reducing unnecessary calculations. It can also create specific effects where a parameter needs to be fixed at the start of a note, such as setting the initial frequency of an oscillator or the cutoff frequency of a filter.

CURVE: Shaping Sounds with Lookup Tables in Crescendo

Here it is a detailed look at the `CURVE` instruction and its associated functions (`CURVE`, `ENVCURVE`, and `MOD2`), highlighting their importance in shaping and modulating sounds within the Crescendo VST plugin. These elements provide a powerful mechanism for introducing non-linear behaviors and complex transformations, expanding the sound design capabilities beyond simple linear relationships.

Defining Custom Curves with the `CURVE` Instruction

The `CURVE` instruction is the foundation for creating custom lookup tables (LUTs) that define mappings between input values and output values. These LUTs can then be applied to various parameters using the `CURVE` function.

Syntax and Parameters

The syntax for the `CURVE` instruction is:

```
CURVE <index>, <x1>, <y1>, ..., <xN>, <yN>
```

where:

- `<index>`: An integer value representing the slot number where the LUT will be stored. You can define multiple LUTs, each with a unique index.
- `<x1>, <y1>, ..., <xN>, <yN>`: Pairs of values that define the points in the LUT. The `<x>` values represent the input, and the `<y>` values represent the corresponding output. The `<x>` values must be strictly increasing, and there must be at least two pairs.

Characteristics

- **Linear Interpolation:** The `CURVE` instruction uses linear interpolation to calculate output values for input values that fall between the defined points in the LUT. This ensures smooth transitions between the specified points.
- **Overwriting Slots:** If you define a `CURVE` with an index that already has a LUT stored in it, the existing LUT will be overwritten.

Example

```
CURVE 1, 0, 0, 0.5, 1, 1, 1
```

This instruction defines a concave curve in slot 1. Input values from 0 to 0.5 will be mapped linearly to output values from 0 to 1, while input values from 0.5 to 1 will be mapped to an output value of 1.

Applying LUTs with the `CURVE` Function

The `CURVE` function allows you to apply a previously defined LUT to a specific parameter or expression. This enables you to introduce non-linear transformations and shape how a parameter responds to modulation.

Syntax and Parameters

The syntax for the `CURVE` function is:

```
CURVE(<index>, <x>)
```

where:

- `<index>`: The slot number of the LUT you want to apply.
- `<x>`: The input value to be mapped using the LUT. This can be a constant, a variable, or the result of an expression.

Characteristics

- **Stereo Handling:** The `CURVE` function can handle both mono and stereo input values. If the input is stereo, the LUT will be applied to each channel independently.
- **Predefined Curves:** For convenience, the `CURVE` function provides several predefined curves, accessed using negative index values. These offer common non-linear transformations like absolute value, square root, and saturation.
- **Invalid Indices:** If you use an index that is out of bounds or refers to an empty slot, the `CURVE` function will perform an identity mapping (output = input).

Example

```
OUT = OSCG("skk", 1, 440 * CURVE(1, MCC(1)))
```

This example applies the LUT defined in slot 1 (from the previous example) to the value of MIDI CC 1 (often the modulation wheel). The result is used to modulate the frequency of a sine wave oscillator. As the modulation wheel changes, the frequency will change non-linearly, following the shape of the defined curve.

`ENVCURVE`: Shaping Envelopes with Custom Curves

The `ENVCURVE` function expands on the `CURVE` concept by allowing you to define custom envelope shapes using LUTs. This provides more control over the attack, decay, sustain, and release stages of a sound, going beyond simple ADSR (attack, decay, sustain, release) envelopes.

Syntax and Parameters

The syntax for the `ENVCURVE` function is:

```
ENVCURVE(<option>, <index>, <H1>, <atk>, <H2>, <dcy>, <H3>, <rel>)
```

where:

- `<option>`: Controls aspects of the envelope behavior, such as looping and triggering modes.
- `<index>`: The slot number of the LUT used to define the envelope shape.
- `<H1>, <atk>, <H2>, <dcy>, <H3>, <rel>`: Define the hold times (H1, H2, H3), attack time (`atk`), decay time (`dcy`), and release time (`rel`) of the envelope. These parameters determine how the input value to the LUT is swept over time.

Characteristics

- **LUT Mapping:** The `ENVCURVE` function sweeps through the LUT based on the specified times, using linear interpolation to determine the output value at each point in time. This allows for complex and expressive envelope shapes. See the detailed explanation for details.

Example

```
VOLENV = ENVCURVE(0, 2, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6)
```

This example defines an envelope using the LUT in slot 2. The envelope will have an initial hold time of 0.1 seconds, an attack time of 0.2 seconds, a hold time of 0.3 seconds after the attack, a decay time of 0.4 seconds, a hold time of 0.5 seconds after the decay (during sustain), and a release time of 0.6 seconds. The output of `VOLENV` will change over time based on the LUT values and the specified times.

Off Trigger Evaluation: Initiating Sound Release

- **Controlling Sound Termination:** Off triggers define the conditions that initiate the **release phase** of an active layer instance. This phase marks the beginning of the sound's decay, governed by envelope settings, until it falls below the defined silence threshold and the layer instance is finally deactivated.

- **Similar Variety to ON Triggers:** Crescendo offers several off trigger types, mirroring the ON trigger options:

    - `OFFMCCT`: Triggers release based on changes in MIDI CC values, including extended MIDI CC, keyswitches or VST VAR (parameter).
    - `OFFGROUP`: Triggers release when a layer belonging to a specific group is triggered.

- **NOTE OFF Messages:** The standard NOTE OFF message, sent when a key is released, inherently acts as an off trigger.
- **"OR" Logic for Off Triggers:** If multiple off triggers are defined for a layer, the **activation of any one of them** will initiate the release phase.
- **Forced Release:** Off trigger instructions allow for a `forcedrelease` parameter, which optionally overrides the layer's default release times defined by its envelopes, enabling abrupt sound termination.

Layer Output: Contributing to the Instrument's Sound

- **Summing Individual Outputs:** The audio output of each active layer instance is summed together.
- **Passing to the Next Stage:** This combined output is then passed to the **Post-Processing Stage** for further manipulation and finalization.

User Interface

Core UI Elements

- **Knobs:** Represent adjustable VST variables (VST Vars). These variables correspond to various instrument parameters, enabling real-time control and automation. Crescendo allows customization of knob appearance, including diameter, border thickness, and color.
- **Dropdowns:** Used for selecting from predefined options.
    - **Temperament Dropdown:** Allows selection of the active musical temperament. This influences the tuning of the notes played by the instrument. Crescendo supports defining custom temperaments, importing them from SCALA, TUN, and KBM files, and switching between them dynamically.
    - **Keyswitch Dropdowns:** Offer a visual way to activate keyswitches, which are used to switch between different instrument settings or behaviors. Each keyswitch can have up to 32 options, and Crescendo supports a maximum of 8 keyswitches. These dropdowns provide a mouse-based alternative to using keys on the MIDI keyboard for keyswitch activation.
- **Buttons:** Initiate specific actions within the plugin.
    - **Browse:** Opens a file dialog to select and load instrument files. During loading, this button changes to "STOP" to allow interrupting the process.
    - **Reload:** Reloads the currently loaded instrument file.
    - **SoundFont Import:** Show a dedicated interface for SoundFont import.
    - **Zoom:** Adjusts the overall scaling factor of the interface. Users can cycle through different scaling levels ranging from 50% to 400% using the mouse wheel while hovering over this button.
- **Text:** Conveys various types of information within the interface. This includes:
    - **Labels:** Identify VST variables on the knobs, keyswitch options in dropdowns, and other UI elements.

    - **Program and Bank Information:** Displays the name and number of the currently loaded program and bank. This information can be hidden using the `HIDEUI` instruction.
    - **Time Signature and BPM:** Shows the current time signature and beats per minute (BPM) of the host DAW's project. Like the program and bank information, these can be hidden using `HIDEUI`.
    - **Debug Messages:** Appear in the optional log window, providing information about errors, instruction processing, and other aspects of the plugin's behavior. The verbosity of these messages depends on the `DEBUG` level setting.
- **Tooltips:** Offer concise explanations of UI elements when the mouse hovers over them. This helps users understand the function of different controls.
- **Optional Log Window:** Acts as a debugging aid and information center. It displays error messages, informative messages about instruction processing, and detailed debugging output, depending on the `DEBUG` level setting. You can customize its position and border thickness using the `DEBUG` instruction. The log window also scrolls automatically when full to ensure visibility of the latest messages.

UI Features and Customization Options

- **Textual VST Parameters:** Display the values of VST variables as text.
- **SAMPLEUI Instruction:** Enables the creation of specialized UI elements that allow end-users to swap audio samples used within the instrument at runtime. This instruction defines the sample slot to control, position of the UI element, and explanatory text to display. Users can browse for samples or drag and drop them onto the UI element. Crescendo saves user-selected samples and settings within the host DAW project.
- **Themes:** Themes (visual styles) can be enabled or disabled to change the overall visual appearance of the plugin. This feature provides a degree of customization and allows the plugin to better integrate visually with different host applications.
- **DPI Awareness:** Crescendo automatically adjusts its display based on the screen resolution to maintain appropriate sizing and readability of interface elements. The `INTERFACE`, `UIMOD`, and `SETFONT` instructions provide different levels of control over interface scaling.
    - `INTERFACE`: Defines the initial size of the plugin window in pixels. These dimensions are scaled according to the user's system font size and DPI settings.
    - `UIMOD`: Allows modification of the size and position of individual interface elements, such as knobs, dropdowns, and text labels. The values provided in this instruction are also scaled based on the system's DPI.
    - `SETFONT`: Enables users to change the font used for text, buttons, and other UI elements. The specified font size is in screen units, with one unit representing half a pixel.
    - **Zoom Button:** Provides a manual way to adjust the overall scaling factor of the interface, independent of system DPI settings. Users can use the mouse wheel while hovering over this button to cycle through scaling levels from 50% to 400%.
    - `DPIAWARE`: Offers the option to force DPI awareness on the host process. This can be helpful when the host application itself doesn't properly support DPI awareness.
- **Customization Options:** Crescendo provides numerous ways to personalize the UI:
    - **Adjusting Knob Appearance:** You can change the diameter and border thickness of knobs using the `UIMOD` instruction. This allows for fine-tuning their visual appearance and creating a more visually appealing layout.
    - **Font and Color Customization:** The `SETFONT` instruction allows for changing the font face, height, and weight used for text, buttons, and other elements. Additionally,

the `UIMOD` instruction provides options for customizing the colors of various UI elements, including the background, text, log window, and knobs.
    - **Hiding UI Elements:** The `HIDEUI` instruction enables selective hiding of interface elements, such as program and bank information, time signature, BPM display, and the entire information section. This can declutter the interface, free up space for more essential controls, or create a more minimalist look.

User Interaction: Keyboard and Mouse Gestures

- **Keyboard Navigation:**
    - **Tab:** Moves focus between interactive elements (knobs, buttons, dropdowns).
    - **Shift + Tab:** Moves focus backward between elements.
    - **Arrow Keys:** Navigate through menus and options within dropdowns.
    - **Page Up:** Increases the value of a focused knob by 1/100th.
    - **Page Down:** Decreases the value of a focused knob by 1/100th.
    - **Home:** Sets a focused knob to its minimum value.
    - **End:** Sets a focused knob to its maximum value.
- **Mouse Interaction:**
    - **Click:** Activates buttons and sets focus to knobs and dropdowns.
    - **Drag:** Modifies the value of a focused knob. Dragging farther away from the initial click point results in accelerated value changes for quicker adjustments.
    - **Mouse Wheel:** Adjusts the value of a knob in 1/100th increments when hovering over it, even if it's not in focus. This allows for precise fine-tuning without having to click on the knob first.
    - **Ctrl/Shift + Click:** Resets a knob to its initial value, as defined in the instrument file or the host DAW project.
- **Drag and Drop:** Supported for loading audio files, SCALA files, TUN files, instrument files, and SoundFont files. Users can drag these files onto the interface to load them into the plugin.

UTF-8 Support

Crescendo supports UTF-8 encoding for instrument files, allowing for the use of extended characters in temperaments names, keyswitch names and options, program names, and VST var strings. However, the current interface only supports UTF-8 internally. While the plugin's internal interface can handle UTF-8, the display of these characters in the host DAW's interface depends on the DAW's UTF-8 support. For instance, Ableton 9.7.5 may display garbage characters if non-ANSI characters are used in VST var names. It's essential to check the UTF-8 compatibility of the specific host DAW being used. TAPE files in Crescendo do not support UTF-8, as they only contain numbers and commas, and ANSI encoding is sufficient for them.

Crescendo's UI prioritizes a balance between functionality and usability while offering a good amount of customization. The developers have actively addressed UI challenges and incorporated user feedback to enhance the overall experience. The UI elements and their associated features contribute to a streamlined workflow for creating, controlling, and manipulating complex instruments.

Error Handling and Debugging

The DEBUG Instruction: A Powerful Debugging Tool

The `DEBUG` instruction plays a central role in Crescendo's debugging capabilities. It enables a log window that displays information about the compilation and execution of instrument files, with different levels controlling the verbosity of the output. This allows developers to selectively view the information needed to diagnose and resolve issues in their code.

DEBUG Levels and their Significance:

- **Level 0:** No log window is displayed, making this level ideal for final instrument versions or situations where debugging is not needed.
- **Level 1:** Displays only error messages, informing developers about syntax errors, undefined variables, and other problems that prevent successful compilation.
- **Level 2:** Presents informative messages about instruction interpretation, settings, and MIDI events, alongside any error messages. This level aids in understanding the processing flow of the instrument file.
- **Level 3:** Activates verbose debugging messages, including insights into the Reverse Polish Notation (RPN) parser used by Crescendo, and detailed information about MIDI messages.
- **Level 4 and Above:** Displays highly detailed debugging output, focusing on incoming and outgoing MIDI messages. This can help analyze MIDI data flow and interactions.

Real-Time Debugging:

The set `DEBUG` level also applies to real-time messages during runtime. Since these messages can fill the window buffer rapidly, it's recommended to debug small sections of code or single lines at a time. Developers can strategically reposition `DEBUG` instructions within their code to focus on specific areas of interest during runtime.

Common Error Scenarios and their Causes

There are common error scenarios that developers might encounter when working with Crescendo. Understanding these scenarios is key to efficient debugging and troubleshooting.

Types of Errors:

- **Syntax Errors:** These result from incorrect usage of keywords, functions, operators, punctuation, or separators. Syntax errors cause the affected line to be ignored during compilation, and a warning message is displayed in the log window if the `DEBUG` level is above zero.
- **Undefined Variables:** Occur when a variable is referenced before it has been declared or assigned a value. Similar to syntax errors, the affected line is ignored, and a warning message appears if debugging is enabled.
- **Logical Errors:** These involve mistakes in the logical flow or algorithms within the instrument file. While the instrument might compile without errors, it could exhibit unexpected or incorrect sound or behavior. High DEBUG levels expose the inner working of the Reverse Polish Notation (RPN) parser. Instrument developer can examine the expression parsing and the order of execution of the functions in search of possible missing parenthesis or other logical errors.

Using Comments as a Debugging Aid

Comments in Crescendo instrument files are not merely explanations but also valuable debugging tools. By commenting out specific sections of code, developers can isolate potential problem areas and test different parts of their instruments or effects.

Additional Debugging Aids

Special Debug Messages: The `DEBUG` instruction's level setting controls the type and amount of debug messages displayed in the log window.

- **FFMPEG Caching:** Crescendo caches files converted by the FFMPEG library (if installed) in the `<My Documents>\Crescendo\Cache` directory, using file size and modification date/time in the cached filenames. This can speed up debugging by reusing previously converted files. The debug log displays information about FFMPEG and cache operations, helping identify issues with conversion or caching.
- **NaN Protection:** Crescendo includes measures to protect against NaN (Not a Number) values, which can disrupt audio processing. It attempts to reset the stores of reverbs and other delays in the `POST` step if the output is affected by NaN values. Additionally, the stores of the `POST` step are reset when the VST is suspended. Messages related to these events may appear in the debug log.
- **SoundFont Import:** In the dedicated importing interface detailed messaged are shown during importing. A DEBUG instruction with a logging level greater than 1 in Settings.ini allows more verbose logging during SoundFont import.

Performance Optimizations

Here there is a detailed overview of the performance optimizations employed in Crescendo. These optimizations aim to ensure smooth operation and prevent CPU bottlenecks, even with complex instruments and effects.

Pre-Compilation Stage

The instrument is pre-compiled in an intermediate language efficiently interpreted.

Moreover the prefiltering of sample slots is performed at this stage.

By completing these operations beforehand, Crescendo avoids redundant calculations during runtime, improving overall performance.

Aggressive Voice Release

To manage polyphony and avoid excessive CPU load, Crescendo uses an aggressive voice release strategy. This means:

- Inactive voices (notes that are no longer being played) are released promptly.
- This frees up resources for new notes and ensures smooth performance, even with high polyphony settings.

Fast Oscillator Functions

Crescendo includes fast oscillator functions that are specifically designed for efficient execution. These functions:

- Are optimized to minimize CPU usage, allowing for the use of multiple oscillators without significantly impacting performance.
- Support default modulations, providing a balance between speed and flexibility.

Simplified Functions for Common Tasks

Crescendo offers simplified functions for common audio processing tasks. For example:

- `FILT0` applies a filter using the default cutoff frequency (`DEFAULTFC`), which is faster than using the more general `FILT` function, which requires the cutoff frequency to be specified as a parameter.

This approach reduces computational overhead by streamlining frequently used operations.

Envelope Status Precision

To avoid roundoff errors with long decay or release times, the precision of envelope statuses was increased. This involved:

- Using a higher-precision data type (e.g., double-precision floating-point) to represent envelope states internally.

This change enhances the accuracy of envelope behavior, particularly for sustained or slowly decaying sounds.

Mono and Single Options for `OSCG`:

The `OSCG` function, a versatile oscillator, supports options for mono output and single phase/frequency/amplitude. These options enable more efficient processing when stereo output or multiple modulation sources are not required.

FFMPEG Caching

If FFMPEG is installed, Crescendo caches files converted by FFMPEG in the `<My Documents>\Crescendo\Cache` directory. This caching mechanism:

- Uses file size and modification date/time in the cached filenames to identify and reuse previously converted files, speeding up the loading process.
- Displays information about FFMPEG and cache operations in the debug log, assisting developers in troubleshooting conversion or caching issues.

NaN Protection

Crescendo includes measures to protect against NaN (Not a Number) values, which can disrupt audio processing. These measures involve:

- Attempting to reset the stores (internal memory buffers) of reverbs and other delays in the `POST` step if the output is affected by NaN values.
- Resetting the stores of the `POST` step when the VST is suspended.
- Logging messages related to these events in the debug log, if enabled.

These actions prevent NaNs from propagating through the audio processing chain and causing audible artifacts.

Code Restructuring and Optimization

Code restructuring and optimization are consistently highlighted as key strategies for performance enhancement in Crescendo. The developer regularly undertakes efforts to:

- Streamline algorithms for greater efficiency.
- Reduce code redundancy to minimize unnecessary processing.
- Optimize critical code paths, focusing on areas that significantly impact performance.

The changelog entries frequently mention "performance enhancements" or "performance optimizations," indicating a continuous process of code refinement.

Compiler Optimizations

Crescendo is frequently recompiled with the latest versions of Visual Studio. This leverages the compiler's optimization capabilities, which translate to:

- Improved code execution speed, making the plugin more responsive.
- Potentially reduced binary size, though the sources don't explicitly mention this.

AVX Versions of Crescendo

Crescendo is available in different versions compiled to utilize various AVX instruction sets, which are extensions to the x86 instruction set architecture. These extensions enable processors to handle Single Instruction, Multiple Data (SIMD) operations more efficiently, leading to performance gains in applications like audio processing.

The available AVX versions are:

- **Vanilla:** This version functions on all x64 CPUs because it doesn't rely on any specific AVX extensions.
- **AVX:** This version leverages the AVX instruction set.
- **AVX2:** This version utilizes the AVX2 instruction set, offering more advanced capabilities compared to AVX.
- **AVX512:** This version is compiled to utilize the AVX512 instruction set, which provides the highest performance optimization among the available versions.

Choosing the appropriate version depends on the user's CPU. Users should select the version that aligns with the highest AVX instruction set supported by their processor to maximize performance. For instance, if a user's CPU supports AVX2, they should opt for the AVX2 version of Crescendo.

Benefits of AVX Optimizations

AVX instruction sets allow Crescendo to perform calculations more efficiently, leading to several performance benefits:

- **Faster Processing:** AVX instructions enable the plugin to process audio signals more quickly, reducing CPU load and improving overall performance.
- **Increased Polyphony:** The performance gains from AVX can allow users to use Crescendo with higher polyphony settings, enabling more complex and layered sounds.
- **Reduced Latency:** AVX optimization can help to minimize latency, resulting in a more responsive and immediate playing experience.

The different AVX versions of Crescendo are included in the plugin's distribution package. Users can choose the appropriate version based on their CPU's capabilities.

SoundFont Generated Code Optimization

Performance enhancements were specifically targeted at the code generated during SoundFont import. This ensures efficient playback of instruments derived from SoundFonts. This consisted in layer merging and simplification to reduce processing overhead during SoundFont playback.

UI Redrawing Optimization

Excessive redrawing of UI elements was identified as a potential performance bottleneck. The developer addressed this through various optimizations, including:

- Reducing the frequency of full window redraws, especially in host applications that don't handle window resizing efficiently.
- Optimizing UI element updates, minimizing unnecessary redrawing operations.

These measures contribute to a smoother and more responsive user interface.

Other topics

SoundFont Import: Bridging the Gap with Legacy Libraries

Crescendo includes a dedicated feature for importing SoundFont 2.04 (.sf2) files, a common format for storing sampled instruments. This feature converts SoundFont instruments into a format playable within the Crescendo environment, broadening the range of sounds available to users. The SoundFont import feature is activated through a button labeled **"SF2 Bulk Import."** Or drag and dropping a SoundFont file into the Crescendo UI.

Import Process:

The import process creates a series of files and directories to store the converted data:

- **"Imported" Directory:** This directory is created within the `<My Documents>\Crescendo` directory if it doesn't exist.

- **SoundFont Subdirectory:** A subdirectory is created within the "Imported" directory, named after the imported SoundFont file (without the .sf2 extension).
- **"samples" Subdirectory:** Located within the SoundFont subdirectory, this subdirectory holds the extracted samples in WAVEFILE format (16-bit or 24-bit, depending on the source).
- **Preset Files:** For each preset in the SoundFont file, a separate text file (.txt) is created within the SoundFont subdirectory. These files are named `<preset_number>_<preset_name>.txt` and contain the code to emulate the corresponding preset in Crescendo.
- **"00000_ALL_INSTRUMENTS.txt" File:** This file contains all instruments from the SoundFont file. It also includes controls for selecting instruments using MIDI Program or Bank controls or dedicated knobs in the Crescendo interface. In this file all 16 channels can be used and if you use the program and bank MIDI CC and the pedals, 16 separate settings are recognized. But the file's knobs are linked to the channel 0.
- **"00000_ALL_INSTRUMENTS_16.txt" File:** This file is similar to the previous, but you will be able to automate all 16 channels with your DAW, because it includes program, bank and pedals for all 16 channels. Be aware that this file has up to 69 VST VARS and some knobs may not be usable by your DAW (e.g. Ableton has a limit of 64 VST VARS).

Log Window:

During the import process, the log window provides information about the conversion process. The amount of information displayed depends on the debug level set using the `DEBUG` instruction.

- **Level 1:** Displays error messages.
- **Level 2:** Displays informative messages about the import process alongside error messages.
- **Higher Debug Levels:** Provide more detailed debugging messages.

Unsupported Features:

While Crescendo supports many SoundFont features, some are not currently implemented. These limitations include:

- **Chorus** is not supported.
- **Reverb gain is not automated**, but a series of VSTVARs to control it are set-up.
- **Linked modulators** (a modulator that modulates another) are not supported.
- Some default modulators are not honored for performance reasons: only Velocity -> Attenuation and Fc, Pitchwheel -> Pitch, but if they are present in the file they will be honored.
- These parameter are regularly applied, but modulations on them are ignored:
    - start/end/loop points, loop type
    - keytrack (called scaleTuning)
    - override of velocity, key number, rootkey

Accessing Imported Presets:

It is important to note that imported preset files are not automatically loaded. To use them, you must manually open them using the "Browse" button in the Crescendo interface.

FFMPEG Support

Crescendo is a programmable VST plugin that supports a wide range of audio file formats. To expand its compatibility beyond natively supported formats, Crescendo integrates with FFMPEG, a free and open-source software suite for multimedia handling. This integration enables Crescendo to extract audio from various media file formats that it might not directly support.

FFMPEG Detection and Utilization

When Crescendo loads, it searches for the FFMPEG executable (`FFMPEG.EXE`) in specific locations on the user's system. If found, Crescendo stores the file path and uses FFMPEG to convert unsupported audio files into compatible formats.

FFMPEG search locations:

- `<My Documents>\Crescendo\`
- `<My Documents>\Crescendo\FFMpeg\`
- `<My Documents>\Crescendo\FFMpeg\x64`
- `<My Documents>\Crescendo\FFMpeg\bin`
- `<My Documents>\Crescendo\FFMpeg\x86`

Audio File Conversion

When a user attempts to load an audio file, Crescendo first checks if the file format is directly supported. If not, and if FFMPEG.EXE has been located, Crescendo utilizes FFMPEG to convert the audio file to a supported format. This process allows the extraction of the first audio track from any file format supported by FFMPEG, including video files.

Caching Converted Files

To enhance efficiency, Crescendo caches the files converted by FFMPEG in the `<My Documents>\Crescendo\Cache` directory. The cached file names incorporate the file size and last modification date and time to facilitate the reuse of previously converted files, speeding up subsequent loading attempts.

Debugging and Log Messages

Crescendo provides debugging features to assist developers in understanding and troubleshooting FFMPEG integration.

Debugging features related to FFMPEG:

- The debug log displays information about FFMPEG and cache operations, aiding in debugging conversion or caching issues.
- The log window indicates if FFMPEG was found and how it's being used.

Benefits of FFMPEG Integration

- **Expanded Media File Support:** Enables Crescendo to work with a wider array of audio formats, including those commonly found in video files.
- **Streamlined Workflow:** Users can directly load various media files without the need for separate conversion steps.
- **Efficient Use of Resources:** Caching converted files minimizes redundant processing and speeds up loading times.

Pictorial depiction of looping modes

Here it is the graphic representation of the various sampled data looping mode.
<relstart>, <startw>, <endw>, <loopstartw>, <loopendw>, <looped> and <direction> allow to specify the following behaviors:

One shot with normal or separate release, forward and backward

```
Forward
|-----------------|===>>=========|===>>=========|----------------------|
0                 startw          relstart*       endw            nsamp  *ignored if normal release
Backward
|-----------------|=========<<===|=========<<===|----------------------|
0                 endw            relstart*       startw          nsamp  *ignored if normal release
```

One shot with normal or separate release, forward+backward and backward+forward
Forward + backward: the selected samples are played first forward and then backward. The eventual separate release is played in the reverse stage and can span maximum half length

```
|-----------------|===>>====<<===|=========<<===|----------------------|
0                 startw          relstart*       endw             nsamp  *ignored if normal release
```

Backward + forward: the selected samples are played first backward and then forward. The eventual separate release is played in the forward stage and can span maximum half length

```
|-----------------|===>>=========|===>>====<<===|----------------------|
0                 endw            relstart*       startw           nsamp  *ignored if normal release
```

* NOTE: if the sample arrives at endw before release triggers, when release is triggered and relstart>0, meaning separate release, that release is not played.

Loop with normal release, forward
Samples are played starting from startw, forward, then until loopendw and suddenly from loopstartw again going forward, and in loop. Crossfade near loopendw.

```
|-------|====>>===||==========>>===========||-----------------|--------|
                  ^^------------------<<-----------------||
0    startw    loopstartw                loopendw        endw    nsamp
```

Loop with normal release, backward
Samples are played starting from startw, backward, then until loopendw and suddenly from loopstartw again going backward, and in loop. Crossfade near loopendw.

```
|-----|--------||==========<<===========||=====<<===|-------------------|
               ||------------------>>-------------------^^
0   endw   loopendw                 loopstartw    startw        nsamp
```

Loop with normal release, forward+backward
Samples are played starting from startw, forward, then until loopendw, then backward until loopstartw and again forward+backward in loop. Crossfade near loopstart and loopendw.

```
|-------|====>>===||===>>=============<<===||-----------------|--------|
0    startw    loopstartw                loopendw        endw    nsamp
```

Loop with normal release, backward+forward
Samples are played starting from startw, backward, then until loopendw, then forward until loopstartw and again backward+forward in loop. Crossfade near loopstart and loopendw.

```
|-------|---------||===>>=============<<===||============<<===|--------|
0     endw     loopendw               loopstartw              startw    nsamp
```

Loop with separate release, forward
Samples are played starting from startw, forward, then until loopendw and suddenly from loopstartw again going forward, and in loop. Crossfade near loopendw.

When release triggers, the samples from relstart to endw are played forward, crossfaded with the loop that continues and fades out.
The relstart position is indicative: it can be anywhere between endw and startw. Typically is after loopend to have a totally separate release, e.g. using a single sample file with a sustain that is looped and jumping suddenly to the release stage at release, e.g. bow noise of a violin.

```
                                                          relstart
|-------------------------------------------------------------------|===>>=====>>====|--------|
|-------|====>>===||==========>>===========||--------------------------------|--------|
                          ^^-----------------<<-------------------||
0    startw        loopstartw                              loopendw                endw     nsamp
```

Loop with separate release, backward
Samples are played starting from startw, backward, then until loopendw and suddenly from loopstartw again going backward, and in loop. Crossfade near loopendw.
When release triggers, the samples from relstart to endw are played backward, crossfaded with the loop that continues and fades out
The relstart position is indicative: it can be anywhere between endw and startw. Typically is after loopend to have a totally separate release, e.g. using a single sample file
with a sustain that is looped and jumping suddenly to the release stage at release, e.g. bow noise of a violin.

```
             relstart
|---|===<<===|--------------------------------------------------------------------------------|
|---|--------------------||==========<<===========||=====<<===|-------------------|
                          ||------------------>>------------------^^
0   endw          loopendw                              loopstartw    startw           nsamp
```

Loop with separate release, forward+backward
Samples are played starting from startw, forward, then until loopendw, then backward until loopstartw and again forward+backward in loop. Crossfade near loopstart and loopendw.
When release triggers, the samples from relstart to endw are played forward, crossfaded with the loop that continues and fades out
The relstart position is indicative: it can be anywhere between endw and startw. Typically is after loopend to have a totally separate release, e.g. using a single sample file
with a sustain that is looped and jumping suddenly to the release stage at release, e.g. bow noise of a violin.

```
                                                            relstart
|-------------------------------------------------------------------|===>>=====>>====|--------|
|-------|====>>===||===>>=============<<===||--------------------------------|--------|
0    startw         loopstartw                           loopendw                endw     nsamp
```

Loop with separate release, backward+forward
Samples are played starting from startw, backward, then until loopendw, then forward until loopstartw and again backward+forward in loop. Crossfade near loopstart and loopendw.
When release triggers, the samples from relstart to endw are played backward, crossfaded with the loop that continues and fades out
The relstart position is indicative: it can be anywhere between endw and startw. Typically is after loopend to have a totally separate release, e.g. using a single sample file
with a sustain that is looped and jumping suddenly to the release stage at release, e.g. bow noise of a violin.

```
            relstart
|---|===<<===|---------------------------------------------------------|
|---|------------------||===>>=============<<===||=============<<===|-----------|
0   endw         loopendw                          loopstartw                     startw     nsamp
```

Detailed Instruction Explanations

Instructions

Initializations Instructions

VSTVARS <number of vstvars>

It specifies the number of VST VARs that can be made available to the host.
Between 1 and 128 in the current version. Default 20.
If an invalid number or a string is specified, then 20 is assumed.
If a number greater than 128 is specified, then 128 is assumed: no error is given.
Higher numbered VST VARs can in any case be accessed in the instrument file but they keep their initial value if not linked to temperament or MIDI CC, they just can't be automated by the host or set manually in the GUI, but only in the instrument file (if not set, the default value of a VST var is zero).
If they are linked to temperament or a MIDI CC, even if hidden, they will reflect the linked object (see MIDICC or TEMPERAMENTCC).

WARNING:

- Some DAWs support less than 128 parameters (e.g. Ableton Live supports 64 parameters max).
- Higher numbered VST vars can still be used, to store hidden constants or to change the scale of a MIDI CC (see MIDICC below).
- Some DAWs (e.g. Ableton) read this parameter only at VST start, moreover, they can cache the value in the own saved file, so even if you change this value in Settings.ini, loading an old file will have the new value in the VST interface and the old value in the DAW interface:
    - to solve you must delete and reinsert the VST and load again the instrument file, but so you lose all the personalizations to the parameters.

INTERFACE <size x>;<size y>

This option sets the interface size in pixels.
Usually it is in the configuration file, but can be specified in the instrument file to e.g. increase the visible space for accommodating more drop boxes or knobs.

NOTE: some checks are performed to assure that all VST vars and keyswitches are visible.
Be aware that in debug mode there can be some overlapping.

NOTE: LMMS (version 1.2.2 tested) fixes the interface size at the first VST loading.
So if you increase the height or width in Settings.ini the interface is clipped. If you reduce them there will be blank space.
Next time the VST is loaded the problem is fixed.

The upper size limits are to fill the whole screen with 50 pixels reserved for a title bar/status bar.

<size x>;<size y> is the size in pixels, of the GUI window.
If you don’t specify <size y>, the old value is kept.
If you set <size y> to a value less than 0, the old value is overwritten with -<size y> if -<size y> is greater than the old value. This to assure a minimum height.

If you set <size x> to a value less than 0, the old value is overwritten with -<size x> if -<size x> is greater than the old value. This to assure a minimum width.
This is to be able to modify only one dimension.

This instruction assumes that the primary font is 16 DLPs (8 points), the default size and the monitor is 96 DPI. All values are scaled accordingly if not true.
E.g. if font is 32 DLPs and the monitor is 192 DPI, these values are multiplied by 4.
This to have coherent relative dimensions.

IO <numInputs>, <numOutputs>

Configure the number of inputs and outputs that the VST has.
The minimum number of inputs is 2 and the maximum 101. Default 2.
The minimum number of outputs is 1 and the maximum 100. Default 1.
Out of range values are clipped.
If the Host does not allow the change (ioChanged() returns false), the values are reverted to the previous values. In this case it's advisable to put this instruction in the Settings.ini file. All the VSTs will see that number of inputs and outputs, but they can be safely ignored if not needed. The overhead to have more inputs and outputs than necessary is minimal, provided that you don't connect inputs or outputs that are not used in the instrument file.

DPIAWARE <flag>

Force DPI_AWARENESS in the host process if <flag> is not ZERO. This is done only at VST loading. However, using this instruction might not be compatible with all DAWs and should be used cautiously.

DEBUG <level>;<log_pos>;<log_border>

Sets the debug level, the debug log window position and debug log window border.

<level> can be any integer greater or equal to zero. Default 0.

<log_pos> is optional and defaults to 50%, or the value set with a previous DEBUG instruction. It is limited between 20% and 80% and defines the horizontal starting point of the log window in terms of percentual of window width.
If you set to 81 and higher, the log window is horizontal and under the knobs, enlarging the window. If the window is too high, the log window is reduced to fit in the screen (see INTERFACE).
If the final log height is below 80 pixels, then the log window is switched to vertical and 50%.

<log_border> is optional and default to 5 pixels, or the value set with a previous DEBUG instruction. Maximum 60 pixels. It is the border of the log window.

When debug level is 0, no log window is drawn. Only the knobs, drop boxes and other essential controls.
When debug level is 1, a log window will appear, containing eventual error messages (if any).
When debug level is 2, the log window contains also more informative messages on the accepted lines (like QUALITY set to xxx, NEW TEMPERAMENT added, MIDI CC set to xxx, etc..) and a list of the operations performed by the expressions of accepted lines.

After the file is loaded, during play mode, all MIDI CC change messages are dumped on the log. Moreover when a drop box is modified or a KEYSWITCH is selected or a TEMPERAMENT change modifies a MIDI CC or VST VAR, an entry in the log is added.
When debug level is 3, all debug messages are on. These are debug messages concerning more details of the Reverse Polish Notation parser. More details on the MIDI messages are given.
When debug level is 4, more details on the incoming and outcoming MIDI messages are given.
Values greater than 4 are reserved for more verbose debugging. Currently they act the same as level 4.
The debug level can be changed anywhere in the instrument file, but the last DEBUG instruction will determine if the log window is drawn: if the level is greater than zero the windows is drawn.
Multiple changes of the level affects what messages are ultimately put in the log window: if you want only debug a section, change the default debug level around it and remember to put a last DEBUG instruction with level > 0.

The last DEBUG level set is used also at run-time for the real time messages.
Since the messages can quickly fill the window buffer, it is advisable to debug few or one line a time and then debug the runtime, moving the DEBUG instructions around the target lines.

INCLUDE "filename"

Include the specified file. All included files should reside in <My Documents>\Crescendo, but full or relative path for the files are supported.

There is a limit on deepness to avoid recursive calls. Currently is 20.

QUALITY <samples>;<ww>;<type>;<exponent>;<oversampling>

This option sets the quality of the sample interpolation.
Usually it is in the configuration file, but can be specified in the instrument file to e.g. lower the quality if the instrument is very complex and CPU intensive, or to use a different window if it sounds better with the samples you are using.

Values outside the given range are clipped. No error is given.

<samples> number of samples in which the window is stored. Default 4096. Allowed values: between 256 and 65536. For performance reasons should be power of two.

4096 uses 16KB of L1 cache. Do not increase too much unless necessary.
Sinc is symmetric so this value is the storage buffer size of half a sinc (<ww> lobes, see next option).

The windowed sinc interpolation is performed with nearest neighbor, so the more the samples, the more precise will be the interpolation.

<ww> gives the window width in terms of sinc lobes.

- For each output sample we need to perform 2*ww+1 calculations: -ww...0...+ww.
  This has to do to how the interpolation is performed: the interpolation window is a windowed and truncated sinc.
- The more the points, the higher the interpolation quality, but the higher the CPU usage.

- The lower the ww, the worse the sample will sound especially when played at pitches lower than the original pitch.
- Zero means nearest neighbor interpolation. In this case it's advisable to avoid playing at frequencies below the original.
- Default: 7. Range 0-100.

<type> is the type of window used for the interpolation. For each output sample we need 2*ww+1 original samples, windowed with this function:

- 0 = sinc with Hamming window.
  windowedSinc(X) = sinc(X) * (25/46 - 21/46 * cos(PI * (1 + X / ww)) ^ exponent
- 1 = sinc with Hann window.
  windowedSinc(X) = sinc(X) * (0.5 - 0.5 * cos(PI * (1 + X / ww)) ^ exponent
- 2 = sinc with Cosine window.
  windowedSinc(X) = sinc(X) * (cos(PI * X / ww / 2)) ^ exponent
- 3 = sinc with Lanczos window.
  windowedSinc(X) = sinc(X) * (lanczos(X)) ^ exponent
- 4 = sinc with Gauss window.
  windowedSinc(X) = sinc(X) * (e ^ (- (X / ww * exponent) ^ 2))
- 5 = sinc with NO window.
  windowedSinc(X) = sinc(X)
- 6 = sinc with Kaiser window.
  windowedSinc(X) = sinc(X) * (I0(exponent*sqrt(1-(X/ww)^2)) / I0(exponent))

With:

- sinc(X) = sin(PI * X) / (PI * X)
- lanczos(X) = sin(PI * X / ww) / (PI * X / ww)
- I0(X) is the approximation of the 0th-order modified Bessel function of the 1st kind

Default: 0
> Note: 5 is the one with the higher frequency content but it is the one with high ringing artifacts risk, if the original sample has full bandwidth.
> To cut as less high frequencies possible and avoid most ringing artifacts, use the other windowing options with an higher ww and/or lower exponent (not for Kaiser window).
> <ww> = 7 is a good compromise.

<exponent> is the exponent of most of the windowing type or the Beta coefficient for the Kaiser window.

- Default=1. Range 0.000001-100, inclusive.
- Higher exponents give warmer sound and less ringing artifacts, but you lose some high frequencies.
- The lower the exponent, the more the window resemble type 5 and so the higher is the frequency content, at expense of ringing artifacts risk.
- Exponent can't be exactly zero: to emulate exponent zero for all windows type, use <type> = 5.
- If <type> = 5, exponent does not have effect.
- For a true sinc interpolation, <type> = 5 and a very high <ww> (theoretically infinity) is needed.
- If <type> = 6 exponent is repurposed as the beta coefficient. Given a sideband attenuation above 50dB the beta to use is 0.1102*(<attenuation>dB - 8.7)

<oversampling> number of sub-samples to calculate and average, to increase the interpolation quality. Default 1. Allowed values: between 1 and 20.

NOTEs:

- high values can cause slower loading speed if using SAMPLES or RENDER and can cause high CPU use for oscillators at runtime.
- Oversampling is **NOT** applied for PSHIFT, PSHIFT2, WAVETABLE, WAVESCAN and GRAINSYNTH or if using Nearest Neighbor interpolation.

UIMOD <UI_item_number>; <P1>; … <P9>

Sets some parameters for some user interface items.
It allows moving or resizing some elements or changing some colors.

This instruction assumes that the primary font is 16 DLPs (8 points), the default size and the monitor is 96 DPI.
All positions and sizes are scaled accordingly if not true.
E.g. if font is 32 DLPs and the monitor is 192 DPI, these values are multiplied by 4.
This to have coherent relative dimensions.

<UI_item_number>: allows selecting the interface element to be modified.

- 0 Default knob diameter (pixels).
  Syntax: UIMOD 0; <diameter>
  Minimum knob diameter is 30 actual pixels and the maximum diameter is 200 scaled pixels.

- 1 Enable or disable Themes (visual styles).
  Syntax: UIMOD 1; <flag>
  If <flag> is below 1, the Themes are disabled. If <flag> is 1 or greater, Themes are enabled.

- 2 Default knob border thickness (pixels).
  Syntax: UIMOD 2; <thick>
  Precision is 0.1 pixels.
  If less than 0 use a default value.
  If >=0, scale for the current zoom factor and apply to the knob.

- 400-527 VST parameter knob #0-127.
  Syntax:
  UIMOD
  <#>,<xpos>,<ypos>,<diameter>,<lineR>,<lineG>,<lineB>,<fillR>,<fillG>,<fillB>,<thick>
  <xpos>,<ypos> position. If negative use default.
  <diameter> Knob diameter is capped to 30 actual and 200 scaled pixels.
  If not set or set to a negative value, use the default.
  <lineR/G/B> color of the line and dots. If any is negative use default (see 714)
  <fillR/G/B> color of the internal of the knob. If any is negative use default (see 713)
  <thick> knob border thickness (pixels). Precision is 0.1 pixels. If less than 0 use a default value. If >=0, scale for the current zoom factor and apply to the knob.

- 600-608 Drop box texts 0-8. 0 is the temperament, 1-8 is the keyswitch number 1-8.
  Syntax: UIMOD <#>,<xpos>,<ypos>,<width>
  <xpos>,<ypos> position. If negative use default.

<width> If not set or set to a negative value, use the default.

- 700-708 Drop boxes 0-8. 0 is the temperament, 1-8 is the keyswitch number 1-8.
  Syntax: UIMOD <#>,<xpos>,<ypos>,<width>
  <xpos>,<ypos> position. If negative use default.
  <width> If not set or set to a negative value, use the default.

- 709 Default background color.
  Syntax: UIMOD 709,<R>,<G>,<B>

- 710 Default text color.
  Syntax: UIMOD 710,<R>,<G>,<B>

- 711 Default log window background color.
  Syntax: UIMOD 711,<R>,<G>,<B>

- 712 Default log window text color.
  Syntax: UIMOD 712,<R>,<G>,<B>

- 713 Default knob fill color.
  Syntax: UIMOD 713,<R>,<G>,<B>

- 714 Default knob line color.
  Syntax: UIMOD 714,<R>,<G>,<B>

Other values have no effect, but no error is given.

Note: button and drop box colors are not changed.
RGB values are between 0 and 255.
If any of the RGB value is negative, then the whole color is set at the default.
Use this feature to assure to have default color even if you included some unknown file.

NOTE:
Knob label font can be different (see SETFONT below): this does not affect the scaling but affects the auto positioning of the knobs.

SETFONT <font name>, <height>, <weight>, <knob font height>, <knob font weight>

Select a font for texts, buttons etc.
True type fonts are searched.
Fonts with only OEM code page cannot be selected (e.g. Modern, Script).

<height> is in screen unit that is half a pixel. Maximum 48 units. Default 16 units.
Negative values specify alternative size (see SetFont help on MSDN).

<weight> is rounded in units of 100 and can be between 0 and 900, default 900:
0 don’t care, 100 thin font, 900 extra bold font.

<knob font height> is in screen unit that is half a pixel. Maximum 48 units.
Negative values specify alternative size (see SetFont help on MSDN).
It’s the font size of the knob’s labels (name and values).
If not specified, defaults to the value for <height>.

<knob font weight> is rounded in units of 100 and can be between 0 and 900:

0 don't care, 100 thin font, 900 extra bold font.
It's the font weight of the knob's labels (name and values).
If not specified, defaults to the value for <weight>.

HIDEUI <option>

Allows hiding some information to have more space for VST VARs. Default = 4.
<option> = 0, means nothing hidden.
<option> = 1, means program and bank information are hidden.
<option> = 2, means program, bank and time partiture are hidden.
<option> = 3, means program, bank, time partiture and BPM are hidden.
<option> = 4, means all hidden.
<option> = 5, means all hidden, but program name, put to the left of the file name.
<option> = 6, means all hidden, but program name and polyphony.
<option> = 7, automatic, depending on debug level, program names and sequencer status.
<option> = 8, means all hidden, but polyphony, put to the left of the file name.
<option> = 9, automatic: 4 or 10, depending if there are program names defined.
<option> = 10, all hidden but a textbox with program names of all 16 channels and pedals status.

SAMPLEUI <Sample_slot>; <X_pos>; <Y_pos>; "<Text>"

Setup up an UI to let the user to swap the sample of a sample slot.
See the SAMPLE instruction below for details on sample slots.
The sample slot should contain sampled data: if the sample slot is synth, the results are UNDEFINED, but no check and no error is given.

<Sample_slot> is the sample slot number to modify with this UI. If more than one SAMPLEUI is assigned the same sample slot, the results are UNDEFINED.

<X_pos> and <Y_pos> are the coordinates of the upper left corner of the SAMPLEUI. The max horizontal size depends on the file name and explanative text length. The maximum vertical size is fixed.

<Text> is an explanative text to print in the header of the UI. Just one row.

The file can be changed with the "Browse" button or with the drag & drop feature.

Note that the pre-processing on the sample slots are ignored: the plain file is loaded.

CURVE <index>,<x1>,<y1>,...,<xN>,<yN>

Define a Look Up Table into the slot number <index>.
The interpolation of missing points is linear.
<xi> must be strictly increasing.
N must be at least 2.
<yN> must be specified (odd number of parameters).
If the slot is already occupied, it will be overwritten.
Used by CURVE, MOD2 and ENVCURVE functions (see below).

SILTH <number>

Sets the output silence threshold to which a layer slot is freed. Default 1e-7 (-140 dB).
Sound is processed in blocks by the DAW and this check is performed at the end of a block.
A slot is freed if in release mode and if MAX(ABS(L)+ABS(R)) is below the <number> in a whole sample block.
So a feedback delay, a reverb or any filter that prolongs the sound will prolong the layer life if not put in the POST step.

WARNING: a layer without any envelope (or where all envelopes are skipped by an IF ... GOTO statement), will end only if the signal fades out and the layer is in release state (note off or note off trigger).
In case of oscillators or looped samples, it will never terminate if the volume is not cut to 0 by other means (e.g. automation).
No error is given: value less than 1e-12 (-240 dB) are treated as 1e-12 and values greater than .1 are treated as .1.

MIDI Instructions, stage I: input

MIDICH <min>;<max>

Sets the range of the MIDI channels to monitor.
Default 0-15.
MIDI messages not coming from these channels are discarded.
This means that MIDI CC slots of disabled channels are left to the default value (0) and the MIDI Output does not include excluded channels.
Use the ONCHANNEL trigger to filter out channels and transmit them to the MDI Output.
The default 0-15 means interpreting all messages of all channels.

POSTCH <channel>

Sets the default channel to use when picking MIDI CC values with MCC and MCC2 in the POST step. Use MCC3 in the expression to use an alternative channel for a single MIDI CC.
Default 0.

MIDICC <number>;<initial value>;<vstvar>;[<channel>]

Initialize a MIDI CC and optionally link it with a VST VAR.

<number> the Midi CC to set/link. 0-199.

- 0-127 are standard MIDI CC.
- 128-199 are extended MIDI CC (Pitch bend, Program, etc...) that can be initialized and linked. Some extended MIDI CC are run time variables (e.g. KEY, IN, OUT), so the initialization has not effect and can not be linked to a VST VAR.
  The MIDI CC initializable and linkable are:
  - 133, 134, 135, 155: Aftertouch, Program, Pitch bend and Bank number.
  - 160-199: Unassigned numbers can be used to store constants. For future compatibility, you can assume that MIDI CCs up to 159 are assigned. 160-199 are left to the user. These values are distinguished also by channel number, so the slots are actually 40*16. See below in the manual for a possible use.
- Other MIDI CC numbers can not be initialized or linked.

<initial value> the initial value to set.

- Uninitialized MIDI CCs start with value 0, if the DAW does not automate them in some way (e.g. a DAW can have its own initial Midi CC settings).

<vstvar> VST variable to link to this MIDI CC. Range: 0-127. To avoid confusion or cut and paste errors, also the 600-727 range is recognized.

<channel> The MIDI Channel of the MIDI CC to initialize/link. Range: 0-15. Default is 0 if not specified.

- If a number off range is given, the MIDI CC is not linked to any VST VAR.
- Default -1 (MIDI CC not linked to any VST VAR).
- The MIDI CC value 0 is mapped to VST min and 127 to VST max.
- The VST scale is preserved.
  E.g. if logarithmic, the MIDI CC sets the VST VAR logarithmically.
  Even the hidden VST VARs can be linked and reflect the MIDI CC value: this can be used to e.g. transform logarithmically a linear MIDI cc, using a hidden VST VAR (e.g. the VST VAR # 127).
- If a MIDI CC changes a VST VAR linked to the temperament, the temperament is updated.
- If a VST VAR changes a MIDI CC linked to the temperament, the temperament is updated.
- If the temperament is linked to a MIDI CC linked to a VST VAR even that is updated.
- If the temperament is linked to a VST VAR linked to a MIDI CC, even the MIDI CC is updated.

The instruction updates also the initial value of the VST VAR.
IMPORTANT considerations:

- ➢ Program, bank msb, bank lsb and full bank have direct numerical mapping if are linked to a VST var.
- ➢ Full bank, after all clipping, overrides bank msb and bank lsb if linked at the same time with VST VARs.
- ➢ This instruction must go after the VSTVAR that initializes the VAR.
  VSTVAR does not update linked MIDI CC.
- ➢ The VST VAR is updated by MIDI CC messages, but any update to the VST VAR does NOT send any MIDI CC message.
  It only changes the internal MIDI CC value used by the instrument file for e.g. some automation. Any further MIDI CC message changes again both values.
- ➢ When the linked MIDI CC changes, the VST sends to the HOST DAW a message as if the user has changed the VST VAR on the GUI and so the automation may or should be paused.
- ➢ Numbers not assigned or MIDI CC never updated at run time act as a compile time constants: they can be set in the instrument file and used in the LAYERs, but they never change.
- ➢ No check is performed on <initial value>. It can be any float value and this value will be used until first MIDI CC message that overwrites it.
  This behavior can be exploited in various ways, e.g. setting a special value that causes silence until the user actually selects a valid value (e.g. checking with an IF ... GOTO statement against the special value).

VSTVAR <number>;<initial value>;"Name";"Unit";<min>;<max>;<scale type>[;<names>…]

VST variable declaration and initialization.

By default all variables have name "VST var #", unit empty, min 0, max 1, initial value 0 and linear scale.

<number> is the number of the VST variable that we are configuring, ranging from 0 to the maximum supported (currently 128). VST VARs above <VSTVARS> - 1 are not visible in the GUIs and keep the initial value given with this instruction.

<initial value> is the initial value at the file loading.
If you move the VST knob (or also the HOST slider/knob) with the host automation on, a message will be given to the host that most probably disables or pauses an eventual automation applied to the knob.

"Name" is the label given to the knob in the GUI.

"Unit" is the measurement unit of the knob (e.g. Hz, dB etc...), put after the VST variable value in the knob text. e.g. <Name> <Value> <Unit>.

<min> and <max> are the minimum and maximum value of the VST var when the knob is at the minimum and maximum position.

<scale type> is 0 for logarithmic, 1 for linear, 2 for integer, 3 for beats and 4 for integer with names.

<names> are optional names to the values for <scale type> = 4.

Notes:

- A number of knobs are drawn on the VST GUI, depending on VST configuration, numbered from 0 to <VSTVARS>-1.
- If a file accesses a VST variable above the maximum, no error is given, but the initial value is returned for that variable, since there is no way to change it (if not linked to a MIDI CC or temperament drop box).
- Accessing in play mode VST VARs above the implemented ones (currently 128) does not generate errors, but the zero value is returned. MIDI CCs form 800 and up to 999 have been assigned.
- A minimum of 12 VST variables is set by default but you can increase or decrease it with the VSTVARS declaration.
- For all scales min and max can be inverted for inverted scales.
- The only limitation is that max and min must be different values and not so close to avoid round off errors.
- For logarithmic scale min and max must be >0.
- For linear scale and integer scale, min and max can be any value.
- For integer scale, only integer values are allowed: they are rounded each time the VST VAR is modified by the user, the HOST or in the instrument file.
- For integer with names, optional names can be given at the values, instead of numbers.
  The VST variable still is numeric.
  This is similar to KEYSWITCH below.
  The names, maximum 19 characters each are given in the last parameters.
  Maximum 64 names are allowed. <max> - <min> is also checked.
  Names not specified are set to “Option #n” by default.

- For beat scale, only integer fractions or multiple of a beat are allowed, e.g. 1/32, 1/16, 3/32, 1/8, 3/16, 1/4, 3/8, 1/2, 3/4, 1, 3/2, 2, 3, 4, 5, 6, etc...
- Values are rounded to the next beat so the VST VAR can assume values from 1/32 and above. Min and max must be > 0.

KEYSWITCH "Name";<type>;<initial_value>;<key1>;<key2>;<min>;<max>;"Name 1";...;"Name N"
OR
KEYSWITCH "Name";<type>;<initial_value>;<key1>;<key2>;<min>;<max>;<channel>;"Name 1";...;"Name N"

Defines a keyswitch.
Keyswitches are useful for options with meaningful names. For numeric options it's better a VST VAR.

Each Keyswitch can have maximum 32 options.
The maximum number of keyswitches is 8.
The keyswitch can monitor the MIDI keys of a specific channel. If using the first syntax, it is assumed the channel 0.
The keyswitch values are global, so with 8 keyswitches and 16 channels, not all MIDI keyboards may host a keyswitch.

A drop box will be created in the GUI, with the given "Name" to be able to activate the keyswitches ALSO with the GUI (e.g. with the mouse).

"Name" is the name of the keyswitch, e.g. "Style".

<type> is the type of KEYSWITCH: 0=<multiple>, 1=<one key>, 2=<two keys>

> <multiple> is a range of keys from <key1> to <key2>. The keys from <key1> to <key2> do not cause note on or off events anymore, but pressing a key activate a keyswitch and deactivate the others in the group. Only the current keyswitch has a value of 1. The other values zero.
>
> <one key> the key <key1> is used to select among the values between <min> and <max>. Pressing the key increase the value by 1 and if above max it returns to min. The value of the key is the current value.
> The key <key1> does not cause note on or off events anymore.
>
> <two keys> the keys <key1> and <key2> are used to increase and decrease the value. If below min, return to max. If above max return to min.
> The current value of the keyswitch is stored in both key1 and key2.
> The keys <key1> and <key2> do not cause note on or off events anymore.

<initial_value> is the value initially selected. For <multiple> it's the key that values 1 (the others value 0). For one and two keys is the initial value of the key value.

<key2> is ignored for <one key> keyswitch.

<min> and <max> are ignored for <multiple> keyswitch.

<channel> specifies which MIDI channel to monitor. If it is not included in the MIDICH range, the keyswitch will never be moved by a key.

"Name 1"... are the names of the keyswitch setting, e.g. "Pizzicato", "Legato", etc.
Names are optional and missing values default to Option 0, 1, 2 etc.

Note: no check is performed to see if some interval of keys is overlapped among multiple KEYSWITCHes. In this case the behavior is undefined.

The MIDI CCs #400-527 are accessible in the LAYERs for automation or more probably for TRIGGERs. They are global for any LAYER and MIDI channel.

These MIDI CCs evaluate to 0 if the corresponding key is not a keyswitch or temperament keyswitch but a normal note.

A specific MIDI CC evaluates to the current temperament number, if the TEMPERAMENTCC with a number between 400 and 527 is used (see below).

A specific MIDI CC evaluates to 0 or 1 if the corresponding key belongs to a <multiple> ranged keyswitch, depending if the option associated with the key is the currently active.

A specific MIDI CC evaluates to a value between a <min> and a <max> if it belongs to an <one key> or <two key> keyswitch and reflects the current selected option.

NOTE: keys used for keyswitches do not cause note on and off, but polyphonic aftertouch are still updated and available with the extended MIDI CC. The value can be used e.g. to automate somehow the effect triggered by the key.
EXAMPLE: <multiple> keyswitch that allows selecting between vibrato, tremolo etc. and the polyphonic aftertouch on the key can be used for the effect amount in real time.

CHOKE <time>

Activate the forced release of <time> seconds for notes already triggered of the same exact key. Used to silence notes of the same key to e.g. simulate a piano string behaviour when the key is retriggered. If <time> = 0 (default), the choking is disabled.

POLY <polyphony>;<discard>;<mode>;<glidetime>

Polyphony, note triggering and gliding settings.

<polyphony> Sets the global polyphony of the VST.

- Default 256.
- This can be used to limit CPU resources in case of huge projects and/or slow CPU, to avoid sound stuttering, or to emulate physical instrument limitations.
- This can be also used to enable retrig/portato/legato effect (depending on polyphony selected).
- Each file can have its separate setting, so more complex instruments can be set to a lower value.
- In the current implementation the absolute number is capped between 1 and 256. No error is given: 0 is treated as 1 and values greater than 256 are treated as 256.

- For <polyphony> >= 0, the polyphony is global: each channel shares the same pool of <polyphony> layers. If you need different polyphonies, you should use separate layers.
- For <polyphony> < 0, the absolute value is used as the per channel polyphony, still capped between 1 and 256: Each channel can have a maximum polyphony of -<polyphony> independently of the other channels. Useful to simulate strings of e.g. a guitar with an MPE capable DAW and MIDI controller. If you need different polyphonies, you should use separate layers. The total polyphony is still capped to 256.

<discard> specifies what note to discard in case of maximum polyphony is met:

- 0 (Default if not specified) means the oldest,
- 1 means the lowest pitch,
- 2 means the highest pitch,
- 3 means the one with currently the least intensity.
  NOTE: with polyphony=+-1 the 4 methods are equivalent.

<mode> selects the operation mode of the voices, when triggering a new note:

- 0=NORMAL (Default): if there is a free voice and the channel polyphony is not exceeded or the polyphony is in global mode, a new voice will be instantiated with fresh phases, envelope positions and pitch.
  If not, an occupied one is selected with the <discard> and it will be reset to the new fresh data. The search will be performed among the notes of the same channel if the polyphony is not global.
  The envelope will be reset to the start of the attack and the frequency abruptly changed: glide time is ignored.
  Some limited fadeout before the new note trigger is made to avoid clicks.
- 1=RETRIG: if there is a free voice and the channel polyphony is not exceeded or the polyphony is in global mode, same as NORMAL.
  If not, an occupied one that is compatible is selected with the <discard>.
  If there is not a compatible voice, then it's the same as NORMAL: an occupied one with the <discard> is selected and abruptly changed. The search will be performed among the notes of the same channel if the polyphony is not global.
  The compatibility is checked by operation slot, result slot number and by MIDI channel. If the layers have the same formulas and come from the same MIDI channel, then they are compatible.
  Other layers can be compatible by chance. Don't use RETRIG for layers with different formula.
  The formulas are truly compatible only if the operations are the same, in the same order and differ only for constant values or MCC numbers or variable names.
  In particular if the formula is the same but an oscillator has sample data different from the old note, a click will most probably be heard.
  If automated, the sample numbers of the oscillators will be resampled at the new trigger, so again the sample could be changed and a click will be heard.
  The phases and envelope positions are maintained, the envelopes start the attack phase starting from the current envelope levels.

    - Note for sampled one shot data: since the phase is not reset, a long train of consecutive notes can cause the sample to eventually end.
      In this case as soon as the release phase is entered (NOTE OFF), the last note in the train will free the slot and the next note will be a fresh one.

The pitch is changed with the glide time given. If glidetime=0 then the frequency abruptly changes.
The RETRIG is effectively a LEGATO/PORTATO if polyphony is +-1 and the sustain level of all envelopes is 1.

- 2=PORTAMENTO: this mode was closely modeled on PORTAMENTO mode as implemented in the sampler of Ableton Live 9.7.5.
  If there are not notes not in release state or if glidetime=0, then it's the same as RETRIG (including the compatibility check). The notes must be in compatible slots (see above).
  Otherwise the newest note with the nearest pitch is picked, a new note is created with new phases and envelope positions and a glide with the given time to the new note is performed, starting from the original frequency.
  If the new note is stopped before the glide time is finished, an eventual new note will start gliding where the previous have finished.
  If the new note is stopped after the glide time is finished, an eventual new note will start gliding starting from a point as if the previous note was gliding back to the original note from when it received the note off.
- 3=GLIDE: this mode was closely modeled on GLIDE mode as implemented in the sampler of Ableton Live 9.7.5.
  If there are not notes not in release state or if glidetime=0, then it's the same as RETRIG (including the compatibility check). The notes must be in compatible slots (see above).
  Otherwise the newest note with the nearest pitch is picked, without changing phases and envelope positions and a glide with the given time to the new note is performed, going or remaining in sustain state.
  The new note is flagged as glided from the old note. At the note off for the new note, a check is performed to see if the note off of the old note had arrived.
  If not, then the new note is transformed in a glided note from the current frequency to the old note frequency, without resetting phases and envelopes.
- $<MCC>: the MIDI CC number <MCC> selects one of the modes above:
  For MIDI CC 0 to 127 the values 0 to 127 are linearly mapped to [0, 3]: 0-31 => 0, 32-63 => 1, 64-95 => 2, 96-127 => 3
  For MIDI CC 400 to 527 (Keyswitch on keys 0 to 127) the mapping is direct. Set min to 0 and max to 3 if you want to select all modes.
  For MIDICC 600 to 727 (VST VAR 0 to 127) the mapping is direct. Set min to 0 and max to 3 if you want to select all modes.
- Other values or other <MCC> numbers out of range: mode 0 (NORMAL).

<glidetime> is the glide time in seconds.

- If it is a real number, it is capped between 0 and 100 without giving error.
- If it is 0 the glide is disabled.
- If it is $<MCC> then:
  For <MCC> between 0 and 127: glide time is given by MIDI CC 0 to 127 value divided by 100, e.g. from 0 (disabled) to 1.27 seconds for all standard MIDI CC.
  For <MCC> between 600 and 727: glide time is given by VST VAR 0-127 (600=VST VAR 0, 727=VST VAR 127).
  The other <MCC> values mean glide disabled.
  The value provided by an eventual VST VAR is capped between 0 and 100.

NOTE: for automated <mode>, the MIDI CC is sampled at trigger time and from the same MIDI channel of the target note.
NOTE: for automated <glidetime>, the value is sampled at trigger time and from the same MIDI channel of the target note.

NOTE for polyphonic PORTAMENTO/GLIDE:

- Using a PC keyboard as key controller allows correct note off message routing only up to 4 keys pressed: in some complex scenarios, NOTE OFF messages get lost, resulting in stuck sounds.
  Thus, for polyphonic PORTAMENTO/GLIDE a MIDI Keyboard is highly recommended.

IMPORTANT NOTE FOR GLIDING/PORTAMENTO:

- The gliding works only with normal mapping between KEYF and oscillator frequency: only oscillators that uses FREQ in some way are glided.
- Oscillators that use a custom mapping based on KEYF or constant or otherwise automated frequency (e.g. LFOs) will not glide.
- You can change the KEYTRACK etc., but only with linear mapping (in dB).
- If you need gliding but different KEYTRACK only for a layer or oscillator you can change the KEYTRACK only for that layer. But if your formula can not be emulated with KEYTRACK etc., you must derive the frequency from FREQ.

PEDAL <mcc1>;<mcc2>;<minkey>;<maxkey>
SOSTENUTO <mcc1>;<mcc2>;<minkey>;<maxkey>

These commands allow configuring up to four hold pedals.
Two of them act on the entire keyboard, two of them on a programmable subset.
Two of them can be used as classic HOLD pedals and two of them as classic SOSTENUTO pedals, but not only that: the most common use is with an instrument with long decay, zero sustain and short release, but other combinations can be used because the pedals simply let the VST use the decay time as release time when enabled.

A declaration overwrites a previous declaration, so to be sure pedals are OFF when including unknown files, a PEDAL -1;-1 and a SOSTENUTO -1;-1 should be included in the file.
MIDI CC initialization is ignored. The pedals are all OFF at instrument loading. The new values are sensed as soon as the linked MIDI CC is changed. This can include HOST initializations.

<mcc1> on both commands let you to specify a MIDI CC to use to command the pedal.

<mcc2> on both commands let you to specify another MIDI CC to use to command the pedal and applied only to keys between <minkey> and <maxkey>.

<mcc1> or <mcc2> can be:

- 0 - 127: normal MIDI CCs. The pedal is considered ON if the MIDI CC value is above 63, otherwise OFF.
  MIDI CC that are not in the range 0-127 (like pitch bend) or special MIDI CC (KEY, FREQ, GAIN, IN, OUT etc.) cannot be used.
- 400 - 527: Keyswitch values of keys 0 - 127. The pedal is considered ON if the value is above 0, otherwise OFF. The keyswitch value is global, so a keyswitch can activate a pedal for all channels. It monitors the NOTE ON message from a single channel, though.
- 600 - 727: VST VAR 0 - 127 value. The pedal is considered ON if the value is above or equal 0.5, otherwise OFF. The VST VAR value is global, so it can activate all channels at the same time. To have single VST VARs for each channel (to be able to use DAW automation), just define multiple VST VARS and link each to a different MIDI CC number and channel. E.g. CC 64 (HOLD) and channel 0, 1, 2…,15 linked to 16 consecutive VST

VARs, then use PEDAL 64,…. Alternatively if you don't need MIDI CC automation you can use User defined MIDI CC (160-199) or another unused MIDI CC instead of standard MIDI CCs.

- All other values: the pedal is considered always OFF.

Default values: -1 (so always OFF).

PEDAL is the classic hold pedal: as long as the pedal is ON, the corresponding keys (all for <mcc1>, only between <minkey> and <makxey> for <mcc2>) will use the decay time for the release time.
SOSTENUTO is the classic sostenuto pedal: as long as the pedal is ON, the keys that were ON and not in release at the moment the pedal has gone from OFF to ON (among all for <mcc1>, or only those between <minkey> and <makxey> for <mcc2>) will use the decay time for the release time.
The four pedals are "additive" in some way: if any of the four conditions is active for a given note, that note will use the decay time as release time.
The fact that the pedals acts on the release time allows to implement other behaviors, e.g. if the sustain is 1, it allows to select between two release rates and so can be thought as a dampen pedal.

Multichannel considerations:
If using per channel MIDI CC (0-127), each layer take the pedal setting from the corresponding channel. The only limitation is that the MIDI CC numbers and keys are shared by all the channels. To have different MIDI CC or key setting for different instruments you should use different VST instances.

PROGRAMNAME <Program_number>; <Bank_number>; "<Program_name>"

This instruction sets a descriptive name for a MIDI program in a given MIDI bank.
<Program_number> and <Bank_number> select the target program. 0 – 127 and 0 – 16383 are the range for them, respectively. Error is given if the parameters are out of range.
<Program_name> sets the actual name. Max 19 characters or an error will be given.
The current version supports maximum 8192 program name entries.

MIDI Instructions, stage II: Post-processing

The MIDI processing pipeline acts on all Note ON and OFF messages, even those generated by the sequencer.
It is comprised of five consecutive steps.
The processing order is fixed, but the instructions below are declarative, so they can be specified in any order in the file.
They are described here in the order in which the actual processing is performed, to help understand better the processing flow.
The first three parameters of each instruction are **<cc>, <min key> and <max key>**.
<cc> is the MIDI CC used to control the effect and the minimum and maximum key to which the step applies.

- The key used for the check is the original key pressed. The same applies to the velocity in the velocity step.
- If you want to disable a step, e.g. to be sure a previous declaration has not effect, just set the keys to -1 and -1.
- If some keys are enabled (e.g. all the keyboard or all the lower keys), then a MIDI CC can be used to further control the effect enablement.

- A value not listed here (e.g. -1 and all negative values), always enables the effect on the keys selected (e.g. -1,0,127 always enables the effects for all keys).
    - 0-127 standard MIDI CC. The effect is enabled if the MIDI CC value is greater than 63.
    - 400-527 Keyswitch value on key 0-127. If the Keyswitch value is below 1, the effect is disabled. Otherwise is enabled.
      It is advised to create an one key keyswitch with minimum value 0 and maximum value 1. This will act as an ON/OFF switch.
      e.g.: 400 means that the keyswitch at key #0 (C-2) will select the enablement of the effect.
      If at the key selected there is not a keyswitch, the effect will be always disabled.
    - 600-727 VST VAR value of variable number 0-127. If the variable value is below 0.5, the effect is disabled. Otherwise is enabled.
      It is advised to set the VST VAR with a range [0; 1], integer scale.
      e.g.: 600 means that the VST VAR number 0 (the first) will select the enablement of the effect.

NOTE: the MIDI CC numbers used for the automation are the same for all channels, but each LAYER use the corresponding default channel value for the actual activation (for MIDI CC 0-127. The other MIDI CCs are global).

NOTE: the current temperament detunes are applied at the end of the 5-th step on all notes generated (see below).
For fractional pitches, the note is rounded for the purpose of semitone picking but if the rounding is disabled, the note remains unrounded.

HOST temperament works only if ARPEGGIO and CHORD are disabled: this is because the HOST detune applies only to the root note, otherwise the detune cannot be applied: in this case EQUAL temperament is used. Other temperaments will work though.

NOTE: the KEY, KEYF, FREQ, GAIN, ONVEL, OFFVEL keywords and MIDI CCs refer to the modified notes created by the MIDI pipeline.

The first step can be one of ARPEGGIO or CHORD, but not both.
Setting an ARPEGGIO, disables a previous CHORD and vice versa.
Disabling an ARPEGGIO with ARPEGGIO -1;-1;-1 disables also an eventual previous CHORD.
Disabling a CHORD with CHORD -1;-1;-1 disables also an eventual previous ARPEGGIO.

NOTE:

If the sequencer is enabled (see above), then both ARPEGGIO and CHORD are disabled.
An eventual ARPEGGIO instruction is used to initialize the sequencer’s TAPE (see above).

ARPEGGIO <cc>; <min key>; <max key>; <size>; <semi1>; <delay1>; <duration1>; <velmul1>....; <semiN>; <delayN>; <durationN>; <velmulN>

This function is used to program ARPEGGIOs.
Defaults are -1;-1;-1
Maximum 1024 notes can be specified.
Minimum 2 must be specified if it is enabled.

The syntax ARPEGGIO -1;-1;-1 is allowed, to disable an arpeggio or chord in e.g. an early included file.
The last note, if incomplete, takes the default values of 0,.25,1 for delay, duration and velmul.

A key hit triggers the ARPEGGIO.
Each key can trigger a different simultaneous ARPEGGIO, with different base note.
Check for polyphony: the loop is retriggered from the last note and the notes are created all the same time.

If the polyphony is too low, then there is risk of losing notes or of not triggering the loop again if the last note is deleted too early.

When the key of a running ARPEGGIO is depressed, it is canceled for that key: the active notes are put in release and fades away and following notes are not issued.

NOTE: the releases observe an eventual further delay or random duration spread set in the TIMING step.

NOTE: tempo changes are observed at each loop retrigger, because a whole loop at a time is created. This means that for very fast varying tempo, the bar durations can be inaccurate.

<size> is the size of the loop. Minimum 1/8 bar (0.125). This number is in bars: 0.25 is a quarter bar, 0.5 is half bar and so on.

- When the last note is triggered, if the key is still pressed, all the notes are regenerated with the correct timing and parameters.
  For this reason the last note must start before the loop should start again, otherwise UNPREDICTABLE things can happen, e.g. the new notes will be created already in release stage or worse already off.
- The actual starting time of the last note can be further away due to TIMING step. This does not matter: the original starting time is used to regenerate the notes.
- <size> cannot be zero, but can be negative: in this case the actual size used is the absolute value, but an optional feature of the arpeggio is enabled: complete loop.
  With this option, the arpeggio completes anyway a whole loop when the original key is released.

Then comes the ARPEGGIO notes. To be able to support also arpeggios not starting on the key note, even the first note must be specified.

<semiI> is the shift in semitones from the note actually played, of the I-th note in the ARPEGGIO. Could be fractional to simulate fretless or continuous instruments.

<delayI> is the delay in bars from the first note of the I-th note. Can be used even for first note for delayed arpeggio.

<durationI> is the duration of the I-th note. It is the number of bars of duration. Must be positive.

<velmulI> is the multiply factor to be applied to the first note velocity, to calculate the I-th note velocity. Ignored for the first note.

Note: the delays should be in ascending order. No check is made. Unpredictable results if not fulfilled. At least the last note must be the one with the maximum delay.

EXAMPLE:

```
// Simple major arpeggio, always on (cc=-1):
//      cc   k1  k2  siz   0 semi       4 semi          7 semi         12 semi          7 semi        4 semi
then repeat from start...
ARPEGGIO -1,  0, 127, 1.5,  0,0,.125,1,  4,.25,.125,1,   7,.5,.125,1,   12,.75,.125,1,   7,1,.125,1,
4,1.25,.125,1
```

Note: if the other algorithms specify some randomness for some parameter, each arpeggio iteration the random factors are generated again, so each repetition would be different.
Note: the other following algorithms apply to all the notes produced by this step or the single original note if the key is out of range or the step is disabled.
Note: pedals and sostenuto modify only the release time of the notes: only the note off are used to stop an arpeggio.

CHORD <cc>;<min key>;<max key>;<semi1>;<velmul1>....;<semiN>;<velmulN>

This function is used to program CHORDs.
Defaults are -1;-1;-1
Maximum 32 notes can be specified. Minimum 2 must be specified if it is enabled. The syntax CHORD -1;-1;-1 is allowed, to disable a chord or an arpeggio in e.g. an early included file.
The last note, if incomplete, takes the default value of 1 for the velmul.

A key hit triggers the CHORD. Each key can trigger a different simultaneous CHORD, with different base note.
When the key of a running CHORD is depressed, all the active notes are put in release and fade away.
NOTE: the releases observe an eventual further delay or random duration spread set in the TIMING step.

To be able to support also chords not starting on the key note, even the first note must be specified.

<semiI> is the shift in semitones from the note actually played, of the I-th note in the CHORD. Could be fractional to simulate fretless or continuous instruments.
<velmulI> is the multiply factor to be applied to the first note velocity, to calculate the I-th note velocity. Ignored for the first note.

```
// Simple major chord on the lower portion of the keyboard:
//    cc   k1 k2  0 semi   4 semi    7 semi
CHORD -1,  0, 47, 0,1,     4,1,      7,1
```

Note: the other following algorithms apply to all the notes produced by this step or the single original note if the key is out of range or the step is disabled.
Note: pedals and sostenuto modify only the release time of the notes: only the note off are used to stop the chord notes.

MAP <cc>;<min key>;<max key>;<actual semitone for C>;....;<actual semitone for B>

This function maps semitones onto other ones.
It can be used to force a scale for played notes or chord and arpeggios to be sure to be in the right scale.

Default values are -1;-1;-1;0...;11.

A note is mapped in the same octave, but with the semitone corresponding to the i-th position.
Note that if two note mapped to the same semitone are played together, they are triggered 2 times.

Values are clipped to [0, 11].
e.g. MAP -1;0;127;0;1;2...;11 is the identity mapping. You can swap semitones, map multiple semitones with the same one in output, etc...

EXAMPLE:
Suppose you have the arpeggio of the example above:

```
// Simple major arpeggio:
//      cc   k1  k2  siz   0 semi       4 semi          7 semi         12 semi          7 semi        4 semi
then repeat from start...
ARPEGGIO -1,  0, 127, 1.5,  0,0,.125,1,  4,.25,.125,1,   7,.5,.125,1,   12,.75,.125,1,   7,1,.125,1,
4,1.25,.125,1
```

If you play the C note, it correctly performs the arpeggio in C Major.
If you play the A note, it plays an A Major arpeggio.
But in a C Major scale, you may want it to perform an A Minor arpeggio, to be in scale.
This can be solved with this mapping:

```
// C Major scale mapping
MAP -1,0,127,0,0,2,2,4,5,5,7,7,9,9,11
```

It forces only notes in the C Major scale and thus the A Minor and all the other corrects mappings for the other chords/arpeggios.
Any custom scale e.g. blues, Dorian, can be used.
The same applies for the CHORDS.

VELOCITY <cc>;<min key>;<max key>;<min vel in>;<max vel in>;<min vel out>;<max vel out>;<drive>;<spread>;<velocity flag>;<mul0>;<mul63>;<mul127>

Velocity modifying step.
Defaults are -1;-1;-1;0;127;0;127;1;0;0;1;1;1

<min vel in>;<max vel in>: notes outside this range of note on velocities are not modified by this filter.
<min vel out>;<max vel out>: specify the rough velocity output range, before the spread adding.
<drive> is a parameter of the velocity formula (see below) and specifies the distortion (exponent) of the mapping. 1 = linear.
<spread> is the range of the random spread added to the final velocity, ranging from -spread to spread.
<mul0>, <mul63>, <mul127> are velocity multipliers for note key=0, 63 and 127 to modify the velocity according to note key, to e.g. have a pre filtering effect.

The final formula is:

Let vin=velocity.

vin is first multiplied by a factor depending from note key, linearly interpolated between 0 and 63 or 63 and 127. Then vin is constrained again between min vel in and max vel in.

Then vout is calculated:

vout = (((vin - min vel in) / (max vel in - min vel in)) ^ drive) * (max vel out - min vel out) + min vel out + spread

vout is clipped to [0, 127].

vout is the final value.

<velocity flag> specifies to what velocity apply the mapping: 0 = Note ON only, 1 = Note OFF only, 2 = Both.

PITCH <cc>;<min key>;<max key>;<shift>;<spread>;<chance>;<rounding>

Pitch modifying step.
Defaults are -1;-1;-1;0;0;0;0

<shift> is the fixed shift in semitones to apply to the notes.
<spread> is the range of the random spread added to the final pitch, ranging from -spread to spread.
<chance> is the chance of applying the spread (1=certainty, 0=never). NOTE: the shift is always applied.
<rounding> is the pitch rounding in semitones: 0 = no rounding. >0 = rounding to n semitones: used to have casualty with fixed steps.

TIMING <cc>;<min key>;<max key>;<delay>;<on spread>;<duration spread>;<quantization>

Timing modifying step. This step can be used both to humanize (randomize) the timings or quantize them.
Defaults are -1;-1;-1;0;0;0;0

<delay> is the delay, in bars, to add to all note on messages.
<on spread> is the random spread, in bars, to add to all note on messages.
<duration spread> is the random spread, in bars, to add to all notes duration.
<quantization> is the quantization in bars of the note on and off timings.

The spreads are always positive.

MIDI Instructions, stage III: Temperaments

TEMPERAMENTCC <number>, [<channel>]

Links the temperament drop box with an extended MIDI CC (including KEYSWITCHes).

<number> is the (extended) MIDI cc number that is linked with the current temperament number.

- If it's <0 (e.g. -1) or in a range not covered by the options below, the temperaments are selectable only in the GUI.
- For values including 0-127, 133, 134, 135, 155, 160-199 and 200 – 327, the selected MIDI CC and the selected channel is used to select the temperaments. Since the actual values (0-

127) are used, a max of 128 temperaments could be selected. Thus some MIDI CC are more suitable (e.g. Bank select) than other (Pedals, etc.).

- If it's between 400 and 527, a key between 0 and 127 is reserved for temperament switch. It can't be used to trigger a note. If it's assigned to a keyswitch it will be used for the temperament so make sure to not use it in a KEYSWITCH declaration (see above).
  The selected key acts as an <one key> keyswitch, meaning that it cycles between all temperaments.
  This choice to support only <one key> style keyswitch for temperaments is justified by the fact that temperament selection is almost performed before start playing the keyboard and so can be effectively performed in the GUI or with a single simple keyswitch.
  Moreover the VST plugin will remember the last temperament if the user saves the project file including the instrument in the DAW.
- If it's between 600 and 727 a VST variable (automatable knob in the GUI) is used to select temperaments.
  The VST VAR is automatically set to linear scale, with a range that covers exactly the temperaments defined, so it’s advisable to put TEMPERAMENTCC after ALL TEMPERAMENT declarations.
  Moreover the VST VAR label is dynamically set to the current temperament name, to be able to know the current temperament even if the VST interface (editor in VST parlance) is closed.

<channel> is the channel number to use for the non global MIDI CCs. If not specified, defaults to 0.

NOTE (1): the temperament is global to all channels. To use different temperaments for each instrument you should use multiple VST instances.
NOTE (2): the temperament can be selected also in the GUI (e.g. with the mouse), and the selection DOES update the VST VAR or MIDI CC eventually linked IN REALTIME. The vice versa is also true. No MIDI message is sent.
NOTE (3): the new temperament applies only on the notes played AFTER the modify. Notes being currently played keep the original temperament they were assigned at trigger time.
NOTE (4): the temperament DOES NOT need to be linked to a MIDI CC or VST var to be available in the instrument file, e.g. for triggers or IF ... GOTO statements. There is a keyword and an extended MIDI CC for this purpose.

TEMPERAMENT "Name";<cents C>;<cents C#>;...;<cents B>;<optional_middke_key>
OR
TEMPERAMENT <number>
OR
TEMPERAMENT "Name"
OR
TEMPERAMENT "SCALA or TUN file or folder path"
OR
TEMPERAMENT "SCALA file or folder path";"KBM file path"/<middle_key>

This instruction is for adding new temperaments to the list of user selectable temperaments.

By default HOST and EQUAL temperament are always defined, with code 0 and 1. Other declared temperaments are given the following codes.

HOST temperament is detune values passed to the VST.

The VST specifications states that the HOST transmits a detune value in cents, an integer between -64 and +63, along each note on message.

If the HOST DAW does not support temperaments, the values passed to the VST are all zeros, thus if the host does not support temperaments, HOST and EQUAL temperaments are equivalent.

Only temperament number #0 (HOST) uses the host values (if supported).

There is in the GUI a drop box used to select the current temperament (e.g. with the mouse).
A maximum of 8192 temperaments can be defined.

The first syntax defines a new temperament.

"Name" is the name of the new temperament, e.g. "Just", shown as option in the temperament drop box in the GUI.

<cents C>... are floating point numbers specifying how much cents detune from equal temperament each semitone. Can have also fractional part for more precision, e.g. 3.14156 cents.
The instruction automatically detects almost increasingly patterns, with a tolerance of +- 100 cents from the 0, 100…,1100 pattern. In this case it assumes detuning in cents from the root note.

<middle_key> is optional and specifies the start of the scale. Default 60 (C3), range 0-127.
Specifying a value different from 60 means that the first detune is applied to the key number <middle_key> and so on.
With the standard temperaments only 12 choices are different: e.g. 72, 48, 84, 36 etc. are the same than 60.

The second and third syntax set the current temperament as if one clicked on the temperament drop box.

<number> is the number of the temperament. If it's out of range, the nearest temperament is chosen (clipping).
If unsure of the temperament number, just declare the wanted temperament as the last one and set a high number that will be clipped.

"Name" is the name of the temperament. The search is case insensitive. If it is not found, a warning is issued in the log window and the temperament is not changed.

If the temperament drop box is associated to a MIDI CC or VST VAR (see above), the change with this instruction (or also in the GUI with the mouse) is reflected in the linked object.

The fourth and fifth syntaxes allow to define a new temperament based on a SCALA or TUN file or to scan a folder to import in BULK all SCALA or TUN files inside it.

If the first parameter appears to be a valid path to a SCALA or TUN file, this file will be imported and a temperament with the name of the file will be created. The path can be absolute, relative to the instrument path or relative to the Crescendo directory and can contain further path specs, e.g. SCALES\foo.scl. The file type is inferred by the file extension (case insensitive): SCL for SCALE and TUN for AnaMark tuning file (version 0 and 1).
Maximum 128 note are supported: files with more than that number of notes are rejected.

With the fourth syntax a file without keyboard remapping is loaded.

For the SCL file, the middle key is assumed to be 60 (middle C) and the pattern repeated up and down. When the temperament is selected, the current, per layer, values of BASEFREQ and KEYCENTER are used to force the frequency of the given key (KEYCENTER, rounded to the nearest integer: in these cases the fine tuning will not work) and the KEYTRACK is ignored.

For the TUN file, there is not middle key, because the TUN file contains all or most key tunings (giving the equal temperament with 440 Hz at the A3 for the unspecified tunings). If the basefreq in the [Exact Tuning] section is not specified, the same behavior of the SCL file is employed for the BASEFREQ, KEYCENTER and KEYTRACK. If the basefreq is specified, the BASEFREQ and KEYCENTER values are ignored and the centering is performed with the new basefreq (with center on the note 0). Periodic TUN files are supported as per specs.

To discern between a file name and a temperament name, if a file or path is not found in the three locations, the string is assumed to be a temperament name and searched in the list.
If an existing file name is given, an error will be raised if the file is not a SCL or TUN file or a parsing error occurs.

With the fifth syntax a valid SCL file path and a valid KBM file path (including the extensions, case insensitive) are needed. Same search strategy is employed.
The SCL file is processed as above and the keyboard mapping specifies the middle note, the base frequency, the key center, the formal octave and a key range to be reordered. When the temperament is selected, the KEYCENTER and BASEFREQ are changed and fixed for all the keys. Just the reordering is performed eventually on a subset. If the KBM files specify a mapping for more scales than the SCL file, the whole couple SCL/KBM is rejected.

Alternatively, if the second parameter is a number and the file is a scala file, the middle key is set to the specified value. KEYCENTER, BASEFREQ and KEYTRACK are treated as the fourth syntax.
If the middle key is different from 60 the value is appended to the temperament name.
If the file is a TUN file, the middle key value is ignored and a warning is issued on the log (if enabled).

If the first parameter is a valid folder, this folder is scanned for valid SCL or TUN files. Non SCL or TUN files or invalid files are silently skipped.
If a valid KBM file is given as second parameter, this will be applied to all SCL files and ignored for TUN files.
If a middle key value is given, it is applied to the scala files, and ignored for the TUN files.
For each file in the folder the same checks are performed and the temperament is rejected if any of them fails: SCALA and TUN files must specify at most 128 scales, if a KBM file is specified, all SCL files with less scales as the KBM file will be skipped etc.…

<middle_key> is rounded to the nearest integer and capped to [0, 127].

MIDI Instructions, stage IV: Triggers, Group and Crossfades

Here we explain along ON triggers, OFF triggers, GROUP and XFADE instructions.

General notes about the triggers:

Although the MIDI CC numbers are shared among all LAYERS and channels, each trigger or MIDI

CC value in the XFADE instruction is taken form the channel corresponding to the channel of the original NOTE ON message evaluated.
GROUP OFF triggers are triggered from notes of the specified group and the same MIDI channel.
Analogously ONLEGATO, ONFIRST and ONRROBIN checks for notes on the same MIDI channel.

Each declaration below can be superseded by one following it in the file.

There exists a form for deleting a previous declared trigger (in another file, in the common section). Just set <min> > <max>.

There exists a form to delete all ON triggers or all OFF triggers. Just set <mcc> < 0 in ONMCCT or OFFMCCT.

OFF triggers are checked also at note ON: if a layer has an OFF trigger valid at note ON, the layer will not be triggered.
ON triggers are in AND. OFF triggers are in OR.

The NOTE OFF MIDI command ALWAYS turns off the corresponding layer(s). NOTE OFF trigger cannot be disabled.

The NOTE ON MIDI command can be filtered by key and velocity AND other ON triggers in AND.

A maximum number of 32 ON triggers plus 32 OFF triggers can be specified per layer.

GROUP specifies to what group the LAYER belongs, because other LAYERs can be instructed to turn off when another LAYER with a specific group triggers. This can be useful for exclusive mode (a group of cymbals, notes payable on a guitar string, etc.) and other effects.

Crossfades are useful if multiple layers should be triggered and the mix should depend from some parameters.

ON Triggers

ONNOTEON <minkey>,<maxkey>,<minvel>,<maxvel>

Default 0,127,0,127.
This is the only trigger declaration that can go only in layer section.
It allows modifying the default key and velocity filter on a layer.

ONMCCT <number>,<min>,<max>

Default -1,-1,-1.
This allows setting, modifying or deleting an ON trigger on a MIDI CC:
the trigger is valid if at trigger time the MIDI CC <mcc> is between <min> and <max>, included.
<number> is the MIDI CC number (see MCC for the codes)
<min> and <max> are the minimum and maximum value of the interval in which the trigger is valid.
They are floating point values.

ONFIRST <min>,<max>

This allows setting a trigger that is valid if there are not any active notes with key between <min> and <max> included, on the same MIDI channel.
Notes in release phase are not counted. This is usually paired with another layer with the ONLEGATO trigger.

ONLEGATO <min>,<max>

This allows to set a trigger that is valid if there is at least one active note with key between <min> and <max> included, on the same MIDI channel.
Notes in release phase are not counted. This is usually paired with another layer with the ONFIRST trigger.

ONRROBIN <remainder>,<divisor>

This allows defining a round robin trigger.
If the counted note on events, divided by <divisor> has remainder equal to <remainder>, then the trigger is valid. This check is performed with per cannel counters.
If all layers have the same <divisor>, then it is a classic round robin of size <divisor>
Note that the counter does not advance if there are holes in the sequence:

- take care to define at least one layer that potentially can trigger: if other triggers don't trigger, the counter advances anyway.
- This means that it is not applied the e.g. C/C++ short circuit rule for AND clauses: if an ON trigger is false, the others are always evaluated and so the counter can be increased.

ONCHANNEL <min>,<max>

This allows triggering a note only if the MIDI channel is in the specified range.
Note that MIDICH must include at least one of those channel, otherwise the layer will never be triggered.

OFF Triggers

OFFGROUP <number>,<forcedrelease>

This trigger specifies an OFF condition on another layer trigger.

If a layer of group <number> is triggered, the layers that have this trigger and is on the same MIDI channel of the initial layer, will immediately go in release with the specified forced release.

If forced release is <=0, then the release of the layer is the normal programmed release, otherwise all the envelopes are forced to this release time.

If the note just triggered has the OFFGROUP trigger with the same group it belongs, then this note will not be silenced: this is exclusive mode; only one note of a group can be active.

If there is an arpeggio or chord, other notes that trigger other layers with the same OFFGROUP clause, will be canceled: only the main note (and other without OFFGROUP clause) will trigger.

OFFMCCT <number>,<min>,<max>,<forcedrelease>

Default -1,-1,-1.
This allows setting, modifying or deleting an OFF trigger on a MIDI CC:
    the trigger is valid anytime the MIDI CC <mcc> is between <min> and <max>, included.
<number> is the MIDI CC number (see MCC for the codes)
<min> and <max> are the minimum and maximum value of the interval in which the trigger is valid.
    They are floating point values.

If the MIDI CC <mcc> is between <min> and <max>, the layer that has this trigger will immediately go in release with the specified forced release.

If forced release is <=0, then the release of the layer is the normal programmed release, otherwise all the envelopes are forced to this release time.

GROUP and Crossfades

GROUP <number>

This declaration can be in the COMMON section or in the LAYER section, but NOT in POST section (because pertains to LAYERs).

This declaration set the group of layers, useful in triggers (see triggers above). Range: -1 to 100000. Start value -1.
-1 means that a layer does not belong to any group.
If given in the COMMON section, it sets the default group of all layers.
Since last declaration is what counts, this declaration can be overwritten in any point in the common section (even with the value -1).

In a layer can be further overwritten, only for that layer (even with the value -1).

In the POST section it is forbidden and an error will be given.

XFADE <type>;<mcc>;<min>;<max>;<power>

This declaration can be in the COMMON section or in the LAYER section, but NOT in POST section (because pertains to LAYERs).

This declaration adds, modify or delete IN and OUT crossfades to a LAYER or the COMMON pool that will be inherited by all the following layers.

Crossfades work as follows:

Let v be the value of MIDI CC number <mcc> (with the <mcc> values like the MCC instruction described below)

An IN crossfade evaluates 0 if v < min and 1 if v > max. For intermediate values a linear interpolation is performed.

An OUT crossfade evaluates 1 if v < min and 0 if v > max. For intermediate values a linear interpolation is performed.

All the active crossfades will be multiplied together and if the product is not 0 the layer will be triggered, with the signal value multiplied by this value.

<type> is the type of the crossfade: 0 is IN and 1 is OUT.

- A special value is -1 that allows resetting and deleting all crossfades.
- In common section it deletes al previous XFADE declarations, e.g. in previous included files.
- In a layer it deletes only for that layer all the previous XFADE declarations to start all over.

<mcc> is the MIDI CC number to monitor.

<min> and <max> are the MIDI CC limit values.

- If <min> >= <max> then the corresponding crossfade will be deleted from the list, if it exists.
  This can be used to delete only a specific crossfade.

<power> is a flag for crossfade shape:

- 0 is linear: the crossfade value sweeps linearly from 0 to 1.
- 1 is power: the crossfade value sweeps linearly from 0 to 1 and then the square root is used.
- 2 is exponential: if 0, it remains 0. Otherwise sweeps linearly from -60 dB to 0 dB.

If an IN or OUT crossfade for a given MIDI CC already exist, it will be overwritten, for the current layer only or all layers if the declaration is in the common section.

All the XFADE declarations in the common section will be inherited by all the layers.
They can be deleted one by one or together or updated in each layer separately.

Sequencer Instructions

The sequencer is an optional programmable module, with a simple programming language defined here.

Its purpose is to generate a sequence of notes, starting from an initial set, possibly augmented by live user generated notes.

The sequencer is able to transform in a programmable way a sequence of notes and issue them on the next stage of processing.

To activate the sequencer, you must choose a VST VAR that will control its status.

To be sure to have the sequencer deactivated, you must issue a SEQUENCER instruction with an invalid MIDI CC number, e.g. SEQUENCER -1,””.

By default it is deactivated.

The sequencer uses as its main structure an abstraction, called the TAPE.

The TAPE contains an array of notes, sorted by trigger time and two positions: the <cursor> and the <end>.

The <cursor> is the index of the note that will be played next, as soon as the trigger time allows it.
Multiple notes can be issued at a time if they have the same trigger time.
Each note can trigger multiple layers, just like normal NOTE ON messages.
The cursor can range from 0 to <end>.
As soon as it reaches <end>, there are no more notes to be issued.
In this case the "Program" will be issued, that hopefully will generate new notes to play next.

The <end> is the index of where the next produced note by the "Program" would go.
It signals the end of the TAPE.

The "Program" will generate new notes, but that new notes will be put on a BAR boundary, since the start of the processing: the last note played trigger time is rounded in excess to the next bar and used as the starting time of the new notes.

Due to possible randomization, the "Program" will be executed up to 10 times or 2000 instructions (there can be loops) until the <cursor> is less than <end> (a program can generate new notes or move the <cursor>). The program is fully executed at least once.

At sequencer activation, the TAPE is empty and will be filled with the notes of an eventual ARPEGGIO instruction. <cursor> will be set to 0 and <end> to the number of notes contained in the arpeggio (possibly zero).

The MIDI instructions are declarative, so the TAPE filling is performed at runtime.
This means that the SEQUENCER and ARPEGGIO instructions can be in any order in the file.
This also means that older specifications will be overwritten.
The ARPEGGIO and CHORD instructions will not have any other effects if the sequencer is activated.

Then the processing begins.
The sequencer has five states, selectable by the associated VST VAR:

- 0 - NORMAL/RESET.
  In this state the sequencer is paused.
  Input NOTE messages are passed to the next stages undisturbed. No arpeggio or chord is applied.
  The TAPE state is reset to the initial value (<cursor> = 0 and TAPE filled with the arpeggio notes) as soon as the Knob is moved forward.
  This means that an eventual TAPE full of notes (including merged ones) is still present until the knob is moved and can be saved in a TAPE file (see below), before being lost.
  NOTE: if the DAW issues a suspend() or transitions the Running state from running to not running (typically when the user clicks on a STOP button on its interface), the sequencer is forced to reset and so you will lose the old TAPE.
- 1 - PAUSE.
  In this state the sequencer is paused.
  Input NOTE messages are passed to the next stages undisturbed. No arpeggio or chord is applied.
  The TAPE state is not modified.

- 2 - SEQ. ONLY.
  In this state the sequencer is running: NOTE messages are generated from the TAPE.
  As soon as the <cursor> reaches <end>, the “Program” is executed.
  Input NOTE messages are discarded for ALL channels. To avoid this you must route the channel you don’t want to lose to another VST instance.
- 3 - LIVE.
  In this state the sequencer is running: NOTE messages are generated from the TAPE.
  As soon as the <cursor> reaches <end>, the “Program” is executed.
  Input NOTE messages are played along with the TAPE notes. No arpeggio or chord is applied.
- 4 - MERGE.
  In this state the sequencer is running: NOTE messages are generated from the TAPE.
  As soon as the <cursor> reaches <end>, the “Program” is executed.
  Input NOTE messages are played along with the TAPE notes. No arpeggio or chord is applied.
  Moreover the Input NOTEs are saved in the TAPE at NOTE OFF, along with the already issued notes, in the correct position.
  This means that the next time the program is issued, it will process also the user inputted notes
  NOTE: all the enabled MIDI channels are merged together and the channel information is deleted when integrating it in the TAPE. The original notes will play with the correct instrument though. To merge only a single instrument, use the SEQCHANNEL instruction.
  NOTE: when the program is executed, if it uses somehow the last n bars and if a merged note is longer than n bars, the trigger time of this note will be too old and it will not be used.

At file loading the state is always initialized to NORMAL, regardless eventual VST VAR initialization except when loading a saved preset or a full file/project in a DAW: in this case the DAW’s cached value takes precedence and the notes can start to play immediately, but only if the DAW does not issue a suspend or a running => not running transition.

TAPE load/saving

The current content of the TAPE can be loaded or saved in a file.
There is an internal status variable, called <tape_file> that specifies the last tape file loaded or saved.
After the instrument file loading, the <tape_file> variable is initialized to an empty value.
The current value of <tape_file> can be seen hoovering the mouse on the sequencer knob. The file name will appear in a tooltip.

By giving focus to the sequencer knob, you can trigger the tape file load procedure with the “L” key or a right click on the knob.
You can trigger the tape file save procedure with the “S” key or a shift and/or ctrl + right click on the knob.

Loading a tape file, makes it the current <tape_file>, loads the tape file in the arpeggio and resets the tape with this data. If the sequencer was running, the notes are played immediately, starting from the first (since the cursor is also reset).
Saving the TAPE on a tape file, makes it the current <tape_file>, temporarily freezes the sequencer and saves all the TAPE slots as they are in that instant (so including any merged notes, previous processing, etc.) in the tape file.
Finally the file is immediately reloaded, so the sequencer is reset as above, so if you are in MERGE mode you can add further notes and save again later, making a layered song.

When resetting the sequencer with the knob, the last tape file loaded or saved, that is stored into the internal arpeggio storage, is put in the tape and the cursor is reset to the start of the TAPE.
The only way to return to the original arpeggio in the instrument file is to reload it, resetting also <tape_file> to empty.
If the ARPEGGIO was empty in the instrument file, the TAPE can contain only merged notes, eventually processed with the program multiple times.
If the program was empty too, then into the TAPE there are only the notes played while the sequencer was in MERGE mode, so saving the TAPE is like having a MIDI recorder.
This TAPE file can be played unmodified in an instrument file with a sequencer with empty program. Just load the file into the TAPE.

The tape file is a csv text file with “tap” extension, with one row for each TAPE slot, so it can be edited, with caution.
The timing data of the TAPE, when saved in the file, are rescaled as if the TAPE is playing with 120 BPM and 4/4.
When the file is read back, the timing data are rescaled to the current BPM and NUM/DEN.

The row format is of 5 comma separated floating point numbers: trigger time, in seconds, from the start (0.0 the first note), release time from the start, note in semitones (69.0 = A3), velocity and off velocity (0.0 – 127.0, like MIDI).

When loading or saving your DAW project or an fx or fxp file, the current <tape_file> full path is stored/retrieved in the DAW file, so the last tape file loaded or saved is remembered in the DAW file: when the DAW file is reloaded, this file will be reloaded and available in the sequencer.

Here follows the description of the sequencer related instructions.
The sequencer language is explained in the SEQUENCER instruction definition.

SEQBASE <bnote>,<bvel>,<boffvel>

The ARPEGGIO instruction specifies relative pitches and velocities.
This instruction sets the absolute base value to use when filling the TAPE.

<bnote> is the base pitch. Can be fractional. Default 60 (C3).
<bvel> is the base ON velocity. Can be fractional. Default 100.
<boffvel> is the base OFF velocity. Can be fractional. Default 100.

SEQCHANNEL <Sequencerchannel>, <MERGEminch>, <MERGEmaxch>

The sequencer notes must be associated with a MIDI channel to be able to correctly process the triggers on the LAYERs.

Moreover you may want to filter the NOTE ON messages when merging them into the TAPE, e.g. to save only the messages coming from specific sources. The chosen channels must be enabled in the MIDICH instruction.
Note that the filter applies only for the sequencer: if there are LAYERs that trigger for the excluded channels, they will normally trigger.
This instruction sets the channels for the sequencer.

<Sequencerchannel> specifies on what MIDI channel the notes will be put, even on the MIDI output. It specifies also what MIDI channel to use for the IF instruction (See below). Default 0.

<MERGEminch> and <MERGEmaxch> specify the MIDI channel range of the accepted merged notes. Default 0-15.

SEED <offset>

The random numbers generator is “seeded” in such a way to have repeatable sequencer behavior, e.g. to be able to play or record the exact note sequence you played earlier.
But this means that the random numbers are not truly random.
If you want to experiment new instances of the sequencer, without changing the program, just change the seed offset.
The same value will give the same sequence. There are about 4 billion of combinations.
<offset> is the unsigned 32 bit integer that will be added to the internal seed generator. Default = 0.

SEQUENCER <mcc>,"Format string"

Here it is the most important function for the sequencer module.

<mcc> specifies the MIDI CC to use for the sequencer control. Only VST VARs are allowed and so this value must be between 600 and 727. Other values causes the sequencer to be completely disabled (the ARPEGGIO and CHORD instructions will work normally). Default -1, hence sequencer disabled.

“Format string” is the string specification of the “Program”.

Some notes:

- Instructions are separated by “;”
- Parameters are separated by “,”
- Token name is case insensitive, but spaces are not deleted, so after the “;” the next token must be exactly attached to it.
- The line continuation character “/” works also in opened strings, so you can put the program in contiguous rows, provided that the above spaces rule is followed (e.g. if you are inside the string, the spaces are not deleted, so the intermediate “rows” must start at column 0).
- Most parameters are optional. The parser will recognize also “<TOKEN><SPACE>;”, meaning an instruction without parameters, that maintain the default values.
- Some instructions have no parameters: those must be specified without the space before the “;”.
- The parsing of the parameters is more relaxed: provided that the numbers are not broken, multiple spaces are allowed before and after the “,”.
  C’s sscanf is used for the parameters input, so if a number is incorrectly written, it can be truncated or the default value for the parameter will be kept.

Concepts used in the descriptions that follow:

- Let <cursor> be the pointer to the last note played (the format string is executed when the TAPE is empty after this note).
- Let <rcursor> be the trigger time of the note at the <cursor>, rounded to the next integer bar since the start of the sequencer.
- Let <end> be the pointer to the last note processed. At processing start <end> = <cursor> but when the processing proceeds, <end> is generally greater than <cursor>.

  - Let <last> be the pointer to the last note of the initial arpeggio used to initialize the sequencer (the sequencer works even with empty arpeggio, but until you merge some notes nothing happens).
  - The note positions and durations are expressed in BAR or fractions of BARs. Some values are rounded to the next integer BAR, as specified in the instruction.
  - The VST monitors BPM and the time partiture values: as soon as detects a variation, all the times are rescaled around the current cursor to have the next notes have the right duration. Already triggered notes are not touched, so beware of fast varying tempo.
  - Most values are random, but are expressed as range. If you want e.g. move a note by a fixed not random value, just specify the same value for upper and lower level of a parameter.

Here is a list of the recognized tokens:

- "PROB <x>;"
  - Prefix for probabilistic execution.
  - Applies a probability <x> to the next instruction.
  - For most instructions this means that it it's executed with a certain probability. This is the default behavior if not specified otherwise.
  - For others the effect is more subtle, e.g. for MOD this is the probability of modify of a single note. See the relevant help for detail.
  - If the next instruction is invalid or another "PROB <x>;" then the prefix is lost and the next instruction has 100% probability, if not further prefixed.
  - If <x> is less than 1, it is assumed to be a fraction and multiplied by 100. Use a number less than 0.01 to set a probability less than 1%.
- "IF [&]num1 <OP> [&]num2;<instruction>"
  - Prefix for deterministic execution.
  - The num1/2 syntax specifies a constant.
  - The &num1/2 syntax specifies a MIDI CC number value. The channel used for this MIDI CC is the sequencer channel (See SEQCHANNEL), if applicable.
  - <OP> can be =, <, <=, >, >=, <> or !=, with the usual meaning.
  - If the condition is true, the <instruction> is executed, otherwise it is skipped.
  - If <instruction> is a GOTO or a BRANCH (see below) this can be used as conditional jump.
  - If prefixed by <PROB>, the IF is skipped randomly and the <instruction> is always executed if the IF was skipped.
  - The spaces are important. The <OP> must not have spaces. & must not have spaces after. One single space between items.
- "RESET;"
  - Move and reset.
  - Move <cursor> to the TAPE start and deletes all the notes AFTER <last>, that means <cursor> = 0 and <end> = <last>.
  - Copy the arpeggio notes on the TAPE, overwriting eventual modified notes.
  - To move only the cursor, without resetting the TAPE, use MOVE.
- "MOVE <minbars>,<maxbars>;"
  - Let <x> be an integer random number between <minbars> and <maxbars>.
  - Move the cursor back to <rcursor>-<x>. Must be positive so that the cursor can go back only.
  - The <cursor> is capped to the first note.
- "APPEND <minbars>,<maxbars>,<mincbars>,<maxcbars>;"
  - Let <x> be an integer random number between <minbars> and <maxbars>.
  - Let <y> be an integer random number between <mincbars> and <maxcbars>.

    - <minbars> and <maxbars> are clipped to the maximum numbers of bars BEFORE generating the random number, so setting an high number will select the first notes.
    - Copy the notes that trigger from the bar <rcursor>-<x> to the bar <rcursor>-<x>+<y> and append them after <end>.
    - The append is performed at a time such that the trigger of the last note is rounded to the next bar and the first appended bar starts at that time.
    - During the copy, <end> is advanced, so if <y> > <x> the copy can still proceed, provided that <x> is not 0.
    - If you want a copy with overwrite, first issue a delete command (see below).
- "DUP <minbars>,<maxbars>,<mincbars>,<maxcbars>;"
    - Let <x> be an integer random number between <minbars> and <maxbars>.
    - Let <y> be an integer random number between <mincbars> and <maxcbars>.
    - Copy the notes that trigger from the bar <rcursor>+<x> to the bar <rcursor>+<x>+<y> and append them after <end>.
- "GOTO <min>,<max>;"
    - Let <x> be an integer random number between <min> and <max>.
    - The instruction pointer is set to <x>.
    - The instruction number starts from 0 of the first instruction, does not include PROB or invalid instructions.
    - To see the correct instruction number, enable the debug and look for Op# <n>, P: <prob>%, "<command>" messages.
    - To avoid infinite loops, backward GOTO should have a PROB prefix.
    - In any case the total number of instructions is limited to 2000, so even an infinite loop will be broken.
    - If the actual pointer is over the last instruction, then the program ends.
    - If the actual pointer is before the first instruction, then the instruction pointer is set to the first instruction.
- "BRANCH <min>,<max>;"
    - Let <x> be an integer random number between <min> and <max>.
    - The instruction pointer is set to <current instruction> + <x>.
    - <min> and <max> can be negative.
    - The instruction number starts from 0 of the first instruction, does not include PROB or invalid instructions.
    - To see the correct instruction number, enable the debug and look for Op# <n>, P: <prob>%, "<command>" messages.
    - To avoid infinite loops, backward BRANCH should have a PROB prefix.
    - In any case the total number of instructions is limited to 2000, so even an infinite loop will be broken.
    - If the actual pointer is over the last instruction, then the program ends.
    - If the actual pointer is before the first instruction, then the instruction pointer is set to the first instruction.
    - Example: perform 50% of the time <instr1> and 50% of the time <instr2>, then perform always <instr3>:
      " ... PROB 50;BRANCH 3,3;<istr1>;BRANCH 1,1;<instr2>;<instr3>; ... "
- "SET <pos>,<pitch>,<duration>,<velocity>,<offvel>;"
    - Modify the first note after the <rcursor>+<pos> bar.
    - The trigger time is not modified.
    - <pos> is trimmed to 0 if negative.
    - If <pos> is too high, no note is modified.
    - All the parameters are optional.
    - If missing: <pos> = 0, <pitch> = 69, <duration> = 1 bar, <velocity> = 100, <offvel> = <velocity>.

    - <duration> can be a floating point number or in fraction form: e.g. 1.5 ===> 1.5 bars, 1/16 ===> 1/16 bars.
- "ADD <pos>,<pitch>,<duration>,<velocity>,<offvel>;"
    - Add a note at the <rcursor>+<pos> bar.
    - <pos> is trimmed to 0 if negative.
    - All the parameters are optional.
    - If missing: <pos> = 0, <pitch> = 69, <duration> = 1 bar, <velocity> = 100, <offvel> = <velocity>.
    - <duration> can be a floating point number or in fraction form: e.g. 1.5 ===> 1.5 bars, 1/16 ===> 1/16 bars.
    - To add chords or arpeggios, use multiple ADD instructions.
    - To have random parameters, put suitable modify instructions afterwards.
- "DEL <minbar>,<maxbar>;"
    - Delete all the notes that triggers between <rcursor>+<minbar> and <cursor>+<maxbar>.
    - <minbar> and <maxbar> can be fractional but must be >= 0. If <maxbar> <= <minbar> then no notes will be deleted.
    - Upper extreme excluded. E.g. DEL 0,1; will delete all notes between bar 0 and 1, but a note that triggers exactly on the edge of bar 1 will not be deleted.
    - Note: the following notes are NOT moved back to fill the hole.
- "DELODD;"
    - Delete all notes after <rcursor> that trigger in odd bars.
    - Note: the following notes are NOT moved back to fill the hole.
- "DELEVEN;"
    - Delete all notes after the <rcursor> that trigger in even bars. 0 is considered even.
    - Note: the following notes are NOT moved back to fill the hole.
- "MOD <minbars>, <maxbars>, <dtrigmin>, <dtrigmax>, <ddurmin>, <ddurmax>, <dpitchmin>, <dpitchmax>, <dvelmin>, <dvelmax>, <doffvelmin>, <doffvelmax>;"
    - Modify notes.
    - For all notes that trigger between <rcursor>+<minbars> and <rcursor>+<maxbars> a random number is generated.
    - If it's below the probability of the instruction (that it's 100% if it is not prefixed by PROB) then its parameters are modified by a random number between min and max (can be negative).
    - <minbar> and <maxbar> can be fractional but must be >= 0. If <maxbar> <= <minbar> then no notes will be modified.
    - Upper extreme excluded. E.g. MOD 0,1,...; will modify all notes between bar 0 and 1, but a note that triggers exactly on the edge of bar 1 will not be touched.
    - Note: for each note is generated a different set of random numbers.
    - Note: the Tape is ordered for trigger time. This instruction may modify the note order.
    - Note: if the last note in the TAPE is modified such that the trigger time crosses a bar border, then the TAPE gains a new bar, since <rcursor> is rounded up to the next bar.
    - Not quantized. Use QUANTIZE to quantize after the modify.
    - Missing parameters are set to 0.
    - <dtrigmin>, <dtrigmax>, <ddurmin> and <ddurmax> can be a floating point number or in fraction form, with sign: e.g. 1.5 ===> 1.5 bars, 1/16 ===> 1/16 bars.
- "QUANTIZE <minbar>,<maxbar>,<qtrigger>,<qduration>,<qpitch>;"
    - Quantize all the notes between <rcursor>+<minbars> and <rcursor>+<maxbars>.
    - Missing parameters are set to 0.
    - If <qxxx><=0 do not quantize.
    - If <qtrigger> = <n> then quantize trigger time to nearest 1/<n> of bar (e.g. 16=1/16).
    - If <qduration> = <n> then quantize duration in excess of 1/<n> of bar (e.g. 16=1/16).

    - If <qpitch> = <n> then quantize to the nearest semitone divisible by <n>. Can be fractional to e.g. quantize to different tonal scale with smaller semitone.
- "ROLLS <minbar>,<maxbar>,<type>;"
    - Modify velocity for all the notes between <rcursor>+<minbars> and <rcursor>+<maxbars>.
    - Missing parameters are set to 0.
    - <type> = 0 rolls 0->127, 1 rolls 127->0
- "C2A <minbars>,<maxbars>,<dir>;"
    - Chord to Arpeggio.
    - All notes that trigger between <rcursor>+<minbars> and <rcursor>+<maxbars> are spread in the interval depending on <dir> option:
    - <dir>=0 ascending pitch, 1 descending, 2 ascending+descending, 3, descending+ascending, 4 ascending with swap of third and fifth (1,3,2,4,5,6..., e.g. fundamental, 5th, third, 7th/8th)
    - If there are less than 2 notes, it does nothing.
    - The notes don't have to be a true chord. Any note in the interval will be moved.
    - Duplicate notes are deleted. The pitch must be almost exact (difference < 1 cent).
- "A2C <minbars>,<maxbars>;"
    - Arpeggio to chord.
    - All notes that trigger between <rcursor>+<minbars> and <rcursor>+<maxbars> are put at <cursor>+<minbars>.
    - Duplicate notes are deleted. The pitch must be almost exact (difference < 1 cent).
    - The notes are put at <rcursor> + <minbars>, which can be fractional. Use MOD to move or increase duration.
- "REV <minbar>,<maxbars>,<ref>;"
    - Reverse pitch pattern around a reference note.
    - For all notes that trigger between <rcursor>+<minbars> and <rcursor>+<maxbars>, <pitch> = <ref> - (<pitch> - <ref>) = 2 * <ref> - <pitch>.
    - Default <ref>=69.
    - The function is reversible: applied 2, 4 etc. times reverts to the original notes.
- "INV <minbar>,<maxbars>;"
    - Inverse pattern.
    - For all notes that trigger between <rcursor>+<minbars> and <rcursor>+<maxbars>, swap positions.
- "RND <minbar>,<maxbars>;"
    - Random pattern.
    - For all notes that trigger between <rcursor>+<minbars> and <rcursor>+<maxbars>, randomly shuffle positions.
- "DOUBLE <minbar>,<maxbars>;"
    - 2x time pattern.
    - For all notes that trigger between <rcursor>+<minbars> and <rcursor>+<maxbars>, stretch 2x the pattern. Move accordingly the following notes.
- "HALF <minbar>,<maxbars>;"
    - 0.5x time pattern.
    - For all notes that trigger between <rcursor>+<minbars> and <rcursor>+<maxbars>, stretch 0.5x the pattern. Move accordingly the following notes.
- "SHIFT <minbar>,<maxbars>,<dir>;"
    - Shift notes.
    - For all notes that trigger between <rcursor>+<minbars> and <rcursor>+<maxbars>, shift according to <dir>, first or last notes are duplicated.
    - <dir> > 0 shift left 1 position only, < 0, shift right 1 position only
    - Example for 4 notes.

    - <dir> = -1, <a><b><c><d> ==> <a><a><b><c> (<d> is lost)
    - <dir> = 1, <a><b><c><d> ==> <b><c><d><d> (<a> is lost)
- "ROT <minbar>,<maxbars>,<dir>;"
    - Rotate notes.
    - For all notes that trigger between <rcursor>+<minbars> and <rcursor>+<maxbars>, rotate according to <dir>, first or last notes go to the other end.
    - <dir> > 0 shift left 1 position only, < 0, shift right 1 position only
    - Example for 4 notes.
    - <dir> = -1, <a><b><c><d> ==> <d><a><b><c>
    - <dir> = 1, <a><b><c><d> ==> <b><c><d><a>
- "INTERP <pos>,<n>,<glide>,<dur>;"
    - Take the nearest note at <rcursor>+<pos> and the next one.
    - If there are not at least two notes after <rcursor>+<pos> until the end of the tape, then do nothing (you must have two notes to interp).
    - If <n> < 2, do nothing.
    - Substitute the first note with <n> smaller notes to fill the interval with the next.
    - If <glide>=0 then all <n> notes are equal to the first, except for the trigger time and duration.
    - If <glide>!=0 then each note has velocity and pitch interpolated.
    - If <glide>=1 the pitch is rounded to the next integer
    - If <glide>=2 the pitch is not rounded.
    - <dur> is the multiplier (default=1 if not specified) of the note duration. If 1, the note will fill completely the interval between the notes. 0.5 means half, etc...
- "FREEZE <minbar>,<maxbar>,<dmin>,<dmax>,<n>,<dur>;"
    - If <n> < 1 do nothing.
    - Let <x> be a random number between <minbar> and <maxbar>.
    - Let <y> be a random number between <dmin> and <dmax>.
    - Pick the first note following <rcursor>+<x>.
    - If there is not a note after <rcursor>+<x> until the end of the tape, then do nothing.
    - Delete all notes between <rcursor>+<x> and <rcursor>+<x>+<y>.
    - If <n> = 1, create a note equal to the selected one, but stretched to duration of <y> bars.
    - If <n> > 1, fill the interval <rcursor>+<x> - <rcursor>+<x>+<y> with <n> copies of the selected note, but of duration <y>/<n>. E.g. if <n> = 16, then fill the interval with copy of the selected note of <y>/16 bar.
    - <dur> is the multiplier (default=1 if not specified) of the note duration. If 1, the note will fill completely the interval between the notes. 0.5 means half, etc...
- "EXIT <seconds>;"
    - Stop the program after at least <seconds>.
    - The notes in the Tape will be played until the end, but no new instructions are executed.
    - To further randomize the song duration, prefix with PROB and a low probability.
    - The only way to start over will be the reset with the VST var knob or the reload of the instrument file.

Sample declaration and prefiltering Instructions

SAMPLE <slot_number>

OR

SAMPLE <slot_number>; <numeric_code>; <2nd_harmonic_numeric_code>; <2nd_harmonic_amplitude>; <2nd_harmonic_frequency>; <2nd_harmonic_phase>;<3rd...

OR

SAMPLE <slot_number>; "filename"; <normalization_mode>; <center_note>; <loops>; <direction>; <startw>; <endw>; <relstart>; <loopstartw>; <loopendw>; <crossfade>; <crossfaderel>; <crossfadefwbw>

This instruction is used to declare waveforms or synth data for further use in the file.

The slots are dynamically allocated, based on the highest slot number used in any SAMPLE, SAMPLEUI, SAMPLES or RENDER instruction.
The limit is 2 billion of slots, but probably the memory will end sooner.
No further memory is consumed if a slot is not occupied.
The slot defined can be even sparse.

Synth data is always mono. Sampled data can be mono or stereo depending on the input file. Mono data is stored in left channel and right is left empty.

OSCG function uses left for both channels if the sample is mono.
Note that stereo OSC processing is stereo even with mono samples: the phase, frequency and gain can be different between left and right due to dephase and/or detune and so even with mono samples the output is stereo.

This instruction can be put everywhere, but the results are seen globally: a first pass gathering all the SAMPLE instructions in the file is performed, eventually overwriting older SAMPLEs. The final state after this step is what is seen in the POST and normal LAYERs. If you use different sample slot number in every instruction, there is no difference where you put the instruction.

First parameter is mandatory. Missing parameters retain their default values.

<slot_number> is the slot in which put the sample. If it's already occupied, the old sample will be overwritten and lost.

In Play mode if the referenced slot is empty, the oscillator will not produce sound. If all oscillators in a layer have empty or invalid slot, the layer will not be triggered.

The slot in the OSCG function is automatable. In this case it is advisable to use consecutive slot numbers for storing the waveforms.
There is the SAMPLEOFF instruction that further modifies the slot number selected: see below.

The first syntax is used to delete a slot already filled in previous instructions/included files.
This syntax is meant to make sure that a slot is empty and does not produce sound, e.g. to have a way to silence a layer.

The second syntax, in which the second parameter is a number, is to declare a synthetized waveform with a maximum of 31+1 harmonics.

The syntax allows specifying different type, frequency, amplitude and phasing for each harmonic, to be able to play detuned unison with a single layer, even with mixed waveform type.

<numeric_code> specify the base waveform type (first harmonic):

- 0 = sine,
- 1 = square (50% PWM),
- 2 = sawtooth,

- 3 = triangle,
- 4 = noise,
- 5 - 8 = sine raised to 0.3, 0.1, 0.03 and 0.01, giving a continuous simil square waveform, with increasing high frequency content.
- 9 - 12 = continuous simil sawtooth with formula (x/PI)-(x/PI)^y, with y set at 9, 19, 39 and 59, giving increasing high frequency content.

  The first 5 base waveforms are very fast, but square and saw have very high frequency content and can lead to artifacts. They are good only for automations.
  To cope with this use the smoother versions, with less high frequencies, so a low pass filter can be avoided if they sound as desired.
  These are slower but with better behavior at high frequencies and suitable for direct use in the oscillators.

The following parameters are optional (default is to not have further harmonics) and specify:

- The type of the 2nd-32nd harmonic: same coding than the first harmonic.
- The relative amplitude (compared to the first harmonic) of the harmonic. Must be positive float.
- The relative frequency (compared to the first harmonic) of the harmonic. Must be positive float. Can be fractional for detuned unison and even less than 1.
- The additional phase (compared to the first harmonic) in radians of the harmonic. Should be between -PI and PI.

Note: human ear is almost insensitive to phase, but with the correct phasing, the amplitude of the unison can be limited and so the sound can be amplified more before clipping.

The first harmonic has always initial phase 0 (before phase dephasing), amplitude 1.0 and the base frequency of the oscillator.
This function is meant to construct more complex synthetized waveforms, in Fourier style. But you can use all types (and mixed) of waveform as base and the frequencies must not be consecutive integer multiplies.
WARNING: if one of the harmonic is incomplete (e.g. only 1, 2 or 3 parameter specified), then the whole instruction is rejected, with an error in the output log.

The third syntax, in which the second parameter is a string, is to declare a sampled waveform.
"filename" is the file name of the waveform. Can contain a relative path, relative to the instrument file path, or to <my documents>\Crescendo, if it's not found.
Absolute paths do work but are not recommended for portability.
The algorithm is the following:

1) Check if "filename" is an absolute path and check for existence.
2) if it does not exist, check if it is a path relative to the instrument file path, e.g. check if <instrument file path>\"filename" exist. (filename can contain also a relative path, e.g. samples\foo.aif)
3) if it does not exist, check if it is a path relative to <my_documents>\Crescendo. (filename can contain also a relative path, e.g. samples\foo.aif)

The file types currently supported are:

- mono and stereo little endian uncompressed WAVE files, PCM, with sample type of 8, 16, 24, 32, bit integer and 32 or 64 bit IEEE floating point.
- mono and stereo big endian uncompressed AIFF files, PCM, with sample type of up 32 bit integer.
- mono and stereo little endian and big endian uncompressed AIFFC files, PCM, with sample type of up 32 bit integer and 32 or 64 bit IEEE floating point.

<normalization_mode> is the modality of normalization of the sample, useful e.g. to have all samples to the same level to apply the correct velocity factor.

0 = no normalization
1 = normalization on maximum absolute value for the whole sample.
2 = normalization on maximum absolute value separately for left and right channel
3 = normalization on RMS value for the whole sample.
4 = normalization on RMS value separately for left and right channel.

Note:
Maximum amplification is 1000x (+60dB) for each normalization method.
RMS normalization is performed to have RMS of -6dB (0.25).
Take care on setting the BASEG or the GAIN to have the desired final level.

RMS value is calculated excluding the first 10% of the samples (supposing that it's the attack) and the last 60% of the samples (supposing it's the release).

<center_note> is the reference note of the sound contained in the sample at the specified position. Default 69 (A3).

➢ Range 0-128. Can be a floating point value for fine tuning (e.g. 69.1234567 meaning A3 plus 12.34567 cents).
NOTE: the center_note is expected to be given assuming the standard tuning of 440Hz and equal temperament: if you then wish to play the sample with another tuning system or another temperament, then the BASEF, KEYTRACK, KEYCENTER and the temperament could be changed to your requirements.

<loops> specify the number of loops. 0=unlimited loops, 1=one shot, >=2 the specified number of loops. Default 1 (one shot).
When the number of loops expires, the sample continues with the rest of the sample after the loop, until the end (for looped=0 this never happens).
If the release arrives before the number of loops completes, there are six cases:

- <loops> = 1 (one shot), separate release off: the sample position is not changed and the sound continues until the end.
- <loops> = 1 (one shot), separate release on: the sample is crossfaded with the new release position and both continue until only the release portion remains.
- <loops> >=2 (limited loop count), separate release off: the sample position is not changed and the sound continues until the end.
- <loops> >=2 (limited loop count), separate release on: the sample is crossfaded with the new release position and both continue until only the release portion remains.
- <loops> =0 (unlimited loop count), separate release off: the sample position is not changed and the sound continues looping.

- <loops> =0 (unlimited loop count), separate release on: the sample is crossfaded with the new release position and both continue until only the release portion remains.

<direction> is the direction of play, even in one shot mode. 0=forward, 1=backward, 2=forward+backward, 3=backward+forward. Default 0.

Let <last> be the number of the last sample, namely the total number of samples in the file.

<startw> is the first sample to be played. If it is less than 0 it is used the absolute value for symmetry with <endw>. Default 0.

<endw> is the last sample to be played (depending on looping, see below). If it is <=0 then it is used <last> less this value (e.g. 0 is last sample, -100, <last>-100 etc...). Default 0.

<relstart> specifies a portion of samples between <relstart> and <endw> to be used as release. Default 1e10.
If it is a large number (1e10 or in any case above <endw>), then separate release is disabled and normal release is used (see below).
If it is <0, <relstart> is set to <endw> - this value, e.g. if it's 10000 it means that release is played from sample <endw>-10000 to <endw> etc...
If the final value is below <startw>, then the separate release is disabled too.
If <relstart> is >= <startw>, then the release is played from sample <relstart> to <endw>.

Normal release, means that when the release kicks on, the sample continue to be played as per the current looping mode.
Separate release means that the sound continue to be played (one shot or looped) as in normal release but it is gradually crossfaded with the release portion from <relstart> to <endw>.
With these options, no separate layers for sustain and release is needed in the common case of same envelope/filter settings for sustain sample and release sample: a single file will contain the samples of the loop and just after comes the release part. Just set relstart to the part where the release portion starts.

<loopstartw> specifies the start loop sample (ignored if <loops> is 1). If it is less than 0 it is used the absolute value for symmetry with <loopendw>. If it is less than <startw>+1, it is clipped to that value. Default 0.

<loopendw> specifies the end loop sample (ignored if <loops> is 1). If it is <=0 then it is set to <endw> less this value. If it is higher than <endw>-1, it is clipped to that value. Default 0.

<crossfade> is the number of samples to crossfade between the end and the start of the loop to avoid clicks. Default 5.
This value is automatically lowered if it's bigger than the loop/sample size or if it's bigger than <loopstartw> - <startw>.
Note: to be able to perform a minimal crossfade, <loopstartw> should be slightly higher than <startw>.

If <loops> is 1, it specifies the number of samples of fading the sample to zero at the end to avoid clicks.

<crossfaderel> is the number of samples to crossfade between the sample and the release range to avoid clicks. Ignored if normal release. Default 5.

This value is automatically lowered if it's bigger than <endw> - <relstart>.

<crossfadefwbw> is the number of samples to crossfade between forward and backward direction to avoid clicks. Ignored if <direction> < 2. Default 5.
This value is automatically lowered if it's bigger than half the loop size.

NOTE: if <startw> and <endw> or <loopstartw> and <loopendw> are not in ascending order, they are swapped, so you can specify backward played data in either way.
To be able to correctly use negative values for <startw> and <endw> for reversed samples <startw> should point to the samples near the start of the file and <endw> to the other end, so <loopstartw> and <loopendw>.

Other adjustments are made. The precise algorithm is:
1) Adjust <startw> and <endw> if they are <=0.
2) Swap back <startw> and <endw> if they are swapped.
3) Clip <startw> to [0,<endw>), Clip <endw> to (<startw>,<filesize>]
4) Adjust <loopstartw> and <loopendw> if they are <=0.
5) Swap back <loopstartw> and <loopendw> if they are swapped.
6) Clip <loopstartw> to [<startw>+1,<loopendw>), Clip <loopendw> to (<loopstartw>,<endw>-1]
7) Check <relstart>.
8) Trim the <crossfadeXXX> if they are too big.

NOTE: the performances were optimized for big files for which you need only a small section (e.g. a 2GB file with all piano samples and you need just one):

- only the data from startw to endw is allocated and read. So set these values (endw in particular) to the strict necessary: endw can be left to default to force the loading of all file, but if the sample is looped and the release is normal, these samples will never be played and are loaded in memory for nothing.
  Also the normalization is affected by <endw>: by including unhearable samples you also alter the normalization factors.
  Moreover the VST does not optimize for overlapping portions of the same file for different sample slots: the shared data will be duplicated.

NOTE: for a pictorial depiction of all the options, see the relative section in this file.

SAMPLES <slot>, <length>, <channels>, <SF>, <center_note>, <normalization_mode>, "<sample_1>", <scaleL_1>, <scaleR_1>, <center_note_1>,..., "<sample_N>", <scaleL_N>, <scaleR_N>, <center_note_N>

This instruction put in the indicated slot a combination of multiple samples, resampled and scaled and optionally normalizes the result.

In the slot number <slot>, an audio sample of <length> seconds, <channels> channels (mono or stereo), <SF> sample frequency and <center_note> reference note is created.

This sample is initialized with silence.

One or more samples (taken from external files) can be accumulated in this buffer, resampled and scaled (see below).

If only six parameters are provided, no error is given, but the sample slot is not touched.

If seven or more parameters are given, the sample slot is eventually erased and then if an error is generated, the slot remains empty.

An incomplete sequence, after the seventh parameter, does not signal error.

As soon as a sequence <sample_i>…<center_note_i> is completed, a check is performed and the instruction is rejected (and the sample slot left empty) if there is some error: file not found or center note out of bound.

"<sample_i>" is the path to the i-th sample
<scaleL/R_i> are the multiplicative factor for left/right channel (to scale and/or pan the sample). Can be negative to invert the phase.
<center_note_i> is the center note of the stored i-th sample (the sample frequency and bit depth/format are read from the file).

If a sequence is complete and no error is signaled, the file is loaded and resampled as if it should be played at <SF> hertz and with <center_note> pitch, assuming that the original file has center note <center_note_i> with standard tuning (440Hz if <center_note_i> = 69).

The resampled file is summed into the buffer, scaling the left and right.

If the buffer is mono and the sample is stereo, the samples are scaled with the correct scale factor and then summed to obtain a mono sample to mix into the buffer.

If the sample is mono and the buffer is stereo, the sample is added to both channels, with the appropriate scale factors.

After all the samples are processed, the <normalization_mode> (the same as of the SAMPLE instruction) is applied.

If a sample is shorter is zero padded. If it is longer it is cut. There is a limited fadeout at the start and end to avoid clicks.

Only one shot samples supported. If you want looping, you must use OSCG with the looping syntax (see below).

This instruction can be put everywhere, but the results are seen globally: a first pass gathering all the SAMPLES instructions in the file is performed, eventually overwriting older SAMPLEs. The final state after this step is what is seen in the POST and normal LAYERs. If you use different sample slot number in every instruction, there is no difference where you put the instruction.

NOTE: SAMPLES uses the current resample quality. QUALITY can be changed multiple times, so a SAMPLES can be made at very high quality and then the quality can be lowered at runtime for speed reasons, e.g.:

```
QUALITY 4096,31,0,1,4  // High quality setting
SAMPLES ...
...
SAMPLES ...
QUALITY 4096,7,0,1    // Lower quality settings for the runtime
```

Vice versa if you need NN interpolation for the samples because you plan to use them as LFOs and don't want overshoot, you can have good runtime resample quality:

```
QUALITY 4096,0,0,0    // NN
SAMPLES ...  // Samples for LFOs
QUALITY 4096,31,0,1,4  // High quality for other samples
SAMPLES ...
QUALITY 4096,7,0,1    // Normal quality for runtime
```

RENDER <slot>,<length>,<channels>,<SF>,<center_note>,<normalization_mode>, <slot_1>,<scaleL_1>,<scaleR_1>,<center_note_1>,<atk_1>,<dcy_1>,<sus_1>,<rel_i_1>,<rel_t_1>,..., <slot_N>,<scaleL_N>,<scaleR_N>,<center_note_N>,<atk_N>,<dcy_N>,<sus_N>,<rel_i_N>,<rel_t_N>

This instruction put in the indicated slot a combination of multiple sample slots, resampled and scaled and optionally normalizes the result.

In the slot number <slot>, an audio sample of <length> seconds, <channels> channels (mono or stereo), <SF> sample frequency and <center_note> reference note is created.

This sample is initialized with silence.

One or more sample slots (already defined) can be accumulated in this buffer, resampled and scaled (see below).

If only six parameters are provided, no error is given, but the sample slot is not touched.

If seven or more parameters are given, the sample slot is eventually erased and then if an error is generated, the slot remains empty.

An incomplete sequence, after the seventh parameter, does not signal error.

As soon as a sequence <slot_i>…<rel_t_i> is completed, a check is performed and the instruction is rejected (and the sample slot left empty) if there is some error: center note out of bound.

<slot_i> is the slot number of the i-th sample
<scaleL/R_i> are the multiplicative factor for left/right channel (to scale and/or pan the sample). Can be negative to invert the phase.
<center_note_i> has different meaning, depending if the sample slot <slot_i> has sampled or synth data:

If <slot_i> is synth, then <center_note_i> is the note at which <slot_i> is rendered and stored into the buffer. The same frequency is heard should the buffer be played at <center_note>.

If <slot_i> is sampled, then <center_note_i> will overwrite the original center note of the sampled slot. If you do not want special effects (like detuning with respect to other slots eventually accumulated), then <center_note_i> should be set to the original center note set in the <slot_i> declaration.

<atk_i>, <dcy_i>, <sus_i>, <rel_i_i> and <rel_t_i> define a standard multiplicative ADSR envelope, added to the sound data. <rel_i_i> is the release instant, in seconds from the start of the buffer, at which the release phase will start.

If a sequence is complete and no error is signaled, the sample slot is resampled as if it should be played at <SF> hertz and with <center_note> pitch. For sampled slots the actual rendered pitch will be <center_note> only if <center_note_i> is equal to the original center note of the sampled slot <slot_i>.

The resampled slot is summed into the buffer, scaling the left and right.

If the buffer is mono and the sample is stereo, the samples are scaled with the correct scale factor and then summed to obtain a mono sample to mix into the buffer.

If the sample is mono (or synth) and the buffer is stereo, the sample is added to both channels, with the appropriate scale factors.

Any type of source sample slot is supported: each sample slot is “played” as if the note on is triggered at the instant 0 and the note off is triggered at instant <rel_i_i>.
All the features, including the looping, direction, separate release, etc. are honored (see SAMPLE).

If <rel_i_i> + <rel_t_i> is above <length>, the sample will be cut. If it is below, the sample will be zero padded.

After all the <slots_i> are processed, the <normalization_mode> (the same as of the SAMPLE instruction) is applied.

The final sample slot will be of one shot type. If you want looping, you must use OSCG with the looping syntax (see below).

This instruction can be put everywhere, but the results are seen globally: a first pass gathering all the RENDER instructions in the file is performed, eventually overwriting older SAMPLEs. The final state after this step is what is seen in the POST and normal LAYERs. If you use different sample slot number in every instruction, there is no difference where you put the instruction.

NOTE for sampled <slots_i>: RENDER uses the current resample quality. QUALITY can be changed multiple times, so a RENDER can be made at very high quality and then the quality can be lowered at runtime for speed reasons, e.g.:

```
QUALITY 4096,31,0,1,4  // High quality setting
RENDER ...
...
RENDER ...
QUALITY 4096,7,0,1     // Lower quality settings for the runtime
```

Vice versa if you need NN interpolation for the samples because you plan to use them as LFOs and don't want overshoot, you can have good runtime resample quality:

```
QUALITY 4096,0,0,0    // NN
RENDER ...  // Samples for LFOs
QUALITY 4096,31,0,1,4  // High quality for other samples
RENDER ...
```

QUALITY 4096,7,0,1   // Normal quality for runtime

FILTER <num>,<frequency_hz>,<resonance>
OR
FILTERSEMI <num>,<frequency_semitones>,<resonance>

This declaration is a prefix applicable to SAMPLE, SAMPLES or RENDER declarations.
Can be combined with EQ1DB, EQ3DB, in any order and all filtering will be performed, in particular multiple instances of the filters are executed, e.g. a 5 band equalizer can be created with 5 consecutive EQ1DB.
Can be combined also with WIDEPAN, in any order, but WIDEPAN is single instance.
Other uses causes the row to be ignored.
The syntax is FILTER/FILTERSEMI <p1>, <p2>, <p3>, SAMPLE/SAMPLES/RENDER instruction.
This prefix declarations activate the pre-filtering of the final waveform, just before the normalizing step or the pre-equalizing step. For SAMPLE instruction the filtering is applied only on sampled slots.
<num> is the type of filter, with the same coding of the FILTER instruction. See below.
<frequency> is in Hz for filter and in semitones (69 = 440 Hz) for FILTERSEMI.
<resonance> is the resonance of the filter if <num> >=0. See FILTER instruction below.
The filtering is applied to the sample directly in the internal memory. So if you have only a fixed constant filter on your samples, that is proportional to the actual frequency of the note, you can avoid of applying it in real time, saving CPU time.
The results are not identical because with this instruction the filtering is applied on the original sample, i.e. before the sinc interpolation and resampling.

EQ3DB <eqtype>,<lowfreq>,<highfreq>,<lowgain>,<midgain>,<highgain>

This declaration is a prefix applicable to SAMPLE, SAMPLES or RENDER declarations.
Can be combined with EQ1DB, FILTER, FILTERSEMI, in any order and all filtering will be performed, in particular multiple instances of the filters are executed, e.g. a 5 band equalizer can be created with 5 consecutive EQ1DB.
Can be combined also with WIDEPAN, in any order, but WIDEPAN is single instance.
Other uses causes the row to be ignored.
The syntax is EQ3DB <eqtype>,<lof>,<hif>,<glo>,<gmi>,<ghi>, SAMPLE/SAMPLES/RENDER instruction.
This prefix declarations activate the pre-equalizing of the final waveform, just before the normalizing step. For SAMPLE instruction the filtering is applied only on sampled slots.
<eqtype> is the type of filter, with the same coding of the EQ3DB instruction. See below.
See EQ3DB instruction below also for the other parameters.
The filtering is applied to the sample directly in the internal memory. So if you have only a fixed constant equalizer on your samples, that is proportional to the actual frequency of the note, you can avoid of applying it in real time, saving CPU time.
The results are not identical because with this instruction the filtering is applied on the original sample, i.e. before the sinc interpolation and resampling.

EQ1DB <eqtype>,<lowfreq>,<highfreq>,<gain>,<lowres>,<highres>

This declaration is a prefix applicable to SAMPLE, SAMPLES or RENDER declarations.
Can be combined with EQ3DB, FILTER, FILTERSEMI, in any order and all filtering will be performed, in particular multiple instances of the filters are executed, e.g. a 5 band equalizer can be created with 5 consecutive EQ1DB.

Can be combined also with WIDEPAN, in any order, but WIDEPAN is single instance.
Other uses causes the row to be ignored.
The syntax is EQ1DB <eqtype>,<lof>,<hif>,<gain>,<lowr>,<hir>, SAMPLE/SAMPLES/RENDER instruction.
This prefix declarations activate the pre-equalizing of the final waveform, just before the normalizing step. For SAMPLE instruction the filtering is applied only on sampled slots.
<eqtype> is the type of filter, with the same coding of the EQ1DB instruction. See below.
See EQ1DB instruction below also for the other parameters.
The filtering is applied to the sample directly in the internal memory. So if you have only a fixed constant equalizer on your samples, that is proportional to the actual frequency of the note, you can avoid of applying it in real time, saving CPU time.
The results are not identical because with this instruction the filtering is applied on the original sample, i.e. before the sinc interpolation and resampling.

WIDEPAN <widening>,<pan>

This declaration is a prefix applicable to SAMPLE, SAMPLES or RENDER declarations.
Can be combined with EQ1DB, EQ3DB, FILTER, FILTERSEMI, in any order and all filtering will be performed, in particular multiple instances of the filters are executed, e.g. a 5 band equalizer can be created with 5 consecutive EQ1DB.
WIDEPAN is single instance.
Other uses causes the row to be ignored.
The syntax is WIDEPAN <widening>,<pan>, SAMPLE/SAMPLES/RENDER instruction.
This prefix declarations activate the widening and panning of the final waveform, just before the normalizing step. For SAMPLE instruction the modify is applied only on sampled slots.
<widening> is the widening factor. See WIDE instruction below.
<pan> is the panning factor. See PAN instruction below.
The modify is applied to the sample directly in the internal memory.
If the sample is mono and PAN is not zero, then the sample is made stereo to perform the panning.
If <widening> and <pan> are both zero, the final sample will be mono: this can be useful to be able to use the fast mono oscillator, which uses only the left channel and so it’s best used with mono samples.

SAMPLEOFF <sample offset>

This declaration is related to sampled data processing, but that can go anywhere in the file, with different meanings.

Set the sample offset value. If put in the COMMON section, it sets the default sample offset. 0 by default.
If put in a POST section or in a LAYER, it sets the sample offset for that section only.

What is sample offset?
The oscillators have a sample slot number, partially automatable (see oscillator description below), that is used to select the actual sample to play.

A fixed number does not allow to put a common expression in the common section, if the sample varies for each layer, e.g. for a multi sampled instrument.
If the automation and modulation expressions are the same, you can put the common expression(s) in the common section, to avoid repeating them and putting a different SAMPLEOFF declaration in each layer to tell the VST to use another sample for that layer.

The first 200 (0-199) slots are not offset, to put in them constant samples like sinusoid for modulation etc.

The offset is applied after the automation to further exploit this behavior (see examples above).

Default Instructions

These instructions allow to specify some default behavior of the oscillators (See OSCG functions). They specify the formulas for some default parameters, like GAIN or FREQ.

Multichannel note: although the formulas are fixed for a given LAYER (or are global to all LAYERs), the actual channel of the LAYER is used when picking MIDI CCs values for the final value calculation. So when the $<MCC> or &<MCC> syntax is used, this channel picking is implied.

BASEG is the base gain for the oscillators, used to calculate the default gain of the oscillators.

Default 1.0. Limits: between 0.001 and 1000.0.
It can be modified in the COMMON section with the **BASEG <number>** instruction.
E.g. BASEG 0.5

If set in a LAYER, it overwrites, for that LAYER, the global value.
If set to a negative value, the global value will be used.
If set to a positive value, the global value will be overwritten.

If the syntax **BASEG $<MCC>** is used, then the MIDI CC number <MCC> is sampled at trigger time and its value is used as BASEG. The scaling of the MIDI CC is the same of MCC2. No check is made on the final value. If an invalid <MCC> number is given, BASEG will be zero.

The instruction is declarative: the last value set is used at runtime in the layers.

VELTRACK is the exponent of the ON velocity, used to calculate the default gain of the oscillators.

Default 2.0. Limits: between 0 and 10.
It can be modified in the COMMON section with the **VELTRACK <number>** instruction.
If set to less than 0 in the COMMON section, the default value will be used.
E.g. VELTRACK 3.0

If set in a LAYER, it overwrites, for that LAYER, the global value.
If set to a negative value, the global value will be used.
If set to a positive value, the global value will be overwritten.

If the syntax **VELTRACK $<MCC>** is used, then the MIDI CC number <MCC> is sampled at trigger time and its value is used as VELTRACK. The scaling of the MIDI CC is the same of MCC2. No check is made on the final value. If an invalid <MCC> number is given, VELTRACK will be zero.

The instruction is declarative: the last value set is used at runtime in the layers.

GAINMOD <MCC1>,<MOD1>,...,<MCCN>,<MODN>

Global or local modulation of gain by MIDI cc.
If the gain is to be modulated by some MIDI CC and linearly in dB (perceptual), then the modulation can be put here for speed reasons.

All the GAINMODs specified in the COMMON section are added to the GAINMODs specified in a LAYER, without duplicates. LAYER's specification has priority.

If a local <MOD<i>>, after the merging, is 0, then the GAINMOD slot is deleted. This is for deleting a COMMON GAINMOD in a specific layer.

No check is made for duplicates in the same level (COMMON or LAYER <i>). The behavior is UNDEFINED.
The default is to not have any modulation.

<MCC1>...<MCCN> is the MIDI CC number of the modulator. See table above for the codes.

<MOD1>...<MODN> is the sensitivity in dB:

- if the MIDI CC goes from 0 to 1 (see MCC2 help below for the scaling: some MCCs are not scaled), the gain is modified by MOD<i> (with sign) dB. 20 means a 10 fold increase in GAIN.
- All the <MOD<i>> can be a real number, in this case capped between -400 and 400 dB, or the expression $<MCC>. In this case <MCC> is capped between 0 and 1000 and specifies a MIDI CC number that contains a value scaled as the MCC2 instruction prescribes. This value is continuously multiplied by the value of the MIDI CC number <MCC<i>>.
- Since the standard MIDI CC are always scaled to [0, 1[, at least one between <MCC<i>> and <MCC> should be a VST VAR with a proper range, otherwise the effect will be negligible.

The GAIN variable will be modified just after the velocity has been applied.
Polyphonic aftertouch (MIDI CC 132), KEYs, Velocities will be correctly applied per-LAYER.

The instruction is declarative: the order is not important. All the couples are merged as if there is one single instruction at the start of the COMMON section or LAYER.

GAINENV <amount>,<delay>,<atk>,<hold>,<decay>,<sustain>,<release>

Global or local multiplicative envelope for the GAIN.
If a mono envelope with constant or automated for the gain is sufficient, then the parameters can be put here for speed reasons.
Global default is 1, 0, 0, 0, 0, 1, and 10000: an envelope with “infinite” release time, 100% sustain and 0 seconds of attack, decay, delay and hold. Local default is to use the global values.

All the parameters can be a real number, in this case capped between -10000 and 10000, or the expression $<MCC>, or the expression &<MCC>. In this last two cases <MCC> is capped between 0 and 1000 and specifies a MIDI CC number that must be sampled at trigger time ($) or continuously (&) and scaled as the MCC2 instruction prescribes.

<amount> is the multiplicative factor for the envelope. Should be 1.0 since the GAIN is multiplied by this factor and there is already BASEG to modify the base gain.

<delay> is the delay before the attack.
<atk> is the attack time. If >0 use exponential curve, else use linear curve.
<hold> is the hold time between end attack and start decay
<decay> is the decay time. If >0 use exponential curve, else use linear curve.
<sustain> is the sustain level (1 = full 100%).
<release> is the release time. If >0 use exponential curve, else use linear curve.

The instruction is declarative: the last values set are used at runtime in the layers.

GAINLFO <num>,<param>,<freq>,<amount>,<delay>,<atk>,<release>

Global or local LFO for the GAIN.
If a multiplicative mono LFO with constant or automated parameters, zero starting phase (after the delay) and modulation in dB for the gain is sufficient, then the parameters can be put here for speed reasons.
Global default is all zeros that means LFO disabled. Local default is to use the global values.

All the parameters can be a real number, in this case capped between -10000 and 10000, or the expression $<MCC>, or the expression &<MCC>. In this last two cases <MCC> is capped between 0 and 1000 and specifies a MIDI CC number that must be sampled at trigger time ($) or continuously (&) and scaled as the MCC2 instruction prescribes.

<amount> is the peak gain and attenuation in dB.

<num>,<param>: Base waveform.
    <num>:
        0 = Sine wave distorted with power <param> like the function POWABS. With <Param> near 0 the waveform approaches the square wave.
        1 = Square wave with PWM(%) = <param> * 100. If <param> <= 0, PWM is zero. If <param> >= 1, PWM is 100%.
        2 = Triangular wave with slope width given by <param>: <=0 and >=1 give sawtooth waveform and 0.5 triangular waveform. The other values give a sawtooth with unequal slopes.
        3 = Smooth sawtooth with formula (x/PI)-POWABS(x/PI,<param>). <param> should be >>1, but no check is made. The more <param>, the more the high frequency content.
        4 = Triangular distorted with power <param> like the function POWABS. With <Param> near 0 the waveform approaches the square wave.
        5 = White noise.

<frequency>
    Frequency of the LFO.
    If >0 then the frequency is in Hz.
    If <0 then the frequency is in BARS: e.g. -1 means 1 BAR.

<delay> is the delay before the attack.
<atk> is the attack time. If >0 use exponential curve, else use linear curve.
<release> is the release time. If >0 use exponential curve, else use linear curve.

The instruction is declarative: the last values set are used at runtime in the layers.

BASEF is the base frequency for the oscillators, in hertz, used to calculate the default frequency of the oscillators.

Default 440 Hz. Limits: between 0.1 and 10000 Hz
It can be modified in the COMMON section with the **BASEF <number>** instruction.
E.g. BASEF 432.0

If set in a LAYER, it overwrites, for that LAYER, the global value.
If set to a negative value, the global value will be used.
If set to a positive value, the global value will be overwritten.

If the syntax **BASEF $<MCC>** is used, then the MIDI CC number <MCC> is sampled at trigger time and its value is used as BASEF. The scaling of the MIDI CC is the same of MCC2. No check is made on the final value. If an invalid <MCC> number is given, BASEF will be zero.

The instruction is declarative: the last value set is used at runtime in the layers.

KEYTRACK is the sensitivity of the frequency to key variations, used to calculate the default frequency of the oscillators.

Default 1.0 (100%). Limits: between 0 and 10.
It can be modified in the COMMON section with the **KEYTRACK <number>** instruction.
If set to less than 0 in the COMMON section, the default value will be used.
E.g. KEYTRACK 0.5

If set in a LAYER, it overwrites, for that LAYER, the global value.
If set to a value less than 0 in a LAYER, the global value will be used.
If set to a positive value, the global value will be overwritten.

If the syntax **KEYTRACK $<MCC>** is used, then the MIDI CC number <MCC> is sampled at trigger time and its value is used as KEYTRACK. The scaling of the MIDI CC is the same of MCC2. No check is made on the final value. If an invalid <MCC> number is given, KEYTRACK will be zero.

The instruction is declarative: the last value set is used at runtime in the layers.

KEYCENTER is the key corresponding to the BASEF frequency.

Default 69.0 (A3). Limits: between 0 and 127.
The BASEF can be modified to change the base tuning, e.g. to use the alternative 432 Hz tuning.
KEYCENTER should still be left to 69.0 (A3).
It accepts also fractional values to perform fine detuning.

It can be modified in the COMMON section with the **KEYCENTER <number>** instruction.
E.g. KEYCENTER 69.12345

If set in a LAYER, it overwrites, for that LAYER, the global value.
If set to a negative value, the global value will be used.
If set to a positive value, the global value will be overwritten.

If the syntax **KEYCENTER $<MCC>** is used, then the MIDI CC number <MCC> is sampled at trigger time and its value is used as KEYCENTER. The scaling of the MIDI CC is the same of MCC2. No check is made on the final value. If an invalid <MCC> number is given, KEYCENTER will be zero.

The instruction is declarative: the last value set is used at runtime in the layers.

FREQMOD <MCC1>,<MOD1>,...,<MCCN>,<MODN>

Global or local modulation of frequency by MIDI cc:
If the frequency is to be modulated in a layer by some MIDI CC and linearly in cents (perceptual), then the modulation can be put here for speed reasons.
All the FREQMODs specified in the COMMON section are added to the FREQMODs specified in a LAYER, without duplicates. LAYER's specification has priority.
If a local <MOD<i>>, after the merging, is 0, then the FREQMOD slot is deleted. This is for deleting a COMMON FREQMOD in a specific layer.
No check is made for duplicates in the same level (COMMON or LAYER <i>). The behavior is UNDEFINED.
The default is to not have any modulation.

<MCC1>...<MCCN> is the MIDI CC number of the modulator. See table above for the codes.

<MOD1>...<MODN> is the sensitivity in cents:

- if the MIDI CC goes from 0 to 1 (see MCC2 help below for the scaling: some MCCs are not scaled), the frequency is moved by MOD<i> (with sign) cents. 100 means a semitone.
- All the <MOD<i>> can be a real number, in this case capped between -24000 and 24000 cents, or the expression $<MCC>. In this case <MCC> is capped between 0 and 1000 and specifies a MIDI CC number that contains a value scaled as the MCC2 instruction prescribes. This value is continuously multiplied by the value of the MIDI CC number <MCC<i>>.
- Since the standard MIDI CC are always scaled to [0, 1[, at least one between <MCC<i>> and <MCC> should be a VST VAR with a proper range, otherwise the effect will be negligible.

The FREQ variable will be modified, after all other default automations are applied (e.g.: gliding, KEY number).
Polyphonic aftertouch (MIDI CC 132), KEYs, Velocities will be correctly applied per-LAYER.

The instruction is declarative: the order is not important. All the couples are merged as if there is one single instruction at the start of the COMMON section or LAYER.

FREQENV <amount>,<delay>,<atk>,<hold>,<decay>,<sustain>,<release>

Global or local multiplicative envelope for the FREQ.
If a mono envelope with constant or automated parameters for the frequency, with modulations in cents is sufficient, then the parameters can be put here for speed reasons.
Global default is all zeros, which means no modulation envelope. Local default is to use the global values.

All the parameters can be a real number, in this case capped between -10000 and 10000, or the expression $<MCC>, or the expression &<MCC>. In this last two cases <MCC> is capped between 0 and 1000 and specifies a MIDI CC number that must be sampled at trigger time ($) or continuously (&) and scaled as the MCC2 instruction prescribes.

<amount> is the peak pitch modulation for the envelope in cents. Can be negative: in this case the envelope is inverted.

<delay> is the delay before the attack.
<atk> is the attack time. If >0 use exponential curve, else use linear curve.
<hold> is the hold time between end attack and start decay
<decay> is the decay time. If >0 use exponential curve, else use linear curve.
<sustain> is the sustain level (1 = full 100%).
<release> is the release time. If >0 use exponential curve, else use linear curve.

The instruction is declarative: the last values set are used at runtime in the layers.

FREQLFO <num>,<param>,<freq>,<amount>,<delay>,<atk>,<release>

Global or local LFO for the FREQ.
If a multiplicative mono LFO with constant or automated parameters, zero starting phase (after the delay) and modulation in cents for the frequency is sufficient, then the parameters can be put here for speed reasons.
Global default is all zeros that means LFO disabled. Local default is to use the global values.

All the parameters can be a real number, in this case capped between -10000 and 10000, or the expression $<MCC>, or the expression &<MCC>. In this last two cases <MCC> is capped between 0 and 1000 and specifies a MIDI CC number that must be sampled at trigger time ($) or continuously (&) and scaled as the MCC2 instruction prescribes.

<amount> is the peak pitch detuning in cents

<num>,<param>: Base waveform.
 <num>:
  0 = Sine wave distorted with power <param> like the function POWABS. With <Param> near 0 the waveform approaches the square wave.
  1 = Square wave with PWM(%) = <param> * 100. If <param> <= 0, PWM is zero. If <param> >= 1, PWM is 100%.
  2 = Triangular wave with slope width given by <param>: <=0 and >=1 give sawtooth waveform and 0.5 triangular waveform. The other values give a sawtooth with unequal slopes.
  3 = Smooth sawtooth with formula (x/PI)-POWABS(x/PI,<param>). <param> should be >>1, but no check is made. The more <param>, the more the high frequency content.
  4 = Triangular distorted with power <param> like the function POWABS. With <Param> near 0 the waveform approaches the square wave.
  5 = White noise.

<frequency>
 Frequency of the LFO.

If >0 then the frequency is in Hz.
If <0 then the frequency is in BARS: e.g. -1 means 1 BAR.

<delay> is the delay before the attack.
<atk> is the attack time. If >0 use exponential curve, else use linear curve.
<release> is the release time. If >0 use exponential curve, else use linear curve.

The instruction is declarative: the last values set are used at runtime in the layers.

FCMOD <BASE_FREQ> <MCC1>,<MOD1>,...,<MCCN>,<MODN>

Global or local base value and modulation of the default filter cutoff frequency by MIDI cc.
If the base frequency for the cutoff is constant or derived from FREQ/FREQ0 and the cutoff frequency is to be modulated in a layer by some MIDI CC and linearly in cents (perceptual), then the modulation can be put here for speed reasons and you should use the FILT0 function.

All the FCMODs specified in the COMMON section are added to the FCMODs specified in a LAYER, without duplicates. LAYER's specification has priority.
If a local <MOD<i>>, after the merging, is 0, then the FCMOD slot is deleted. This is for deleting a COMMON FREQMOD in a specific layer.
If <BASE_FREQ> is below -999 in a layer, then the global value for <BASE_FREQ> will be used.
No check is made for duplicates in the same level (COMMON or LAYER <i>). The behavior is UNDEFINED.
The default is to have DEFAULTFC = FREQ and to not have any modulation.

<BASE_FREQ> is the starting frequency for the DEFAULTFC variable, which can be optionally modulated by some MIDI CCs. Default: 0, that means to use FREQ.

- If <BASE_FREQ> < -10 or <BASE_FREQ> = 0, then the base frequency for DEFAULTFC is FREQ.
- If <BASE_FREQ> > 0 and <BASE_FREQ> <= 10, then the base frequency for DEFAULTFC is FREQ * <BASE_FREQ>.
- If <BASE_FREQ> >= -10 and <BASE_FREQ> < 0, then the base frequency for DEFAULTFC is FREQ0 * ABS(<BASE_FREQ>), where FREQ0 is the value of FREQ before all the modulations, envelope and LFO, but after the gliding.
- If <BASE_FREQ> > 10, then the base frequency for DEFAULTFC is <BASE_FREQ> Hertz.

<MCC1>...<MCCN> is the number of the modulator. See table above for the codes.

<MOD1>...<MODN> is the sensitivity in cents:
if the MIDI CC goes from 0 to 1 (see MCC2 help below for the scaling: some MCCs are not scaled), the frequency is moved by MOD<i> (with sign) cents. 100 means a semitone.
All the <MOD<i>> can be a real number, in this case capped between -24000 and 24000 cents, or the expression $<MCC>. In this case <MCC> is capped between 0 and 1000 and specifies a MIDI CC number that contains a value scaled as the MCC2 instruction prescribes. This value is continuously multiplied by the value of the MIDI CC number <MCC<i>>.

Since the standard MIDI CC are always scaled to [0, 1[, at least one between <MCC<i>> and <MCC> should be a VST VAR with a proper range, otherwise the effect will be negligible.

The DEFAULTFC variable will be modified, starting from the <BASE_FREQ> frequency.

Polyphonic aftertouch (MIDI CC 132), KEYs, Velocities will be correctly applied per-LAYER.

The instruction is declarative: the last value for <BASE_FREQ> set is used at runtime in the layers and all the couples are merged as if there is one single instruction at the start of the COMMON section or LAYER.

FCENV <amount>,<delay>,<atk>,<hold>,<decay>,<sustain>,<release>

Global or local multiplicative envelope for the DEFAULTFC.
If a mono envelope with constant or automated parameters for the cutoff frequency, with modulations in cents is sufficient, then the parameters can be put here for speed reasons.
Global default is all zeros, which means no modulation envelope. Local default is to use the global values.

All the parameters can be a real number, in this case capped between -10000 and 10000, or the expression $<MCC>, or the expression &<MCC>. In this last two cases <MCC> is capped between 0 and 1000 and specifies a MIDI CC number that must be sampled at trigger time ($) or continuously (&) and scaled as the MCC2 instruction prescribes.

<amount> is the peak pitch modulation for the envelope in cents. Can be negative: in this case the envelope is inverted.

<delay> is the delay before the attack.
<atk> is the attack time. If >0 use exponential curve, else use linear curve.
<hold> is the hold time between end attack and start decay
<decay> is the decay time. If >0 use exponential curve, else use linear curve.
<sustain> is the sustain level (1 = full 100%).
<release> is the release time. If >0 use exponential curve, else use linear curve.

The instruction is declarative: the last values set are used at runtime in the layers.

FCLFO <num>,<param>,<freq>,<amount>,<delay>,<atk>,<release>

Global or local LFO for the DEFAULTFC.
If a multiplicative mono LFO with constant or automated parameters, zero starting phase (after the delay) and modulation in cents for the cutoff frequency is sufficient, then the parameters can be put here for speed reasons.
Global default is all zeros that means LFO disabled. Local default is to use the global values.

All the parameters can be a real number, in this case capped between -10000 and 10000, or the expression $<MCC>, or the expression &<MCC>. In this last two cases <MCC> is capped between 0 and 1000 and specifies a MIDI CC number that must be sampled at trigger time ($) or continuously (&) and scaled as the MCC2 instruction prescribes.

<amount> is the peak pitch detuning in cents.

<num>,<param>: Base waveform.
    <num>:
        0 = Sine wave distorted with power <param> like the function POWABS. With <Param> near 0 the waveform approaches the square wave.

1 = Square wave with PWM(%) = <param> * 100. If <param> <= 0, PWM is zero. If <param> >= 1, PWM is 100%.
2 = Triangular wave with slope width given by <param>: <=0 and >=1 give sawtooth waveform and 0.5 triangular waveform. The other values give a sawtooth with unequal slopes.
3 = Smooth sawtooth with formula (x/PI)-POWABS(x/PI,<param>). <param> should be >>1, but no check is made. The more <param>, the more the high frequency content.
4 = Triangular distorted with power <param> like the function POWABS. With <Param> near 0 the waveform approaches the square wave.
5 = White noise.

<frequency>
Frequency of the LFO.
If >0 then the frequency is in Hz.
If <0 then the frequency is in BARS: e.g. -1 means 1 BAR.

<delay> is the delay before the attack.
<atk> is the attack time. If >0 use exponential curve, else use linear curve.
<release> is the release time. If >0 use exponential curve, else use linear curve.

The instruction is declarative: the last values set are used at runtime in the layers.

PHASELFO <num>,<param>,<freq>,<amount>,<delay>,<atk>,<release>

Global or local LFO for the default phase of some oscillator variants (notably the fast ones).
If a mono frequency modulation with constant or automated parameters, zero starting phase (after the delay) and a linear modulation in radians or seconds for starting phase is sufficient, then the parameters can be put here for speed reasons.
Global default is all zeros that means LFO disabled. Local default is to use the global values.

All the parameters can be a real number, in this case capped between -10000 and 10000, or the expression $<MCC>, or the expression &<MCC>. In this last two cases <MCC> is capped between 0 and 1000 and specifies a MIDI CC number that must be sampled at trigger time ($) or continuously (&) and scaled as the MCC2 instruction prescribes.

<amount> is the peak dephasing in radians (synth oscillators) or seconds (sampled oscillators).

<num>,<param>: Base waveform.
<num>:
0 = Sine wave distorted with power <param> like the function POWABS. With <Param> near 0 the waveform approaches the square wave.
1 = Square wave with PWM(%) = <param> * 100. If <param> <= 0, PWM is zero. If <param> >= 1, PWM is 100%.
2 = Triangular wave with slope width given by <param>: <=0 and >=1 give sawtooth waveform and 0.5 triangular waveform. The other values give a sawtooth with unequal slopes.
3 = Smooth sawtooth with formula (x/PI)-POWABS(x/PI,<param>). <param> should be >>1, but no check is made. The more <param>, the more the high frequency content.

4 = Triangular distorted with power <param> like the function POWABS. With <Param> near 0 the waveform approaches the square wave.
5 = White noise.

<frequency>
Frequency of the LFO.
If >0 then the frequency is in Hz.
If <0 then the frequency is in BARS: e.g. -1 means 1 BAR.

<delay> is the delay before the attack.
<atk> is the attack time. If >0 use exponential curve, else use linear curve.
<release> is the release time. If >0 use exponential curve, else use linear curve.

The instruction is declarative: the last values set are used at runtime in the layers.

COMPLEXMOD <DEST>,<MCC1>,<MCC2>,<AMOUNT>

Global or local modulation of some parameters by a complex formula.
Each instruction adds a single complex modulation. You must issue separate instructions for each modulation.
Note that this modulation is constantly applied: it can be used to modify a modulation, like an *LFO or *ENV, that has trigger fixed parameters.

All the COMPLEXMODs specified in the COMMON section are added to the COMPLEXMODs specified in a LAYER, without duplicates. LAYER's specification has priority.

A COMPLEXMOD is equal to another if it differs only for the <AMOUNT> parameter.

No check is made for duplicates in the same level (COMMON or LAYER <i>). The behavior is UNDEFINED.

If a local <AMOUNT>, after the merging, is 0, for <DEST> <=2 or 1 for <DEST> >=3, then the COMPLEXMOD slot is deleted. This is for deleting a COMMON COMPLEXMOD in a specific layer.

The default is to not have any complex modulation.

<MCC1> is the MIDI CC number of the first modulation parameter. If it is less than zero, the first parameter is assumed to be 1, effectively disabling it. This can be used to modulate with a single MIDI CC.

<MCC2> is the MIDI CC number of the second modulation parameter. If it is less than zero, the second parameter is assumed to be 1, effectively disabling it. This can be used to modulate with a single MIDI CC.

<AMOUNT> is a real number capped between -10000 and 10000. It can be also the expression $<MCC3>. In this case the MIDI CC <MCC3>, scaled as the MCC2 instruction prescribes, is used as the value of <AMOUNT>. This means that up to 3 different MIDI CCs can be multiplied together.

Let <MOD> = <AMOUNT> * MCC2(<MCC1>) * MCC2(<MCC2>), if <AMOUNT> is a real number

OR

<MOD> = MCC2(<MCC3>) * MCC2(<MCC1>) * MCC2(<MCC2>), if <AMOUNT> is $<MCC3>

(See the definition of the MCC2 function below)

THEN

<DEST> specifies the destination of the modulation <MOD>:

0, means that the default frequency is moved by <MOD> (with sign) cents.
1, means that the default gain is moved by <MOD> (with sign) dB.
2, means that the default cutoff frequency is moved by <MOD> (with sign) cents.
3, means that the default frequency ENVELOPE is multiplied by <MOD>.
4, means that the default frequency LFO is multiplied by <MOD>.
5, means that the default frequency LFO and ENVELOPE are multiplied by <MOD>.
6, means that the default gain ENVELOPE is multiplied by <MOD>.
7, means that the default gain LFO is multiplied by <MOD>.
8, means that the default gain LFO and ENVELOPE are multiplied by <MOD>.
9, means that the default cutoff frequency ENVELOPE is multiplied by <MOD>.
10, means that the default cutoff frequency LFO is multiplied by <MOD>.
11, means that the default cutoff frequency LFO and ENVELOPE are multiplied by <MOD>.
12, means that the default phase LFO is multiplied by <MOD>.

Polyphonic aftertouch (MIDI CC 132), KEYs, Velocities will be correctly applied per-LAYER.

The instruction is declarative: if a triplet <DEST>,<MCC1>,<MCC2> is repeated more than one time (in particular in a LAYER after the same specification in COMMON), the last specification of <AMOUNT> is used.

Other Instructions

<u>The HOLD construct:</u>

The syntax is **<u>HOLD <var> = <complex_expression></u>**.

This construct calculates the variable <var> once, for performance reasons or special effects.

On POST step the variable is calculated near the start of the audio processing or recalculated at each suspend/resume sequence (e.g. change in sample rate, starting or stopping the DAW simulation, etc.)

On LAYERs the variable is calculated ONLY at trigger time and retains that value until layer end. In particular:

Envelopes are stuck to zero and oscillators are stuck at the initial value.

The gliding will not work: if the variable FREQ is used in the <complex_expression>, the variable will not be glided.

Each new note, fresh or not, glided or not has a fresh trigger time, so the expression will be re-evaluated again once.

HOLD <var> = <expr> is faster than **<var> = SHOLD(<expr>, 0)**: use SHOLD only for variable sample and hold time or periodic resampling or post-release holding.

If used with the LAST construct, the correct order is **LAST HOLD <var> = <expr>**

LAST <instruction>

The <instruction> is saved in a temporary file to be imported in all the LAYERs (not the POST section) after all its instructions.

The temporary file contains all the instructions in the COMMON section that are prepended with the LAST keyword and in the same order.

They must not be consecutive but can be also sparse: only instructions prepended with LAST are skipped and saved in the file for later use.

The syntax checking and variable checking is not performed in that moment but in every LAYER as if the <instruction> is part of the LAYER itself.

This means that eventual variables must be present in the LAYER.

This can be used to put a single repeated instruction (or a bunch) in the COMMON section like in this example:

```
        SAMPLE 200,...
        SAMPLE 201,...
        ...
        SAMPLE <N>,...

        // Default PITCHMOD and AMPMOD: if in a LAYER are redefined, these instruction will
be ignored
        AMPMOD = 0
        PITCHMOD = 0

        // This instruction will be repeated at the end of each LAYER
        LAST OUT = OSCENVMOD(200,AMPMOD,PITCHMOD,.1,.30,0,.3)

        // First LAYER
        LAYER
        SAMPLEOFF 0
        AMPMOD = ...
        PITCHMOD = ...
        // Second LAYER
        LAYER
        SAMPLEOFF 1
        AMPMOD = ...
        PITCHMOD = ...
```

// And so ON...

Here after the PITCHMOD is inserted the OUT = OSCENVMOD(...) instruction in each LAYER and the SAMPLEOFF instruction transforms the 200 in 200, 201 etc...

Interaction with **HOLD** and **EXECIF**:

In case of concurrent use of **LAST**, **HOLD** and/or **EXECIF** the valid combinations are:

LAST EXECIF HOLD
LAST HOLD EXECIF
LAST EXECIF
LAST HOLD

FEND <instruction>

The <instruction> is saved in a temporary file to be imported in the main file after all its instructions.

The temporary file contains all the instructions in the COMMON section that are prepended with the FEND keyword and in the same order.

They must not be consecutive but can be also sparse: only instructions prepended with FEND are skipped and saved in the file for later use.

The syntax checking and variable checking is not performed in that moment but at the end of the file.

This can be used to put commands to assure a particular setting is not overwritten.

Examples:
To assure that debug is disabled, you can use FEND DEBUG 0 in Setting.ini
To assure that debug is enabled, you can use FEND DEBUG 1 in Setting.ini
To force a particular QUALITY, INTERFACE size, HIDEUI setting, UIMOD setting, SILTH setting or SETFONT setting.

Conditional execution:

LABEL AND IF ... GOTO/EXIT STATEMENTS

LABEL <label_name>

Must be on a single row.
It attaches the label <label_name> to the next EXPRESSION or IF ... GOTO statement in the first non-declarative following row.

Labels don't get "attached" to declarative instructions.

If after the last LABEL statement in the COMMON section there are not EXPRESSIONs or IF ... GOTO/EXIT statements the behavior is undefined.
To skip all instructions until the end of the layer use IF … EXIT.

<label_name> is a separate space than variable name, it's case insensitive and has similar constraint of the variable names: 31 characters max, alphanumeric, but also the first can be a number.

So also BASIC/FORTRAN-style numeric labels are allowed.
If there are two or more consecutive LABEL instructions, the label that gets attached is the last.

Label space is local to a LAYER or POST section. Labels in the COMMON section are inherited by POST and LAYERs.

An error is given if a label is redefined. So a COMMON label can't be redefined in POST or LAYERs.

IF ... GOTO/EXIT statement

This construct is used for conditional execution, sample by sample.
If trigger time selection of instructions is acceptable, there is the EXECIF construct that's faster. See below.

IF <op1> <comp> <op2> GOTO <label>
OR
IF <op1> <comp> <op2> EXIT

<op1> and <op2> can be a constant, an existing variable, a KEYWORD, an MCC(#)

No expressions are allowed. If you need to compare an expression, just assign it to a dummy variable. Since the MIDI channel is the one of the LAYER or the POST step, to use a different channel you should use MCC3 and so a dummy variable.

No AND, OR, NOT and parenthesis. Combine the IF ... GOTO/EXIT like an assembler would do.

<comp> can be =, >, <, >=, <=, <>, !=

If <label> does not exist, the instruction is ignored.

Since LABELs need a non-declarative instruction to be attached at, if you need to skip all instructions until end of the layer, you can use IF … EXIT, without the need to insert a dummy instruction for a LABEL to be attached at.

Instructions with a label are always executed: the optimization that tries to see if the result is needed is disabled for those instructions.

IF ... GOTO/EXIT (if valid) are always executed.

To avoid uninitialized variables, the optimization that tries to see if the result is needed is disabled also for most instructions before an IF ... GOTO/EXIT (the actual algorithm is complex).
If there are not active envelopes (because they are not calculated due to IF ... GOTO/EXIT), the behavior of the layer is as if there are not envelopes.

IMPORTANT NOTE: skipping or re-executing instructions with some memory (e.g. Delay, Filters etc.) does not work as intended. Skipping does not advance the state and re-executing them advance again the state. This because you are running one copy only of the instruction, with only one memory storage.

It can be used for conditional execution (e.g. driven by MCC or VST VARs) or loops of memoryless functions.
It can be used, for instance, to substitute two or more separated layers with triggers, just merging all in one layer with conditional execution and so sparing some resources.
This is particularly useful in polyphony constrained instruments: using one single layer can ensure that all the sounds are triggered.

EXECIF construct

EXECIF <mcc>,<min>,<max> <var> = <expression>
OR
EXECIF <minkey>,<maxkey>,<minvel>,<maxvel> <var> = <expression>
OR
EXECIF <minkey>,<maxkey>,<minvel>,<maxvel>,<mcc>,<min><,max> <var> = <expression>

Conditional execution of the expression **<var> = <expression>**, with the choice performed at trigger time.

The first syntax allows the execution of the expression if the MIDI CC number <mcc> has value, at trigger time, between <min> and <max>, extremes included. No check is made on parameters: if <max> is less than <min> the instruction will never be executed; if <mcc> is an invalid number, zero is compared against <min> and <max>. The MIDI channel used is the one of the LAYER or POST step.

The second syntax allows the execution of the expression if the key number and the velocity, at trigger time, are between <min> and <max>, extremes included. No check is made on parameters: if <max> is less than <min> the instruction will never be executed.

The third syntax is the combination of the first two: if all conditions are satisfied, the instruction will be executed.

The instructions are not executed if the conditions are not met: this means that if the unexecuted instruction is the one responsible for a value of a variable, that variable will bear an UNDEFINED value.

This construct is typically useful to conditionally add to a variable (e.g. the OUT variable) another expression or to conditionally overwrite a previously calculated value.

This is particularly useful in polyphony constrained instruments: coalescing multiple samples that are to be fired simultaneously, using one single layer can ensure that all the sounds are triggered.

Optionally the <var> = <expression> can be preceded by the **HOLD** keyword, with the usual meaning. **HOLD** and **EXECIF** can be also swapped.

Nested **EXECIF** are allowed, but the inner specification will overwrite the outers.
EXECIF is ignored in POST step.

MIDI CC numbers and Keywords

MIDI CC Numbers

Here follows a table with the numbering and meaning of all MIDI CC (standard and extended).

Some MIDI CCs are fixed at note trigger time, they are signaled with a "Y" in the "FIXED" column.

Some MIDI CCs are per LAYER, some per channel and some are global: they are signaled in the "CONTEXT" column.

ALTERNATE KEYWORD	MIDI CC #	FIXED	CONTEXT	NOTE
N/A	0-127	N	PER CHANNEL	Standard MIDI CCs. Range: 0-127. Higher precision values must be composed manually, e.g. Volume=(MCC(7)+MCC(39)/128)/128.
KEY/KEYI	128	Y	PER LAYER	Key pressed. 69 is A3. 0-127.
KEYF	129	Y	PER LAYER	Key pressed including temperament and MIDI step post-processing. The range is roughly 0-127, but can be slightly over.
ONVEL	130	Y	PER LAYER	Velocity ON, including MIDI post-processing step. The range is 0-127.
OFFVEL	131	Y	PER LAYER	Velocity OFF, including MIDI post-processing step. The range is 0-127. If unavailable, it's the same than velocity on.
POLYAFT	132	N	PER LAYER	Current polyphonic aftertouch for current layer, given the current key pressed. 0-127.
AFTERTOUCH	133	N	PER CHANNEL	Monophonic aftertouch. 0-127.
PROGRAM	134	N	PER CHANNEL	Program number. 0-127.
WHEEL/PBEND	135	N	PER CHANNEL	Wheel/Pitch bend. Automatically rescaled to [-1, +1[.
BPM	136	N	GLOBAL	Beats per minute.
NUM	137	N	GLOBAL	Time partiture numerator. E.g. 3 for ¾.
DEN	138	N	GLOBAL	Time partiture denominator. E.g. 4 for ¾.
RANDOM	139	Y	PER LAYER	Random number. 0-1. Fixed at trigger time. For continuously varying random numbers, use a noise oscillator or the RND function.
GAIN	140	N	PER LAYER	Default gain to use for normal oscillators, also available with the GAIN keyword: GAIN=BASEG*(ONVEL/127)^VELTRACK. BASEG and VELTRACK can be varied per layer (default: 1.0 and 2.0). All the GAINMOD, GAINENV, GAINLFO modulations are then applied.
FREQ	141	N	PER LAYER	Default oscillator frequency, also available with the FREQ keyword: FREQ=SEMI(BASEF,KEYTRACK*(KEYF-KEYCENTER))*<gliding_factor>. BASEF, KEYTRACK and KEYCENTER can be varied per layer (def: 440.0, 1.0, 69.0). This is one of the MIDI CC that is varied during gliding/portamento. All the FREQMOD, FREQENV, FREQLFO modulations are then applied.
OUT	142	N	GLOBAL/PER LAYER	Current value of OUT variable. OUT keyword is an alias and the one way to use it on the left of equal sign.
IN	143	N	GLOBAL	Current value of IN variable. IN keyword is an alias.
TEMPERAMENT	144	Y	GLOBAL	Current temperament for the current layer, sampled at trigger time.
SAMPLERATE SAMPLEFREQ	145	N	GLOBAL	Current sample rate. Useful in custom filters.
TIME	146	N	PER LAYER	Time variable: time in seconds from trigger time. In POST step the time is in seconds from the start of the song.
SENDS<i>	147-150	N	GLOBAL	SENDS<i> sum, only in POST step. Reads as zero into the LAYERs. Layers use SENDS<num>=<expr> to accumulate the signal.
DEFAULTFC	151	N	PER LAYER	Default cutoff frequency of the FILT0 function. Can be based on a fixed value or on FREQ, modulated or not. Then the FC modulations are applied. See FCMOD, FCENV, FCLFO keywords.
DURATION	152	N	PER LAYER	Note duration. Contains valid data at release and after. Until then it grows linearly from 0 at trigger time, so can't be used then. Useful to automate release time of envelopes in function of note duration. Can be used in FREQENV and GAINENV.
GAIN0	153	N	PER LAYER	Gain to use for normal oscillators, also available with the GAIN0 keyword: GAIN0=BASEG*(ONVEL/127)^VELTRACK. BASEG and VELTRACK can be varied per layer (default: 1.0 and 2.0). All the GAINMOD, GAINENV, GAINLFO modulations are **NOT** applied.
FREQ0	154	N	PER LAYER	Oscillator frequency, also available with the FREQ0 keyword: FREQ0=SEMI(BASEF,KEYTRACK*(KEYF-KEYCENTER))*<gliding_factor>. BASEF, KEYTRACK and KEYCENTER can be varied per layer (def: 440.0, 1.0, 69.0). This is one of the MIDI CC that is varied during gliding/portamento. All the FREQMOD, FREQENV, FREQLFO modulations are **NOT** applied.
BANK	155	N	PER CHANNEL	Full bank number composed by MIDI cc #0 and #32.
<NOT ASSIGNED>	156-159	N/A	N/A	Reserved for future use.
<USER DEFINED>	160-199	N/A	PER CHANNEL	Unassigned and available for user constants.
<POLYAFTi>	200-327	N	PER CHANNEL	Polyphonic aftertouch. 0-127. MCC2(200+KEYI) is the same as POLYAFT/127.
<KEYSWITCHi>	400-527	N	GLOBAL	Current value of KEYSWITCH associated with key 0-127.
<VSTVARi>	600-727	N	GLOBAL	Current value of VST variable 0-127.
OUT0 – OUT99	800-899	N	GLOBAL	Current value of the output #n+1.
IN0 – IN99	900-999	N	GLOBAL	Current value of the input #n+2.

MIDI CCs, KEYSWITCH values, VST var and system keywords are always mono expressions.

They can be accessed with alternate keywords or three special functions: MCC(#number) MCC2(<expression>) and MCC3(<expression>,<channel>) (see below).

For MCC2 and MCC3 some MIDI CCs are rescaled in the interval [0, 1[for use with MOD2 and CURVE to be able to emulate SoundFont behavior. In particular #0-135 and #200-327.

For MCC, MCC2 and Keywords, the channel used to pick the value is the default channel of the current LAYER or POST step.
For MCC3 the channel is available as parameter. For GLOBAL or PER LAYER MIDI CCs (see table above) the channel is ignored.

MIDI CC Keywords

Here follows a detailed description of the keywords defined for some MIDI CCs and system variables.

<u>**Keywords that can be used in LAYERs and in POST step:**</u>

<u>**PROGRAM**</u> keyword is the current MIDI program selected. 0 - 127. It is an integer. Continuously variable.

<u>**BANK**</u> keyword is the current MIDI bank selected. 0 - 16383. It is an integer. Continuously variable.

<u>**SAMPLERATE or SAMPLEFREQ**</u> keyword is the current sample rate. Useful in custom filters.

<u>**BPM, NUM, DEN**</u> keywords are the current Beats per minute and time signature specification. Continuously variable. Sampled at each sample block.

<u>**WHEEL or PBEND**</u> keyword is the current pitch bend. [-1, +1[. Floating point with 14 bit precision. Continuously variable.

<u>**TEMPERAMENT**</u> keyword is the temperament currently in effect at layer trigger time. Fixed at trigger time. Can be used in automations. Can be used for triggers, with the MCC number.

<u>**IN**</u> is the VST input signal (INPUT # 2). Stereo. See the sections above for details.

<u>**IN0 – IN99**</u> is the VST input signal (INPUT # n+2). Stereo. See the sections above for details.

IN0 and IN00 are aliases of IN. For n < 10 both version with and without heading 0 are accepted. The unused variables can be used like normal variables.

<u>**OUT and OUT0 – OUT99**</u> are symbols that have various meaning depending on the context.

OUT and OUT00 are aliases of OUT. For n < 10 both version with and without heading 0 are accepted. The unused variables can be used like normal variables.

<u>In an instrument file LAYER section:</u>
When used in the expression on the right of the equal sign, it contains the current value of the corresponding layer output sample, to be used for further processing.

If accessed at layer start, before any initialization, it contains the value 0.
When used on the left of the equal sign then the current value of the corresponding layer output sample is updated with the value of the expression on the right of the equal sign.

In an instrument file POST section:
When used in the expression on the right of the equal sign, it contains the current value of the corresponding final output sample, to be used for further processing.
If accessed at POST section start, before any initialization, it contains the sum of all the corresponding active layers output samples, and for the OUT, OUT0 and OUT00 also the INPUT #1 signal is added.
When used on the left of the equal sign then the current value of the corresponding final output sample is updated with the value of the expression on the right of the equal sign.

For the POST step to have any effect, the OUTnn variables should be assigned some expression or chain of expressions, function of the value of the OUTnn variable itself and optionally the INnn or the other OUTnn.

SENDS1, SENDS2, SENDS3, SENDS4 are symbols that have various meaning depending on the context.

They operate like the OUT variable, except that the values are lost after the POST step.

In an instrument file LAYER section:
When used in the expression on the right of the equal sign, it contains the current value of the layer SENDS<i> sample, to be used for further processing.
If accessed at layer start, before any initialization, it contains the value 0.
When used on the left of the equal sign then the current value of the layer SENDS<i> sample is updated with the value of the expression on the right of the equal sign.
The expression is correctly crossfaded before adding to the SENDS channel.

In an instrument file POST section:
When used in the expression on the right of the equal sign, it contains the current value of the final SENDS<i> sample, to be used for further processing.
If accessed at POST section start, before any initialization, it contains the sum of all the active layers SENDS<i> samples.
When used on the left of the equal sign then the current value of the final SENDS<i> sample is updated with the value of the expression on the right of the equal sign.

The SENDS<i> are for passing summed values to the final POST step.
To have any effect you should use them in the final OUTnn formulas.
If there are no LAYERs, the SENDS<i> values zero. They can be used as normal variables, but they have no other meaning.

EXAMPLE:

```
SAMPLE 1,3

POST
out=.1*SIGM(1000,SENDS1)

LAYER
out=GAIN*OSCG(…,1,...)
```

```
        SENDS1=out
```

Keywords that can be used ONLY in LAYERs:

GAIN is an extended MIDI CC that takes the default gain of the fast oscillator functions.

It can be accessed in expressions with the GAIN keyword if you want to calculate a more complex GAIN for use with the full oscillator function.
The actual formula used for calculating the value is GAIN = BASEG * (ONVEL / 127) ^ VELTRACK
Then the modulations, the envelope and the LFO are applied, eventually modulated with a COMPLEXMODulation (see below).

GAIN0 is an extended MIDI CC that takes a possible gain of the fast oscillator functions.

It can be accessed in expressions with the GAIN0 keyword if you want to calculate a more complex gain for use with the full oscillator function.
The actual formula used for calculating the value is GAIN0 = BASEG * (ONVEL / 127) ^ VELTRACK
The modulations, the envelope and the LFO are **NOT** applied.

FREQ is an extended MIDI CC that takes the default frequency of the fast oscillator functions.

It can be accessed in expressions with the FREQ keyword if you want to calculate a more complex FREQ for use with the full oscillator function.
The actual formula used for calculating the value is FREQ = SEMI(BASEF, KEYTRACK * (KEYF - KEYCENTER)), which is FREQ = BASEF * 2.0 ^ (KEYTRACK * (KEYF - KEYCENTER) / 12.0)
This is the steady state value, because during gliding or portamento, this value is varied between two values.
Then the modulations, the envelope and the LFO are applied, eventually modulated with a COMPLEXMODulation (see below).

FREQ0 is an extended MIDI CC that takes a possible frequency of the fast oscillator functions.

It can be accessed in expressions with the FREQ0 keyword if you want to calculate a more complex frequency for use with the full oscillator function.
The actual formula used for calculating the value is FREQ0 = SEMI(BASEF, KEYTRACK * (KEYF - KEYCENTER)), which is FREQ0 = BASEF * 2.0 ^ (KEYTRACK * (KEYF - KEYCENTER) / 12.0)
This is the steady state value, because during gliding or portamento, this value is varied between two values.
The modulations, the envelope and the LFO are **NOT** applied.

DEFAULTFC is an extended MIDI CC that takes the default cutoff frequency that will be employed by the FILT0 function, in Hz.

This is composed by a base frequency and some optional modulations, envelope and LFO (see FCMOD, FCENV, FCLFO and COMPLEXMOD below).

If you are satisfied with the modulations, then you can use the FILT0 function. Otherwise you can use DEFAULTFC as base or even calculate in a custom way the cutoff frequency and use the full FILT function.

KEY or KEYI keyword is the note number currently to be played in the layer. It is not corrected for temperament and is an integer value. 0 - 127. 69 is A3. Fixed at trigger time.

If there is some pitch modifications with fractional detuning, the nearest integer is given by this keyword.
The actual pitch played by the layer depends on the oscillator definitions. FREQ is based on the KEY pressed.

KEYF keyword is the actual current note to be played, corrected for temperament. 0 - 127. 69.0 is A3. Fractional part multiplied by 100 is the cents of detuning. Fixed at trigger time.

The actual pitch played by the layer depends on the oscillator definitions. FREQ is based on this value.

ONVEL keyword is the note ON velocity of the note that triggered this layer. 0 - 127. Can be fractional if there was some MIDI post-processing. Fixed at trigger time.

GAIN is based on this value. The actual gain depends of the oscillator definitions in the layer.

OFFVEL keyword is the note OFF velocity of the note that triggered this layer. 0 - 127. Can be fractional if there was some MIDI post-processing. Fixed at trigger time.

This value is not currently used. Can be used by the user to automate something in the release stage.

AFTERTOUCH keyword is the current monophonic aftertouch. 0 - 127. It is an integer. Continuously variable.

POLYAFT keyword is the current polyphonic aftertouch for the current note that triggered the layer. 0 - 127. It is an integer. Continuously variable.

RANDOM keyword is a random number between 0 and 1, calculated at trigger time.

DURATION keyword is the note duration. Contains valid data at release and after. Until then it grows linearly from 0 at trigger time, so can't be used for trigger time automation. Useful to automate release time of envelopes in function of note duration.

Functions

Oscillator Functions

OSCG("format string", <parameters>…)

General oscillator function.
The format string is a string of 0 to 7 characters, case sensitive.
The characters over the 7th are ignored.
The missing characters take the value from the corresponding character of the default string.
If a character is not a valid option, then the corresponding character of the default string is used.

The default string is “s1f0000”.

The format string defines the options for the various components of the oscillator and implies also the number and positions of the parameters, following the format string: first come the parameter of the first character, then for the second character and so on.
See below for the details.

The first character defines the oscillator type:

‘s’ means single SINUSOIDAL waveform, no parameters needed.

‘y’ means single SYNTH waveform, 2 parameters needed (<num>, <param>).
    The coding of <num> and <param> is the same of GAIN/FREQ/PHASELFO.
    <num> is mono and sampled at trigger time.
    <param> is stereo and can be continuous variable.

‘S’ means STEREO SLOTed oscillator, 1 parameter needed (slot).
    The slot is mono and sampled at trigger time.

‘L’ means STEREO SLOTed oscillator and configurable loops, 4 parameters needed (slot, loopstart, loopend, loops)
    The slot is mono and sampled at trigger time.
loopstart, loopend, loops are stereo and continuous variable and specify the start and end samples of the loop and the loop count (see SAMPLE instruction for details).

‘V’ means STEREO SLOTed oscillator and configurable loops, 6 parameters needed (slot, start, end, loopstart, loopend, loops)
    The slot is mono and sampled at trigger time.
Start, end, loopstart, loopend, loops are stereo and continuous variable and specify the start and end samples, the start and end samples of the loop and the loop count (see SAMPLE instruction for details).

‘O’ means single MONO SINUSOIDAL waveform, with only one phase, frequency and amplitude, no parameters needed.

‘m’ means single MONO SYNTH waveform, with only one phase, frequency and amplitude, 2 parameters needed (<num>, <param>).
    The coding of <num> and <param> is the same of GAIN/FREQ/PHASELFO.
    <num> is mono and sampled at trigger time.
    <param> is stereo and can be continuous variable.

‘M’ means MONO SLOTed oscillator (one phase, frequency and amplitude), 1 parameter needed (slot).
    The slot is mono and sampled at trigger time.

‘O’, ‘m’ and ‘M’ options are the faster mono oscillators. Only one phase is calculated for each sample and only one amplitude. The left values are used, also for the sampled data. If your sampled data are stereo, consider using the WIDEPAN prefix to made it mono. See above.

The second character defines the gain formula:

‘1’ means that the gain is fixed to 1, no parameters needed.

‘C’ means that the gain is given by <C>, 1 parameter needed (<C>).
    <C> is stereo and can be continuous variable.

‘g’ means that the gain is given by GAIN variable, no parameters needed.

‘k’ means that the gain is given by GAIN * <k>, 1 parameter needed (<k>).
    <k> is stereo and can be continuous variable.

‘d’ means that the gain is given by GAIN * 10 ^ (<dB> / 20), 1 parameter needed (<dB>).
    <dB> is stereo and can be continuous variable.

‘G’ means that the gain is given by GAIN0 variable, no parameters needed.

‘K’ means that the gain is given by GAIN0 * <k>, 1 parameter needed (<k>).
    <k> is stereo and can be continuous variable.

‘D’ means that the gain is given by GAIN0 * 10 ^ (<dB> / 20), 1 parameter needed (<dB>).
    <dB> is stereo and can be continuous variable.

The third character defines the frequency formula:

‘A’ means that the frequency is fixed to 440 Hz, no parameters needed.

‘v’ means that the frequency is given by <F>, 1 parameter needed (<F>).
    <F> is stereo and can be continuous variable.

‘f’ means that the frequency is given by FREQ variable, no parameters needed.

‘k’ means that the frequency is given by FREQ * <k>, 1 parameter needed (<k>).
    <k> is stereo and can be continuous variable.

‘c’ means that the frequency is given by FREQ * 2 ^ (<cents> / 1200), 1 parameter needed (<cents>).
    <cents> is stereo and can be continuous variable.

‘F’ means that the frequency is given by FREQ0 variable, no parameters needed.

‘K’ means that the frequency is given by FREQ0 * <k>, 1 parameter needed (<k>).
    <k> is stereo and can be continuous variable.

‘C’ means that the frequency is given by FREQ0 * 2 ^ (<cents> / 1200), 1 parameter needed (<cents>).
    <cents> is stereo and can be continuous variable.

If the actual frequency is negative, then a conversion from BARS to Hz is performed.

The 4th, 5th and 6th characters defines the dephasing formula:

The 4th character:

‘0’ means dephase = 0, 0 parameters needed

‘L’ means dephase = AUTOPHASE, no parameters needed
See above for the definition of AUTOPHASE.

The 5th character:

‘0’ means no operation on the dephase, no parameters needed

‘L’ means to add PHASELFO to the dephase, no parameters needed
See above for the definition of PHASELFO.

The 6th character:
‘0’ means no operation on the dephase, no parameters needed

‘P’ means to add <param> to the dephase, 1 parameter needed (param)
<param> is stereo and can be continuous variable.

The dephase is in radians for synth oscillators and in seconds for sampled oscillators.

The 7th character defines the optional envelope to use:

‘0’ means no envelope, no parameters needed

‘L’ means LFO type envelope: attack + release, no decay, sustain = 100%, 2 parameters needed (attack, release)
Attack and release are mono and can be continuous variable.

‘S’ means simple envelope: attack, decay, sustain, release, 4 parameters needed
Attack, decay, sustain and release are mono and can be continuous variable.

‘D’ means double decay envelope: attack, decay1, decay1_level, decay2, sustain, release, 6 parameters needed
Attack, decay, sustain and release are mono and can be continuous variable.

‘F’ means full envelope: option, delay, attack, hold, decay, sustain, release, 7 parameters needed
option is mono and sampled at trigger time.
Delay, attack, hold, decay, sustain and release are mono and can be continuous variable.

For details on the meaning of the parameters, see the help of the ENV functions (L0 and L3 are assumed 0, L1 is assumed 1 and L2 is the sustain value).

Examples:

Operator style frequency modulation.

You need phase modulation sinusoidal waves, of unitary amplitude:

OSCG(“s1f0000”): the frequency is FREQ Hz. Useful for earliest modulators.
OSCG(“s1f00P0”,<phase>): the frequency is FREQ Hz and the phase is <phase> radians.
OSCG(“s1k00P0”,<freqm>,<phase>): the frequency is <freqm> * FREQ Hz and the phase is <phase> radians.

OSCG("s1v00P0",<freq>,<phase>): the frequency is <freq> Hz (or -<freq> BARS if <freq> is < 0) and the phase is <phase> radians.

They are mono oscillators and the parameters can be mono and continuously variable.

The steps needed to create an operator style chain for a frequency modulated instrument are the following:

- Create the earlier operators with the given functions, eventually multiplying for an envelope, and/or an LFO.
- If the operator should have self-feedback, use the PREV function as follows:
  FOO = OSCG(..., <modulation_from_previous_stages> + <feedback> * PREV(FOO))
- If the operator should have other feedback, use the PREV function as follows:
  // earliest oscillator in the loop
  FOO = OSCG(..., <modulation_from_previous_stages> + <feedback> * PREV(BAR))
  …
  // latest oscillator in the loop
  BAR = OSCG(…,<modulation_from_previous_stages>)
- Then add eventual gain, LFOs and/or ENV.
- The last level operators should be summed with the needed gains, ENVs or LFOs. GAIN could be used if the same gain, ENV and LFO are adequate.
- Add autophase, PHASELFO or FREQ0 as needed.

SUPERSAW(<freq>,<phase>,<detune>,<mix>,<n>,<filter>)
SUPERSAW0(<detune>,<mix>)
SUPERSAW1(<detune>,<mix>,<n>)

Stereo SAW waveforms, with multiple harmonics.
Frequency of the main harmonic is <freq> Hz or -<freq> BARS if <freq> is negative. Can be stereo.
Phase of all harmonics is <phase> radians. Can be stereo.
<n> is the number of couple of harmonics, max 15. The total number of harmonics is 2|<n>|+1.
If <n> >=0 all the harmonics have AUTOPHASE active. If <n> <0 AUTOPHASE is inactive.
If <filter>=1 then the 6DB/oct high pass filter of FC=<freq> is activated.
If <detune> >=0 then the frequency of the harmonics are <freq>, <freq> * (1+- <detune>), <freq> * (1+- 3*<detune>), <freq> * (1+- 6*<detune>), <freq> * (1+- 9*<detune>), etc…
If <detune> <0 then the frequency of the harmonics are <freq>, <freq> * (1+- <detune>), <freq> * (1+- 2*<detune>), <freq> * (1+- 3*<detune>), <freq> * (1+- 4*<detune>), etc…
<detune> can be stereo.
The amplitude of the center frequency is always 1.
If <mix> >=0, then the amplitude of all harmonics is <mix>.
If <mix> <0, then the amplitude is decreasing with the dinstance from the main: <mix>, <mix>/2, <mix>/3, etc…
<mix> Can be stereo.
For SUPERSAW0 and SUPERSAW1 the filter is enabled, <phase> = 0, <freq> = FREQ and for SUPERSAW0 <n> =3.

SINC(<op1>)

Stereo SINC. The amplitude is 1 and the starting phase is always zero at trigger time.
The actual formula is SIN(<op1>*T)/(<op1>*T) with T growing linearly after trigger time and it is 2 * PI after 1 second.

The automation can be stereo and time varying.

WAVETABLE(<num>,<op1>)

At each sample, <op1>L/R is interpreted as a value in seconds and the interpolated value of the sample slot <num> at the given time is calculated.
Valid only for sampled data. Separate release disabled. If looped, only a single loop is rendered. It is advisable to use only with one shot data.

To use synth data or other sampled data type (e.g. looped, with separate release), use the normal oscillator functions with a tiny fixed frequency (e.g. 1e-30) but not zero and the phase parameter. For one shot samples this function is still a little bit faster.

GRAINSYNTH(<num>,<op1>,<op2>,<op3>)

Grain synthesis of sample <num>, grain size <op1>L/R (samples), crossfade <op2>L/R (samples), grain number <op3>L/R.
At trigger time the grain numbers are sampled and the first grain is played from start, with the crossfade at start and end of the grain.
At the end of the grain, the grain number is resampled and the process begins again.

WAVESCAN(<num>,<op1>,<op2>,<op3>,<op4>)

Wavescan of sample <num>, grain size <op1>L/R (samples), grain number <op2>L/R, frequency <op3>L/R (Hz), dephase <op4>L/R (samples).
At trigger time the grain numbers are sampled and the first grain is played starting from the initial dephase.
The grain is sampled at a speed given by the frequency, assuming that a whole grain is one period (and that there is zero crossing at start and end).
At the end of the grain, the grain number is resampled and the process begins again.
All the processing is stereo, so the grain phase and the grain numbers can be different.
The dephase is applied continuously and the grain number is resampled as soon as the combined dephasing and current phase crosses the grain border.

Envelope Functions

ENV(<option>,<L0>,<L1>,<L2>,<L3>,<H1>,<H2>,<H3>,<atk>,<dcy>,<rel>)

Linear/exponential envelope with custom levels and 3 hold times.
All parameters are mono and continuously automatable.
The envelope is a mono expression.

<option> is a value with the following meaning:

0: normal HAHDSHR envelope behavior.
1: looped HAHD envelope. The envelope triggers again at the end of decay stage.
2: trigger. Ignores note off and sustain is forced to zero.
1001 - 2000: beat. The envelope retriggers every (<option> - 1000) / 32 of beat (not quantized).
2001 - 3000: sync. The envelope retriggers every (<option> - 2000) / 32 of beat. Quantized and in sync with tempo if playing.
NOTE: on most DAWs sync mode Behaves like beat mode if you are not in playback or live mode: the VST detects if the clock is stopped and defaults to beat mode.

NOTE: if the attack is too slow, the retriggers cause the envelope to retrigger in the middle, with irregular effects.

<L0>: initial level.
<L1>: final attack level.
<L2>: decay level. Sustain level.
<L3>: final release level.
<H1>: initial hold time (seconds). Before attack. Minimum 0.001.
<H2>: L1 hold time (seconds). Before decay. Minimum 0.001.
<H3>: L2 hold time (seconds). Before release. Minimum 0.001.
<atk>: attack time (seconds). Minimum 0.001 in absolute value. If > 0, then use exponential curve. Else use linear curve (use absolute value as the time).
<dcy>: decay time (seconds). Minimum 0.001 in absolute value. If > 0, then use exponential curve. Else use linear curve (use absolute value as the time).
<rel>: release time (seconds). Minimum 0.001 in absolute value. If > 0, then use exponential curve. Else use linear curve (use absolute value as the time).

ENVCURVE(<option>,<index>,<H1>,<atk>,<H2>,<dcy>,<H3>,<rel>)

Custom envelope with 3 hold times.
All parameters are mono and continuously automatable.
The envelope is a mono expression.

<option> is a value with the following meaning:

0: normal HAHDSHR envelope behavior.
1: looped HAHD envelope. The envelope triggers again at the end of decay stage.
2: trigger. Ignores note off and sustain is forced to zero.
1001 - 2000: beat. The envelope retriggers every (<option> - 1000) / 32 of beat (not quantized).
2001 - 3000: sync. The envelope retriggers every (<option> - 2000) / 32 of beat. Quantized and in sync with tempo if playing.
NOTE: on most DAWs sync mode Behaves like beat mode if you are not in playback or live mode: the VST detects if the clock is stopped and defaults to beat mode.
NOTE: if the attack is too slow, the retriggers cause the envelope to retrigger in the middle, with irregular effects.

<index> is the CURVE slot in with a LUT is defined. Such LUT should contain at least the interval 0..3.

The LUT value at 0.0 is kept during Hold1 time.
The LUT input value is linearly swept from 0.0 to 1.0 during attack.
The LUT value at 1.0 is kept during Hold2 time.
The LUT input value is linearly swept from 1.0 to 2.0 during decay.
The LUT value at 2.0 is kept during Sustain and Hold3 time.
The LUT input value is linearly swept from 2.0 to 3.0 during release.

<H1>: initial hold time (seconds). Before attack. Minimum 0.001.
<atk>: attack time (seconds). Minimum 0.001.
<H2>: final attack value hold time (seconds). Before decay. Minimum 0.001.
<dcy>: decay time (seconds). Minimum 0.001.
<H3>: H3, sustain value hold time (seconds). Before release. Minimum 0.001.
<rel>: release time (seconds). Minimum 0.001.

ENV0(<atk>,<dcy>,<sustain>,<rel>)

Simple linear/exponential ADSR envelope.

All parameters are mono and continuously automatable.
The envelope is a mono expression.

<atk>: attack time (seconds). Minimum 0.001 in absolute value. If > 0, then use exponential curve. Else use linear curve (use absolute value as the time).
<dcy>: decay time (seconds). Minimum 0.001 in absolute value. If > 0, then use exponential curve. Else use linear curve (use absolute value as the time).
<sustain>: sustain level.
<rel>: release time (seconds). Minimum 0.001 in absolute value. If > 0, then use exponential curve. Else use linear curve (use absolute value as the time).

ENV1(<atk>,<dcy>,<sustain>,<rel>,<delay>,<hold>)

Simple linear/exponential ADSR envelope, with delay before attack and hold between attack and decay.
All parameters are mono and continuously automatable.
The envelope is a mono expression.

<atk>: attack time (seconds). Minimum 0.001 in absolute value. If > 0, then use exponential curve. Else use linear curve (use absolute value as the time).
<dcy>: decay time (seconds). Minimum 0.001 in absolute value. If > 0, then use exponential curve. Else use linear curve (use absolute value as the time).
<sustain>: sustain level.
<rel>: release time (seconds). Minimum 0.001 in absolute value. If > 0, then use exponential curve. Else use linear curve (use absolute value as the time).
<delay>: initial hold time (seconds). Before attack. Minimum 0.001.
<hold>: L1 hold time (seconds). Before decay. Minimum 0.001.

ENV2(<atk>,<dcy1>,<dcy1_level>,<dcy2>,<sustain>,<rel>)

Simple linear/exponential AD1D2SR envelope.
All parameters are mono and continuously automatable.
The envelope is a mono expression.

<atk>: attack time (seconds). Minimum 0.001 in absolute value. If > 0, then use exponential curve. Else use linear curve (use absolute value as the time).
<dcy1>: decay 1 time (seconds). Minimum 0.001 in absolute value. If > 0, then use exponential curve. Else use linear curve (use absolute value as the time).
<dcy1_level>: level at the end of decay 1 and the start of decay 2.
<dcy2>: decay 2 time (seconds). Minimum 0.001 in absolute value. If > 0, then use exponential curve. Else use linear curve (use absolute value as the time).
<sustain>: sustain level.
<rel>: release time (seconds). Minimum 0.001 in absolute value. If > 0, then use exponential curve. Else use linear curve (use absolute value as the time).

Filter Functions

FILT(<num>,<input>,<Fc>,<res>)

General filter, on sample <input>, cutoff frequency <Fc>, resonance <res> [0, 1].
Note:

> Resonance values near 1 (greater by 0.9) give very unstable filters, especially the high order ones).

Peak is about 1/(4(1-<res>)), so for <res> = .75 the peak is about 0dB.

The values of <Fc>, <res> are continuously sampled, can be stereo and are applied separately for both channels.

<num> is mono, integer, sampled at NOTE ON time and specifies the filter type:

-2:	6dB/oct HP (<res> ignored)
-1:	6dB/oct LP (<res> ignored)
0:	12dB/oct LP
1:	24dB/oct LP
2:	36dB/oct LP
3:	48dB/oct LP
4:	12dB/oct HP
5:	24dB/oct HP
6:	36dB/oct HP
7:	48dB/oct HP
8:	12dB/oct BP
9:	24dB/oct BP
10:	36dB/oct BP
11:	48dB/oct BP
12:	12dB/oct NOTCH
13:	24dB/oct NOTCH
14:	36dB/oct NOTCH
15:	48dB/oct NOTCH
16:	12dB/oct ALL PASS / DEPHASE (LP-HP)
17:	24dB/oct ALL PASS / DEPHASE (LP-HP)
18:	36dB/oct ALL PASS / DEPHASE (LP-HP)
19:	48dB/oct ALL PASS / DEPHASE (LP-HP)

FILT0(<num>,<input>,<res>)

General filter, on sample <input>, cutoff frequency DEFAULTFC (mono and automated as specified by FCMOD, FCENV and FCLFO), resonance <res> [0, 1].
Note:

Resonance values near 1 (greater by 0.9) give very unstable filters, especially the high order ones).
Peak is about 1/(4(1-<res>)), so for <res> = .75 the peak is about 0dB.

The value of <res> is continuously sampled, can be stereo and is applied separately for both channels.

<num> is mono, integer, sampled at NOTE ON time and specifies the filter type:

Same codes as FILT.

EQ3DB(<type>,<input>,<lowfreq>,<highfreq>,<lowgain>,<midgain>,<highgain>)

3 band equalizer, on sample <input>.

The signal is filtered with a low pass filter of frequency <lowfreq> Hz and an high pass filter of frequency <highfreq> Hz.

Then the low, mid and high frequency components of the signal are calculated.

The <type> specifies the order of the filters: 1 is a one pole (6Db/oct) filter, 2 a two pole filter (12Db/oct) with resonance 0.75, to have the maximum flatness. 3, 4 and 5 mean a 4, 6 and 8 pole filter, with resonance 0.75.

The low, mid and high components are then amplified by the respective gains, expressed in decibel and the final value is returned as the result of the function.

All the parameters are continuously automatable and are stereo, except <type> that is mono.

EQ1DB(<type>,<input>,<lowfreq>,<highfreq>,<gain>,<lowres>,<highres>)

1 band equalizer, on sample <input>.

The signal is filtered with a low pass filter of frequency <highfreq> Hz and an high pass filter of frequency <lowfreq> Hz.

This signal is then multiplied by (DB2LIN(<gain>) – 1) and then added to the input signal and the final value is returned as the result of the function.

The <type> specifies the order of the filters: 1 is a one pole (6Db/oct) filter, 2 a two pole filter (12Db/oct), 3, 4 and 5 mean a 4, 6 and 8 pole filter.

The resonance of the low frequency high pass filter is <lowres>.
The resonance of the high frequency low pass filter is <highres>.

All the parameters are continuously automatable and are stereo, except <type> that is mono.

Effect Functions

Here it is a family of delay functions.
They are linear interpolating delay, suitable for time varying effects.

If the delay is fixed and the rounding to one sample is acceptable (e.g. for a reverb) there are the non-interpolating functions that are faster.

Just prepend NI to the function name, e.g. NIDELAY, NIDELAYF etc...

There are predefined and fast REVERB functions that use non interpolating delays.

DELAY(<input>,<DELAY>)

Input ->----- DELAY -----> Output

Signal <input> delayed <DELAY> seconds.

<DELAY> can be stereo and is continuously sampled.

<DELAY> is in seconds or in BARS (if < 0, converted in seconds) and is clipped between 0 and 2097151 samples in this implementation.

DELAYF(<input>,<DELAY>,<FEEDBACK>)

```
                         |---------------------------> Output
Input ->---------->+---^->- DELAY ->----|
                     ^--<- FEEDBACK  -<-|
```

Signal <input> delayed <DELAY> seconds, with feedback signal multiplied by <FEEDBACK>.
<DELAY> and <FEEDBACK> can be stereo and are continuously sampled.

<DELAY> is in seconds or in BARS (if < 0, converted in seconds) and is clipped between 0 and 2097151 samples in this implementation.

<FEEDBACK> should be between -1 and 1.

DELAYFF(<input>,<DELAY>,<FEEDBACK>,<FC>)

```
                         |---------------------------> Output
Input ->---------->+---^->- DELAY ->----|
                     ^--<- FEEDBACK  -<-|
```

Signal <input> delayed <DELAY> seconds, with feedback signal multiplied by <FEEDBACK>.
and filtered with a 6dB/oct LP or HP filter of cutoff frequency |<FC>|.
<DELAY>, <FEEDBACK> and <FC> can be stereo and are continuously sampled.
<DELAY> is in seconds or in BARS (if < 0, converted in seconds) and is clipped between 0 and 2097151 samples in this implementation.
<FEEDBACK> should be between -1 and 1.
The sign of <FC> determines the type of the filter (> 0 Low Pass, < 0 High Pass).

DELAY2C(<input>,<DELAY>,<FEEDBACK>,<FEEDFORWARD>)

```
                          |-----FEEDFORWARD-----|
Input ->----------->+---^->- DELAY ->--|----->----+---> Output
                     ^--<- FEEDBACK--<-|
```

Signal <input> delayed <DELAY> seconds with feedback multiplied by <FEEDBACK> and with feedforward coefficient given by <FEEDFORWARD>.
<DELAY>, <FEEDBACK> and <FEEDFORWARD> can be stereo and are continuously sampled.
<DELAY> is in seconds or in BARS (if < 0, converted in seconds) and is clipped between 0 and 2097151 samples in this implementation.
<FEEDBACK> should be between -1 and 1.
<FEEDFORWARD> can be any value.

DELAYFF2(<input>,<DELAY>,<FEEDBACK>,<FC>)

```
Input ->----------->+----->- DELAY ->---|----------> Output
                     ^--<- FEEDBACK--<-|
```

Signal <input> delayed <DELAY> seconds, with feedback signal multiplied by <FEEDBACK>.
and filtered with a 6dB/oct LP or HP filter of cutoff frequency |<FC>|.
<DELAY>, <FEEDBACK> and <FC> can be stereo and are continuously sampled.

<DELAY> is in seconds or in BARS (if < 0, converted in seconds) and is clipped between 0 and 2097151 samples in this implementation.
<FEEDBACK> should be between -1 and 1.

The sign of <FC> determines the type of the filter (> 0 Low Pass, < 0 High Pass).
The difference with **DELAYFF** is the point where the output signal is taken.

DELAYFF3(<input>,<DELAY>,<FEEDLF>,<FEEDHF>,<FC>)

```
                              |----------------------------> Output
Input ->---------->+---^->- DELAY ->---|
                       ^--<- FEEDBACK  -<-|
```

Signal <input> delayed <DELAY> seconds with feedback multiplied by <FEEDLF> for low frequencies and <FEEDHF> for high frequencies, with <FC> as the cutoff of the LP filter that separates the high and low frequencies (6dB/oct).
<DELAY>, <FEEDLF>, <FEEDHF> and <FC> can be stereo and are continuously sampled.
<DELAY> is in seconds or in BARS (if < 0, converted in seconds) and is clipped between 0 and 2097151 samples in this implementation.
<FEEDLF> and <FEEDHF> should be between -1 and 1.

DELAYFF4(<input>,<DELAY>,<FEEDLF>,<FEEDHF>,<FC>)

```
Input ->---------->+------>- DELAY ->--|-------------------> Output
                       ^--<- FEEDBACK  -<-|
```

Signal <input> delayed <DELAY> seconds with feedback multiplied by <FEEDLF> for low frequencies and <FEEDHF> for high frequencies, with <FC> as the cutoff of the LP filter that separates the high and low frequencies (6dB/oct).
<DELAY>, <FEEDLF>, <FEEDHF> and <FC> can be stereo and are continuously sampled.
<DELAY> is in seconds or in BARS (if < 0, converted in seconds) and is clipped between 0 and 2097151 samples in this implementation.
<FEEDLF> and <FEEDHF> should be between -1 and 1.
The difference with **DELAYFF3** is the point where the output signal is taken.

DELAYFF4X(<input>,<DELAY>,<FEEDLF>,<FEEDHF>,<FC>)

```
Input ->----------->+------->- DELAY ->---|-----------> Output
                       ^--<X- FEEDBACK -<-|
```

Signal <input> delayed <DELAY> seconds with feedback multiplied by <FEEDLF> for low frequencies and <FEEDHF> for high frequencies, with <FC> as the cutoff of the LP filter that separates the high and low frequencies (6dB/oct).
Left and right channels of the feedback are exchanged to enhance the stereo effect.
<DELAY>, <FEEDLF>, <FEEDHF> and <FC> can be stereo and are continuously sampled.
<DELAY> is in seconds or in BARS (if < 0, converted in seconds) and is clipped between 0 and 2097151 samples in this implementation.
<FEEDLF> and <FEEDHF> should be between -1 and 1.

PPDELAY(<input>,<DELAY>,<FEEDLF>,<FEEDHF>,<FC>)

```
                                                 |-----------------------> Output Left
                                                 |
Input ->--MONO--->+-->- DELAY LEFT-^->- DELAY RIGHT  ->|-----> Output Right
                    ^--<---------------- FEEDBACK ----------------<-|
```

Ping pong delay with filtered feedback.
The signal <input> is converted to MONO and fed into a first delay of <DELAY>L seconds.
Here the Left output signal is taken.
This signal is fed into another delay of <DELAY>R seconds.
Then the Right output signal is taken.
This last one is filtered with an LP filter of 6dB/oct.
The LP and HP components are given the gain <FEEDLF> and <FEEDHF> respectively and then fed back as feedback. (only the LEFT value, since this signal is mono).
<FC> is the LF cutoff frequency. Also only the L component.
<DELAY>, <FEEDLF>, <FEEDHF> and <FC> are continuously sampled and <DELAY> can be stereo.
<DELAY> is in seconds or in BARS (if < 0, converted in seconds) and is clipped between 0 and 2097151 samples in this implementation.
<FEEDLF> and <FEEDHF> should be between -1 and 1.
If <DELAY> is mono, this filter is the classic ping pong delay, but varying left and right independently can give interesting effects.
The left channel has the first echo. Use WIDE(...,-1) to invert the channels (see below).

SNG(<input>,<thres>,<ABS FC>)

Stereo Noise Gate of input signal, with <thres> as threshold (stereo and continuously automatable).
The absolute value of the input signal is filtered with a low pass filter of <ABS FC> Hertz of cutoff frequency (stereo and continuously automatable).
Let’s call this signal <A>.
For each channel a multiplicative factor is calculated: if <A> is below <thres>, this factor is 0. If <A> is above or equal 2*<thres>, this factor is 1. For intermediate values this factor is between 0 and 1.
The <input> signal is multiplied by this factor (for each channel) and the result is the output of the function.
The <A> signal is capped to 2*<thres> so to start immediately the cutoff of the signal as soon as it falls under the threshold.

SNG2(<input>,<thres>,<atk>,<dcy>)

Stereo Noise Gate of input signal, with <thres> as threshold (stereo and continuously automatable).
The absolute value of the input signal is filtered with a low pass filter of variable cutoff frequency so to have the right ramp time, depending on the situation.
Let’s call this signal <A>.
The initial state is “attack”, so the ramp time is <atk> seconds. This state remains until <A> is below 2 * <thres>.
At this point the state is switched to “decay” and the ramp time is <dcy> seconds (<atk> and <dcy> are stereo and continuously automatable).
The state remains “decay” until <A> goes below <thres> or the absolute value of the <input> signal goes above <thres>: in this case the state is switched again to “attack” and the cycle restarts.
For each channel a multiplicative factor is calculated: if <A> is below <thres>, this factor is 0. If <A> is above or equal 2*<thres>, this factor is 1. For intermediate values this factor is between 0 and 1.
The <input> signal is multiplied by this factor (for each channel) and the result is the output of the function.
The <A> signal is capped to 2*<thres> so to start immediately the cutoff of the signal as soon as it falls under the threshold.

COMP(<input>,<thres>,<amount>,<atk>,<dcy>)

Stereo compression (or expansion) of input signal, with <thres> as threshold (stereo and continuously automatable).
The absolute value of the input signal is filtered with a low pass filter of variable cutoff frequency so to have the right ramp time, depending on the situation.
Let's call this signal <A>.
The initial state is "attack", so the ramp time is <atk> seconds. This state remains until <A> is above the absolute value of the input signal.
At this point the state is switched to "decay" and the ramp time is <dcy> seconds (<atk> and <dcy> are stereo and continuously automatable).
The state remains "decay" until <A> goes below the absolute value of the input signal.
For each channel a multiplicative factor is calculated: if <A> is below <thres>, this factor is 1. If <A> is above or equal <thres>, this factor is (<thres>+(<A>-<thres>)/<amount>)/<A>.

The <input> signal is multiplied by this factor (for each channel) and the result is the output of the function.
If <amount> >1 then the function acts as a compressor.
If <amount> <1 acts as an expansor.
No check is made for <amount> to be >0.

COMP2(<input>,<thres1>,<slope>,<thres2>,<amount>,<atk>,<dcy>)

Stereo bilateral compression (or expansion) of input signal, with <thres1> and <thres2> as thresholds (stereo and continuously automatable).
The absolute value of the input signal is filtered with a low pass filter of variable cutoff frequency so to have the right ramp time, depending on the situation.
Let's call this signal <A>.
The initial state is "attack", so the ramp time is <atk> seconds. This state remains until <A> is above the absolute value of the input signal.
At this point the state is switched to "decay" and the ramp time is <dcy> seconds (<atk> and <dcy> are stereo and continuously automatable).
The state remains "decay" until <A> goes below the absolute value of the input signal.
For each channel a multiplicative factor is calculated:

- if <A> is below <thres1>, this factor is max(0,(<thres1>+(<A>-<thres1>)*<slope>)/<A>).
- if <A> is between <thres1> and <thres2>, extremes included, this factor is 1.
- If <A> is above <thres2>, this factor is (<thres2>+(<A>-<thres2>)/<amount>)/<A>.

The factor is limited to 1000 (+60Db) and care is taken to avoid NaNs.
The <input> signal is multiplied by this factor (for each channel) and the result is the output of the function.
If <amount> >1 then the function acts as a compressor on high signal levels.
If <amount> <1 acts as an expansor.
No check is made for <amount> to be >0.
If <slope> >1 then the function acts as a compressor on low signal levels (soft/hard noise gate).
If <slope> <1 acts as an expansor.
No check is made for <slope> to be >0.
<thres1> should be greater than 0 and <thres2> greater than <thres1> but no check is made.

COMPNG(<input>,<thres>,<slope>,<atk>,<dcy>)

Stereo compression (or expansion) of low portion of input signal, with <thres> as threshold (stereo and continuously automatable).
The absolute value of the input signal is filtered with a low pass filter of variable cutoff frequency so to have the right ramp time, depending on the situation.
Let's call this signal <A>.
The initial state is "attack", so the ramp time is <atk> seconds. This state remains until <A> is above the absolute value of the input signal.
At this point the state is switched to "decay" and the ramp time is <dcy> seconds (<atk> and <dcy> are stereo and continuously automatable).
The state remains "decay" until <A> goes below the absolute value of the input signal.
For each channel a multiplicative factor is calculated:

- if <A> is below <thres>, this factor is max(0,(<thres>+(<A>-<thres>)*<slope>)/<A>).
- if <A> is above or equal <thres>, this factor is 1.

The factor is limited to 1000 (+60Db) and care is taken to avoid NaNs.
The <input> signal is multiplied by this factor (for each channel) and the result is the output of the function.
If <slope> >1 then the function acts as a compressor on low signal levels (soft/hard noise gate).
If <slope> <1 acts as an expansor.
No check is made for <slope> to be >0.

REVERB(<input>,<LPROOM>,<HPROOM>,<FC>,<STEREO>)

Reverb with 12 parallel DELAYs with feedback (equivalent to DELAYFF4X) and 6 ALLPASS filters (equivalent to DELAY2C) in series.

The DELAYs are noninterpolating and care is taken to use only prime numbers as delays.

The <input> signal is reverberated. Just add the scaled output of this function to the original signal.

The minimum delay is about 661 samples (the 121st prime number, about 13ms at 48KHz of sample rate) so if a greater delay is needed, to simulate a bigger room, just use the DELAY function before adding to the signal or use the REVERB1 function. The other 11 stages take the successive prime numbers.

The delay of the all pass filters are from the 41st (179) to the 46th (199) prime samples.
The feedback coefficient of the all pass filters is 0.618 (1/golden ratio) to have a true all pass. Use REVERB2 to change it to e.g. change the diffusion component gain.

<LPROOM> and <HPROOM> are the feedback coefficients for the low and high frequency portion of the signal. Should be between 0 and 1 but no check is made. If under 0.5 the reverb is negligible. If near 1 the reverb tends to infinite time.

<FC> is the cutoff frequency of the 6dB/oct filter used to separate the frequencies.

<STEREO> is the offset, in primes, applied to left and right delay on the DELAYFF4X filters. The value is subtracted to the left delay and added to the right delay. E.g. if <STEREO> is 1, the left DELAY uses from the 120th to the 131st prime and the right DELAY uses from the 122nd to the 133rd prime.

The parameters are mono and continuously automatable. <STEREO> is an integer.

For speed reasons no check is made to <STEREO>. Take care of its value so the prime offset remains always between 0 and 1027 or program crash can occur (a good range is -10, 10).

REVERB1(<input>,<LPROOM>,<HPROOM>,<FC>,<STEREO>,<DELAY_OFFSET>)

Reverb with 12 parallel DELAYs with feedback (equivalent to DELAYFF4X) and 6 ALLPASS filters (equivalent to DELAY2C) in series.

The <input> signal is reverberated. Just add the scaled output of this function to the original signal.

The DELAYs are noninterpolating and care is taken to use only prime numbers as delays.

The <input> signal is reverberated. Just add the scaled output of this function to the original signal.

The minimum delay is about 661 samples (the 121st prime number, about 13ms at 48KHz of sample rate) so if a greater or lesser delay is needed, to simulate a bigger or smaller room, just use the <DELAY_OFFSET> parameter: this offset is added to the prime index, e.g. if <DELAY_OFFSET> is -10, the prime from 111st to 122nd are used. The other 11 stages take the successive prime numbers.

The delay of the all pass filters are from the 41st (179) to the 46th (199) prime samples.
The feedback coefficient of the all pass filters is 0.618 (1/golden ratio) to have a true all pass. Use REVERB2 to change it to e.g. change the diffusion component gain.

<LPROOM> and <HPROOM> are the feedback coefficients for the low and high frequency portion of the signal. Should be between 0 and 1 but no check is made. If under 0.5 the reverb is negligible. If near 1 the reverb tends to infinite time.

<FC> is the cutoff frequency of the 6dB/oct filter used to separate the frequencies.

<STEREO> is the offset, in primes, applied to left and right delay on the DELAYFF4X filters. The value is subtracted to the left delay and added to the right delay. E.g. if <STEREO> is 1, the left DELAY uses from the 120th to the 131st prime and the right DELAY uses from the 122nd to the 133rd prime.

The parameters are mono and continuously automatable. <STEREO> and <DELAY_OFFSET> are integer.

For speed reasons no check is made to <STEREO> and <DELAY_OFFSET>. Take care of their values so the prime offset remains always between 0 and 1027 or program crash can occur (a good range is -10, 10 for <STEREO> and -100, +800 for <DELAY_OFFSET>).

REVERB2(<input>,<LPROOM>,<HPROOM>,<FC>,<STEREO>,<DELAY_OFFSET>,<G_ALL_PASS>)

Reverb with 12 parallel DELAYs with feedback (equivalent to DELAYFF4X) and 6 ALLPASS filters (equivalent to DELAY2C) in series.

The <input> signal is reverberated. Just add the scaled output of this function to the original signal.

The DELAYs are noninterpolating and care is taken to use only prime numbers as delays.

The <input> signal is reverberated. Just add the scaled output of this function to the original signal.

The minimum delay is about 661 samples (the 121st prime number, about 13ms at 48KHz of sample rate) so if a greater or lesser delay is needed, to simulate a bigger or smaller room, just use the <DELAY_OFFSET> parameter: this offset is added to the prime index, e.g. if <DELAY_OFFSET> is -10, the prime from 111st to 122nd are used. The other 11 stages take the successive prime numbers.

The delay of the all pass filters are from the 41st (179) to the 46th (199) prime samples.
The feedback coefficient of the all pass filters is <G_ALL_PASS> (use 0.618 (1/golden ratio) to have a true all pass). Use it to e.g. change the diffusion component gain. Should be between -1 and +1, but no check is made.

<LPROOM> and <HPROOM> are the feedback coefficients for the low and high frequency portion of the signal. Should be between 0 and 1 but no check is made. If under 0.5 the reverb is negligible. If near 1 the reverb tends to infinite time.

<FC> is the cutoff frequency of the 6dB/oct filter used to separate the frequencies.

<STEREO> is the offset, in primes, applied to left and right delay on the DELAYFF4X filters. The value is subtracted to the left delay and added to the right delay. E.g. if <STEREO> is 1, the left DELAY uses from the 120th to the 131st prime and the right DELAY uses from the 122nd to the 133rd prime.

The parameters are mono and continuously automatable. <STEREO> and <DELAY_OFFSET> are integer.

For speed reasons no check is made to <STEREO> and <DELAY_OFFSET>. Take care of their values so the prime offset remains always between 0 and 1027 or program crash can occur (a good range is -10, 10 for <STEREO> and -100, +800 for <DELAY_OFFSET>).

REVERB3(<input>,<LPROOM>,<HPROOM>,<FC>,<STEREO>,<FIRST_DELAY>,<NUM_DLY_FF>,<FIRST_ALL_PASS>,<NUM_ALL_PASS>,<G_ALL_PASS>)

Reverb with <NUM_DLY_FF> parallel DELAYs with feedback (equivalent to DELAYFF4X) and <NUM_ALL_PASS> ALLPASS filters (equivalent to DELAY2C) in series.

The <input> signal is reverberated. Just add the scaled output of this function to the original signal.

The DELAYs are noninterpolating and care is taken to use only prime numbers as delays.

The minimum delay depends on <FIRST_DELAY> (e.g. if it is 120, it is about 661 samples, the 121st prime number, about 13ms at 48KHz of sample rate) so if a greater or lesser delay is needed, to simulate a bigger or smaller room, just use the <FIRST_DELAY> parameter: this is the first prime index, e.g. if <FIRST_DELAY> is 111 and <NUM_DLY_FF> is 12, the prime from 111st to 122nd are used. The other 11 stages take the successive prime numbers.

The delay of the all pass filters depend on <FIRST_ALL_PASS> (e.g. if it is 40 and <NUM_ALL_PASS> is 6, then the delays go from the 41st (179) to the 46th (199) prime samples).

The feedback coefficient of the all pass filters is <G_ALL_PASS> (use 0.618 (1/golden ratio) to have a true all pass). Use it to e.g. change the diffusion component gain. Should be between -1 and +1, but no check is made.

<LPROOM> and <HPROOM> are the feedback coefficients for the low and high frequency portion of the signal. Should be between 0 and 1 but no check is made. If under 0.5 the reverb is negligible. If near 1 the reverb tends to infinite time.

<FC> is the cutoff frequency of the 6dB/oct filter used to separate the frequencies.

<STEREO> is the offset, in primes, applied to left and right delay on the DELAYFF4X filters. The value is subtracted to the left delay and added to the right delay. E.g. if <STEREO> is 1, <FIRST_DELAY> is 111 and <NUM_DLY_FF> is 12, the left DELAY uses from the 120th to the 131st prime and the right DELAY uses from the 122nd to the 133rd prime.

The parameters are mono and continuously automatable. <STEREO>, <FIRST_DELAY>, <NUM_DLY_FF>, <FIRST_ALL_PASS> and <NUM_ALL_PASS> are integers.

For speed reasons no check is made to <STEREO>, <FIRST_DELAY>, <NUM_DLY_FF>, <FIRST_ALL_PASS> and <NUM_ALL_PASS>. Take care of their values so the prime offset remains always between 0 and 1027 or program crash can occur (a good range is -10, 10 for <STEREO> and 10, +1000 for <FIRST_DELAY>).
<NUM_DLY_FF> must be max 32 and <NUM_DLY_FF> + <NUM_ALL_PASS> must be max 256 for this implementation.

REVERB0(<input>,<ROOMSIZE>)

Fast reverb with 12 parallel DELAYs with feedback (equivalent to DELAYFF4X) and 6 ALLPASS filters (equivalent to DELAY2C) in series.

The <input> signal is reverberated. Just add the scaled output of this function to the original signal.

The DELAYs are noninterpolating and care is taken to use only prime numbers as delays.

The minimum delay is 653 samples, the 119th prime number, about 13ms at 48KHz of sample rate. The other 11 stages take the successive prime numbers.

The delay of the all pass filters go from the 41st (179) to the 46th (199) prime in samples.

The feedback coefficient of the all pass filters is 0.618 (1/golden ratio) to have a true all pass.

<ROOMSIZE> is the feedback coefficient for the low frequency portion of the signal. The high frequency portion is not fed back. Should be between 0 and 1 but no check is made. If under 0.5 the reverb is negligible. If near 1 the reverb tends to infinite time.

The cutoff frequency of the 6dB/oct filter used to separate the frequencies is 15000 Hz.

The offset, in primes, applied to left and right delay on the DELAYFF4X filters is 1, so the left DELAY uses from the 119th to the 130th prime and the right DELAY uses from the 121st to the 132nd prime.

The <ROOMSIZE> parameter is mono and continuously automatable.

PSHIFT(<input>,<shift>)

Pitch shifter without tempo changing of the signal <input>.
Pitch shift <shift> semitones. Can be stereo and continuously variable.
If <shift> > 0 the pitch is raised. If <shift> < 0 the pitch is lowered.
Can be also used for real-time (e.g. live) pitch shift.
The block size and the latency is 20ms.
The algorithm is a time domain type with optimum overlap calculated to have minimum sum of square of differences around the block edges (on a range of 20ms).
The search is performed going backward up to 20ms.
A crossfade of 20ms (with Hann window) is performed.
The interpolation for the repitching is the same of the sampled oscillators and is configurable with the QUALITY instruction.

PSHIFT2(<input>,<shift>,<Size>,<Backward>,<Range>)

Pitch shifter without tempo changing of the signal <input>.
Pitch shift <shift> semitones. Can be stereo and continuously variable.
If <shift> > 0 the pitch is raised. If <shift> < 0 the pitch is lowered.
Can be also used for real-time (e.g. live) pitch shift.
The block size and the latency is <Size> ms.
The algorithm is a time domain type with optimum overlap calculated to have minimum sum of square of differences around the block edges (on a range of <Range> ms).
The search is performed going backward up to <Backward> ms.
A crossfade of <Range> ms (with Hann window) is performed.
<Size> is continuously sampled and is clipped between 256 and 8192 samples.
<Backward> is continuously sampled and is clipped between 100 and 6144 samples.
<Range> is continuously sampled and is clipped between 10 and 16383 - <Size> samples.
The interpolation for the repitching is the same of the sampled oscillators and is configurable with the QUALITY instruction.

Spatial Image Functions

MONO(<op1>)

Convert <op1> to mono, averaging left and right channels. No effect if <op1> is already mono.

LEFT(<op1>)

Extract left channel from <op1>. If <op1> is mono, then the same value is returned.

RIGHT(<op1>)

Extract right channel from <op1>. If <op1> is mono, then the same value is returned.

JOIN(<op1>,<op2>)

Joins <op1> and <op2>. Left channel is MONO(<op1>), right channel is MONO(<op2>).

WIDE(<op1>,<W>)

Stereo image manipulation. <W> is continuously sampled and will cause WIDE to result in:

    MONO(<op1>) if zero.
    No change in <op1> if 1.
    Channel swap if -1.

All intermediate values give an intermediate result. Values outside -1, 1 are accepted.

GAINS(<input>,<L>,<R>)

Give different left and right gains to <input>.
<L> is for left channel and <R> is for right.
Only left channels of <L> and <R> are used so take care they are mono expressions.

PAN(<input>,<pan>)

Give different left and right gains to <input>.
Pan is in percentage and can be fully automated.

-100 means all on left. 100 means all on right. 0 means no modify.

Stereo data is modified accordingly: going versus -100 and +100 they become gradually mono.
<pan> is clipped at [-100, +100].

Other Functions

MCC(<num>)

Used to access standard or extended MIDI CC, KEYSWITCH values, VST VARS, Poly aftertouch values.
<num> is mono and must be a numeric constant.
There is a slight smooth to avoid abrupt changes (not on discrete MIDICC like keyswitches, program, BPM, NUM and DEN or integer or beats VSTVARs).
The default MIDI channel is used to assess the MIDI CC value.
See above table for details on number assignment.

MCC2(<num>)

Similar to MCC: <num> is still mono, but can be automatable.
Moreover MCC 0-133 and polyphonic aftertouch (200-327) are already divided by 128 for use with MOD2.
Finally there is a slight smooth to avoid abrupt changes (not on discrete MIDICC like keyswitches, program, BPM, NUM and DEN or integer or beats VSTVARs).
The default MIDI channel is used to assess the MIDI CC value.
See above table for details on number assignment.

MCC3(<num>,<channel>)

Similar to MCC2, but <channel> (still mono, but automatable) selects the MIDI channel used to assess the MIDI CC value.
See above table for details on number assignment.

SHOLD(<op1>,<T>)

Sample and hold.
In the following points, the phrase "<op1> is sampled" means that the value of <op1> is put in a buffer that is output by the function. When the sampling is paused, the last value written in that buffer is output by the function.

- When <T> = 0 The value of <op1> is sampled at trigger time (at song or clock start for POST step).
- When <T> > 0 The value of <op1> is sampled at trigger time (at song or clock start for POST step) and every <T> beats.
- When <T> < 0 and not in release stage, the value of <op1> is sampled continuously.
- When <T> < 0 and in release stage, <op1> sampled is paused.
- UNDEFINED behavior when <T> < 0 if in the POST step.
- Warning: <T> is converted in samples and truncated. To be sure <T> < 0, use a large enough value, e.g. -1.

Applied to both channels: there are two buffers and the <T> control is independent for the two channels.
Can be used to disable or quantize a continuous automation e.g. for the ENVx: by default the automation is continuously sampled.
Can be used to sense a MIDI CC or VST VAR up until release and then hold the value, e.g. to implement the correct behavior on MPE instruments.
NOTE: if <T> is constant zero it's faster to use the HOLD construct.

CURVE(<index>,<x>)

Apply the LUT/CURVE number <index> to <x> with linear interpolation for custom distortions.

To create a periodic custom waveform, use OSCG with the positive sawtooth (<param> = 1) and create a LUT that contains at least the -1, +1 interval. E.g.
CURVE(42,OSCG("ykk",2,1,<frequency>,<phase>)).

<index> is stereo and automatable.
If <index> is out of bound or the curve slot is empty, CURVE will perform identity mapping.

Negative values up to -17 specify predefined curves:

-1	<result> = ABS(<x>)	// Absolute value
-2	<result> = <x>	// Linear
-3	<result> = <x> * ABS(<x>)	// Concave curve
-4	<result> = SIGN(<x>) * SQRT(ABS(<x>))	// Convex curve
-5	<result> = 0 if <x> <.5, 1 otherwise	// 0.5 Binarization
-6	<result> = 1 - <x>	// Inverse linear
-7	<result> = (1 - <x>) * ABS(1 - <x>)	// Inverse concave curve
-8	<result> = SIGN(1 - <x>) * SQRT(ABS(1 - <x>))	// Inverse convex curve
-9	<result> = 1 if <x> <.5, 0 otherwise	// Inverse 0.5 binarization
-10	<result> = 2 * <x> - 1	// Bipolar linear
-11	<result> = (2 * <x> - 1) * ABS(2 * <x> - 1)	// Bipolar concave curve
-12	<result> = SIGN(2*<x>-1) * SQRT(ABS(2*<x>-1))	// Bipolar convex curve
-13	<result> = -1 if <x> <.5, 1 otherwise	// Bipolar 0.5 binarization
-14	<result> = 1 - 2 * <x>	// Inverse bipolar linear
-15	<result> = (1 - 2 * <x>) * ABS(1 - 2 * <x>)	// Inverse bipolar concave curve
-16	<result> = SIGN(1-2*<x>) * SQRT(ABS(1-2*<x>))	// Inverse bipolar convex curve
-17	<result> = 1 if <x> <.5, -1 otherwise	// Inverse bipolar 0.5 binarization

MOD2(<op1>,<op2>,<curve1>,<curve2>,<amount>,<transf>)

Two signal modulator.

Where:
<op1>,<op2>: input signals. Can be stereo.

<curve1>,<curve2>: algorithm to use to transform <op1> and <op2>. Can be stereo and continuously automated. See CURVE above for the numeric codes.

<amount>: multiplicative factor of the modulation. Can be stereo and continuously automated.

<transf>: transfer function. Can be stereo and continuously automated. See CURVE above for the numeric codes.

This is a similar, but more general, function of the modulators in soundfont 2.04:

- First the two input signals are transformed with the <curve1> and <curve2> respectively.
- The results are multiplied together and further multiplied by <amount>.
- Finally the curve <transf> is applied to the result.

In short the operations performed are:
<result> = CURVE(<transf>, <amount> * CURVE(<curve1>, <op1>) * CURVE(<curve2>, <op2>))

PREV(<var>)

Used to access the final value of the variable at the preceding sample.
This formula retrieves the last value that was assigned to the variable at the previous sample.
It can be used also in the expression updating <variable_name> (see example 1).
Instructions after the last OUT = instruction of a layer are not executed and so do not count for PREV.

Example (1): updating a variable using the previous value
LAYER
…
FOO = <some_function>(PREV(FOO))
…

It is applicable only to variables. To use the previous value of some keyword (e.g. OUT) you must use a dummy variable.

Example (2): saving OUT
PREVOUT = PREV(FOO)
... processing of inputs and PREVOUT
FOO = final expression involving also PREVOUT or expression calculated from PREVOUT
OUT = FOO

Note that trigger fixed keywords, being trigger fixed, have the same value, so PREV is a waste of resources.
Note also that if the variable <variable_name> is already defined before the PREV call, then it's that value that will be returned by PREV: to use the previous sample, the <variable_name> must be assigned after the PREV call and before the last OUT assignment.

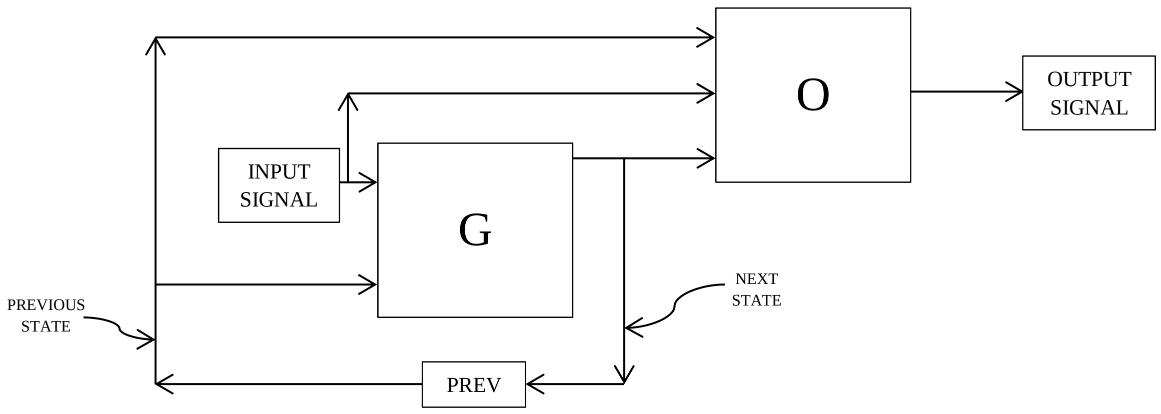
PERFORMANCE NOTE: if you need only the previous value, writing <var1>=PREV(<var2>) occupy an operation slot for almost nothing.
For performance reasons always use PREV(<var2>) in a more complex expression: you will have PREV(<var2>) for free.
Careful use of the PREV function and intermediate variables can be used to retrieve further early samples and so construct any FIR and IIR filter.

Example (3):
XNM = PREV (XNM1)
...
XN2 = PREV (XN1)
XN1 = PREV (INPUT)
INPUT = ... input signal // E.g. in a filter VST instance or in a POST step of an instrument VST file this can be just the OUT variable
...
YNM = PREV (YNM1)
...
YN2 = PREV (YN1)
YN1 = PREV (Y)

Y = any expressions involving INPUT, XNs and YNs   // This is the core of the IIR filter. If a FIR filter is to be designed, the YNs are not needed.

OUT = Y
NOTE: the order of the assignments is fundamental.

Example (4):
The PREV function can be used to implement the classical state machine depicted in the following figure:



LAYER

INPUT = ... expression calculating the input signal, e.g. OUT if it's a POST layer, an expression, an oscillator, a sample, etc...
PREVIOUS_STATE = PREV (NEXT_STATE)
...
NEXT_STATE = ... some function of INPUT and PREV_STATE (G (INPUT, PREVIOUS_STATE) in the picture above)
...

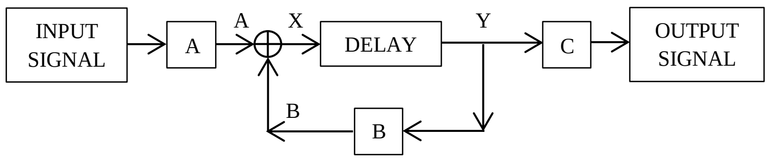
OUT = ... some function of INPUT, PREVIOUS_STATE and/or NEXT_STATE (O (INPUT, PREVIOUS_STATE, NEXT_STATE) in the picture above)

The example given is with a single state variable, a single input and a single output. More complex systems with more state variables require more variables, each for each state variable.

Each function can include also delays with feedback, non linearities etc...

The next state can be also modified with any function, before being used in the G function.
E.g. to implement the Extended Karplus-Strong (EKS) algorithm:

Example (5):



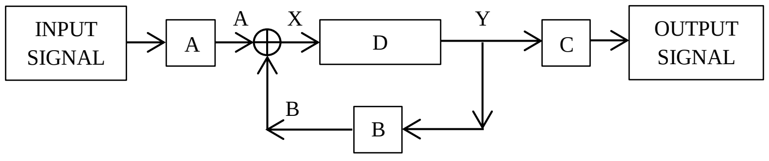
The implementation is the following:

INPUT = ... expression calculating the input signal, e.g. OUT if it's a POST layer, an expression, an oscillator, a sample, etc...
A = ... complex function of INPUT implementing the block labeled A in the picture above
PY = PREV(Y)
B = ... complex function of PY implementing the block labeled B in the picture above
X = A + B
Y = DELAY(X,(N-1)/SAMPLEFREQ)
OUT = ... complex function of Y implementing the block labeled C in the picture above

The "complex function" mentioned above can be an unlimited series of assignment, filters, distortions, using even delays and all the other functions available.

In the more general case, the DELAY can be substituted by any complex function "D" that derives Y from X:
A, B, C, D functions need not to be linear and can contain further delays or complex topologies.

Example (6):



The implementation is the following:

INPUT = ... expression calculating the input signal, e.g. OUT if it's a POST layer, an expression, an oscillator, a sample, etc...
A = ... complex function of INPUT implementing the block labeled A in the picture above
PY = PREV(Y)

B = ... complex function of PY implementing the block labeled B in the picture above
X = A + B
Y = ... complex function of X implementing the block labeled D in the picture above
OUT = ... complex function of Y implementing the block labeled C in the picture above

ABS(<op1>)

Calculates the absolute value of an expression. Applied to both channels.
Can be used for extracting pseudo amplitude information: just smooth the ABS of a signal with a slow filter and you can automate something (e.g. the cutoff frequency of another filter) with the amplitude of the signal.

SIGN(<op1>)

Calculates the sign of an expression. Applied to both channels.
Can be used for nonlinear symmetric distortions. E.g. even order distortion: Y = SIGN (X) * ABS(X) ^ 2

COPYSIGN(<op1>,<S>)

Copies the sign of <S> on <op1>. Applied to both channels.
Can be used for nonlinear symmetric distortions. E.g. even order distortion: Y = COPYSIGN (X ^ 2, X)

POWABS(<X>,<Y>)

Calculates SIGN(<X>) * ABS(<X>) ^ <Y>. Applied to both channels.
Can be used for nonlinear symmetric distortions. E.g. even order distortion: Y = POWABS (X, 2)

SAT(<op1>)

Saturation between 0 and 1. If <op1> is between 0 and 1 is left untouched. If it is below 0, evaluates to 0, if above 1 it evaluates to 1.
Applied to both channels.

SAT2(<op1>)

Saturation between -1 and 1. If <op1> is between -1 and 1 is left untouched. If it is below -1, evaluates to -1, if above 1 it evaluates to 1.
Applied to both channels.

ROUND(<op1>) / FLOOR(<op1>) / CEIL(<op1>)

Rounding of the input parameter. Useful to e.g. round the Pitch bend to perform a discrete glide.
Applied to both channels.

FRAC(<op1>)

Fractional part of the input parameter. Useful to e.g. make a periodic waveform with the TIME variable and an expression (or a WAVETABLE), to construct custom oscillators.
Applied to both channels.

ROUNDF(<op1>)

Rounding of the input frequency at the nearest semitone and with the current temperament applied.
Useful to perform a discrete glide.
Applied to both channels.
Uses the LAYER's BASEF, KEYCENTER and KEYTRACK for the rounding.

ROUNDF2(<op1>,<N>)

Rounding of the input frequency at the nearest <N> semitone (<N> should be integer, for semitone effect, but could be float) and with the current temperament applied.
Useful to perform a discrete glide.
Applied to both channels and stereophonically.
Uses the LAYER's BASEF, KEYCENTER and KEYTRACK for the rounding.

QUANTIZE(<op1>,<N>)

Rounding of the input signal modulus to <N> bits (<N> should be >=0 but no check is made).

Useful to simulate a low bit A/D converter.
Applied to both channels but <N> is mono.
E.g.: if <N> = 0 the signal is rounded to the nearest integer, so a in range value could be only -1 or 1 (2 values).
To simulate an N bit converter without saturation, use <N> = N – 1.
Use SAT2 to apply a saturation and so simulate a real A/D converter.

QUANTIZE2(<op1>,<N>)

It’s similar to QUANTIZE, but uses a slower formula that allows fractional values of <N> and moreover <N> can be stereo.

SIGM(<op1>,<op2>)

Sigmoid saturation. Evaluates to 2/(1+exp(-<op1> * <op2>))-1.
Applied to both channels.

SIGMDW(<op1>,<op2>,<DW>)

Sigmoid saturation with dry/wet control in <DW>: 0 original signal, 1 distorted signal. Even values outside 0, 1 can be used.
Applied to both channels. Dry/wet can be stereophonic.

LOG(<op1>)

Natural logarithm of <op1>. Applied to both channels.

EXP(<op1>)

e raised to <op1>. Applied to both channels.

SIN(<op1>) / COS(<op1>) / TAN(<op1>)

Trigonometric function of <op1>. Applied to both channels.

ASIN(<op1>) / ACOS(<op1>) / ATAN(<op1>)

Inverse trigonometric function of <op1>. Applied to both channels.

RND(<op1>)

Continuously variable random uniform noise of max amplitude <op1>. Stereophonic. <op1> can be continuously variable.

RNDN(<op1>)

Continuously variable random Gaussian noise of standard deviation <op1>. Stereophonic. <op1> can be continuously variable.

LIN2DB(<op1>)

<op1> converted to dB, i.e. 20*LOG10(<op1>). Applied to both channels.

DB2LIN(<op1>)

<op1> in dB converted to linear value, i.e. 10 ^ (<op1> / 20). Applied to both channels.

SEMI(<F>,<S>)

Modify the frequency <F> by the semitones in <S>, i.e. RESULT = <F> * 2 ^ (<S> / 12). Applied to both channels.

CENTS(<F>,<C>)

Modify the frequency <F> by the cents in <C>, i.e. RESULT = <F> * 2 ^ (<C> / 1200). Applied to both channels.

Appendixes:

Appendix A: MPE tutorial.

```
* Instrument file start.
* 1 Zone MPE instrument with Channel 0 as Master channel.

* Number of VST parameters:
VSTVARS 3

* Setting up HOLD and SOSTENUTO pedals:
* Channel 0 controls the VST VARs that control ALL the channels.
VSTVAR 0, 0, "HOLD", "", 0, 1, 4, "OFF", "ON"
VSTVAR 1, 0, "SOSTENUTO", "", 0, 1, 4, "OFF", "ON"
MIDICC 64, 0, 0
MIDICC 66, 0, 1
PEDAL 600
SOSTENUTO 601

* Setting up a simple piano like envelope.
VSTVAR 2,1,"VOLUME","",0,1,1
GAINENV &602,0,.1,0,10,0,.5

* Defining the expressions in the COMMON section so they are global

* MASTER Volume, product of the Master channel volume and
* the Member channel volume, sampled up until release stage.
MASTER = MCC3(7,0) * SHOLD(MCC2(7), -1) * GAIN

* Pitch bend, sum of the Master channel pitch bend and
* the Member channel pitch bend, sampled up until release stage,
* scaled for the right scale factor.
BEND = 2 * MCC3(135,0) + 48 * SHOLD(PBEND, -1)

* After touch, sum of the Master channel after touch and
* the Member channel after touch, sampled up until release stage.
AFT = MCC3(133,0) + SHOLD(AFTERTOUCH, -1)

* Timbre, sum of the Master channel timbre and
* the Member channel timbre, sampled up until release stage.
TIMBRE = MCC3(74,0) + SHOLD(MCC2(74), -1)

LAYER

* Simple SAW oscillator

* Frequency modulated by final pitch bend
F = Cents(FREQ, BEND)

* Here we assume channel pressure increases volume
G = MASTER + AFT

* Saw with high slope with Gain = G * GAIN and frequency = F
* AUTOPHASE on
O = OSCG("ykvL000",2,0.9,G,F)

* Filtering: 12dB/oct, 0.7 resonance LP filter.
* TIMBRE varies from 0 to 2, so Fc varies from 0 to 4 * F
OUT = FILT(0, O, 2 * F * TIMBRE, 0.7)
```