

Table of contents

Introduction.....	2
Overview.....	2
Processing Types.....	3
Installation and Setup.....	4
Initialization	7
User Interface.....	10
Instrument Files	16
Host Engagement Rules	27
Error Handling and Debugging.....	41
Performance Optimizations	43
Current implementation limits	44
Programming the Crescendo Engine.....	45
The COMMON Section.....	45
The POST Section.....	46
The LAYER Sections.....	48
Initializations Instructions	51
MIDI Instructions, stage I: Input/Output.....	106
MIDI Instructions, stage II: Post-processing.....	138
MIDI Instructions, stage III: Triggers, Group and Crossfades	171
Sequencer Instructions	183
Sample declaration and prefiltering Instructions	201
Defaults Instructions	217
MIDI CCs and keywords list	238
Oscillator Functions.....	253
Envelope Functions.....	278
Filter Functions.....	284
Advanced Signal Processing and Effects.....	296
Reverb Engine: Technical Overview.....	311
Chorus, Flanger and Phaser	324
Spatial Image Functions	335
Other Instructions and functions	337
FAQs and Tutorials	360
System and Installation	360
Programming and Logic	360
List all the randomization options in Crescendo	361
Audio Processing, Signal Flow and MIDI.....	363
Performance and Optimization	364
Synthesis and Tuning	365
Practical Recipes & Creative Solutions.....	365
Advanced Topic: Recursive Stereo Self-Modulation	388
Appendix A: Pictorial depiction of looping modes.	395

Introduction.

Overview

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

This language, detailed below, includes instructions, expressions, assignments and functions for manipulating audio and MIDI data, allowing for the creation of highly customized MIDI processors, instruments and effects.

The first sections of this document are an introductory overview of concepts, useful to setup and start to work with Crescendo. After that there is a thorough examination of the instructions and features, with extensive tutorials, how-tos and hello world examples, to show most of the features described. The final section includes a series of complete instrument files, commented and explained as a form of final tutorial.

Crescendo handles audio and MIDI data through a well-defined **multi-stage processing pipeline**, ensuring a structured flow of information and enabling users to control the sequence of operations. The pipeline consists of **MIDI Processing, Layer Processing and Post-Processing**.

Crescendo utilizes **VST variables**, also known as VST parameters, as user-controllable parameters that can be accessed and manipulated from the host DAW's interface or Crescendo's GUI. These variables allow for real-time control and automation of various aspects of the instrument or effect. They can be linked to MIDI CCs for external control, offering extensive flexibility in shaping sound and behavior.

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 provide access to parameters like pitch bend, program change, aftertouch, internal variables within the plugin, keyswitch values, and even VST variables.

Actually the MIDI pre-processing step, that includes a programmable sequencer, can generate, store or modify any type of extended MIDI CC messages, including VST parameters change.

There exist functions for accessing those MIDI CCs given the number or, for selected MIDI CCs, there are keywords for direct access.

In this document, when we use the term MIDI CC or MCC, if not specified, it is intended ALL the extended range. Details on the numbering and Keywords used are given in a following section.

Crescendo adopts a **layered architecture** for sound design, where individual layers function as independent processing units, each with its unique set of properties. By combining multiple layers, each with its own triggers and processing instructions, users can build complex instruments that emulate the behavior of acoustic instruments or create entirely new sonic textures.

The layer subdivision and other features, like crossfades, were included to make it easier for developers to port instruments from other plugins into Crescendo. However, because Crescendo

supports an unlimited number of oscillators, envelopes, and other elements per layer, it is not necessary to split layers. It is more efficient to sum components inside a single layer. Multiple layers should only be used if strictly necessary, for example, when splitting for notes, velocity ranges, program or bank number.

Assignments and expressions are essentially mathematical statements that can manipulate almost any parameter within the instrument, empowering users to automate parameters, calculate intermediate results, process audio signals, modify local MIDI CC storage, output the value of an expression on the MIDI Output, set the final output of a layer or the whole plugin, implement Conditional Logic.

Other features included in Crescendo:

Multi I/O: Crescendo supports multiple audio inputs and outputs, enabling sidechaining, routing to separate channels, multi-track routing, and more.

Multi Channel: Crescendo supports multiple MIDI channels, allowing for separate data streams and parameter values per channel.

Multi Instrument: Crescendo allows users to support multiple instruments in real time using MIDI program and bank numbers.

Temperaments: Crescendo supports various musical temperaments, including custom temperaments, those defined in SCALA and TUN files, and the host DAW's temperament.

Processing Types

The processing steps included in the processing pipeline are all optional. Depending on the active stages, the Crescendo VST plugin can function as different types of VST: **MIDI effect, audio effect, or instrument.**

MIDI Effects

- **Functionality:** A MIDI effect processes incoming MIDI data, modifying, filtering or transforming it **before it reaches an instrument or sound generator and/or producing new MIDI messages**. This includes altering note pitches, velocities, or timings, generating arpeggios or chords, quantizing note timings, routing MIDI messages to different channels, and more.
- The MIDI Effect step includes provisions to create new MIDI CC messages and/or Output them on the MIDI Output.
- The MIDI Effect step includes also a powerful programmable SEQUENCER, to create complex melodies starting from user input or even from scratch: these notes are also passed to the MIDI processing described above. The sequencer can create or modify also other types of MIDI messages, like MIDI CCs.
- **Instrument File Structure:** A MIDI effect instrument file typically consists of only a **COMMON** section. This contains instructions that specifically handle MIDI processing as detailed in a following section of this document.

Audio Effects

- **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 most instructions **are not used for audio processing within the POST section**: in the **COMMON** section generally there are configuration and initialization instructions. MIDI processing instructions may modify some MIDI CC that is used in the Post step, but for example note management instructions are not relevant for an audio effect.
 - The **POST** section is where the audio processing instructions are defined, operating on the audio signals on the inputs, eventually parametrized by MIDI CCs or VST parameters.

Instruments

- **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.

Installation and Setup

Crescendo is a programmable VST plugin, conforming to the version 2.4 of the VST specification. Currently only x86-32 and x86-64 Windows version is implemented.

It was tested in Windows 10 x64 edition, in Ableton Live 9.7.5 x64, LMMS 1.2.2 x64, Reaper 7.19 x64, VSTHost 1.57 x64, SaviHost x86 1.42 and SaviHost64 x64 1.42.

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 at least these three DLL files:

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

The package may also include:

- Some PDF manuals (E.g. this file).
 - Example instrument files (`.txt`).
 - Other dlls with Crescendo compiled for advanced instructions sets, like AVX or others.
- Note: these versions may not work on your CPU. The original ones are compiled without any special CPU instruction and will work on any x86-64 CPUs.

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. 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 the DLL files into the VST folder.**
3. **Restart your DAW** (if it was open during the copying process) or force the rescan of the plugin directory, so that it can recognize the new plugin.

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

Crescendo.dll, **Crescendo-effect.dll** and **Crescendo-MIDI.dll** are three 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 and the initial number of inputs and outputs.

The need for three 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.
- **Number of inputs and outputs fixed at first plugin loading** (e.g. Ableton).
- **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 and VSTHost, however, are more flexible and allow all 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. "Crescendo-MIDI.dll" is treated like "Crescendo.dll".
- **Reaper and VSTHost:** they are more flexible and treat all DLLs interchangeably, allowing them to be used as either instruments or effects. Moreover they support dynamic changing of number of inputs and outputs.
- **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 three files.

Functionality and Intended Roles

Despite the enforced distinction, the three files are identical in actual content, byte for byte; only the name is different and at plugin loading the name will determine how the plugin will identify itself to the Host.

If you have only one file, just copy it and rename all with the names given above, then copy them in the VST folder of your DAW.

- **Crescendo.dll** will identify itself as a VST instrument. Some DAWs will allow it only on MIDI tracks or similar.
- **Crescendo-effect.dll** will identify itself as an audio effect. Some DAWs will allow it only after an instrument or on an audio only track (e.g. audio, sends or master track).
- **Crescendo-MIDI.dll** will identify itself as a VST instrument with 0 audio inputs and outputs. Some DAWs will treat a 0 audio inputs and outputs as a pure MIDI effect.

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

- **MIDI Message Routing:** Some 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 applied to the input audio, but playing MIDI keys might not trigger any sound even if the instrument file has some layer.

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.
 - `Crescendo-MIDI.dll`: This version calls `isSynth(true)`, identifying itself as an instrument and set the default inputs and outputs number to 0. Some DAWs identify such a configuration as a pure MIDI effect. If your DAW supports dynamic change of the inputs and outputs, then an IO instruction can still let you use audio inputs and outputs.

Initialization

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 first loads in the DAW, 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 in the main Crescendo folder. If it is 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. This instruction is ignored in the first stage by the Crescendo-MIDI version. It is honored in the second stage, but if your DAW does not support dynamic changing of inputs and outputs number then the plugin remains stuck without inputs and outputs.
- **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:**
 - 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.
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.

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.

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.

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

Stage 2: Instrument File Loading

- 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.
- **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.

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.

The `settings.ini` Configuration File

Crescendo looks for a file named `settings.ini` in its root directory upon startup. This file acts as the "Global Brain" of the plugin, defining default UI behaviors, engine quality, and microtonal settings. The details of the instructions in this section are given in following chapters. Here we highlights some important things regarding initialization.

1. Global Engine Defaults

These parameters define how the audio engine and interface behave across all scripts:

- **VSTVARS** `<number>`: Sets the number of automatable parameters (0–128) visible to your DAW.
- **QUALITY** `<samples>; <ww>; <type>; <exponent>; <oversampling>`: Configures the windowed-sinc resampling engine. (Default: 4096, 7, 0, 1, 1).
- **INTERFACE** `<width>; <height>`: Defines the default canvas size for the plugin UI.
- **CHOKE** `<time>`: The CHOKE instruction provides a specialized mechanism for handling note re-triggers: it forces a brief, automated release phase for any active voices sharing the exact same MIDI key when a new Note On event for that key is received.
- **IO** `<numInputs>, <numOutputs>`: The IO instruction defines the number of physical stereo audio channels (inputs and outputs) the VST exposes to the DAW (Host).

2. Advanced Temperament Support

Crescendo is uniquely powerful for microtonal and historical music. The `settings.ini` allows you to define custom tuning systems using cent-deviations from Equal Temperament (12-TET).

The Syntax: `TEMPERAMENT "Name", C, C#, D, D#, E, F, F#, G, G#, A, A#, B`

The "Temperaments" Folder

Beyond the hard-coded "fancy" temperaments (like Werckmeister or Vallotti), you can now instruct the engine to index an entire external directory:

- **TEMPERAMENT** `"Temperaments"`: This special instruction tells Crescendo to look for a folder named `Temperaments` in the plugin directory. Any valid scale files found within will be added to the internal list automatically. You can add your custom directories with the same syntax. For details see the Temperaments section below.

3. UI Styling & Debugging

Customize the look and feel of the development environment:

- **SETFONT** `"<name>", size, weight, small_size, small_weight`: Sets the global typography for labels and tooltips.
- **UIMOD** `<theme>, enable`: Toggles visual themes and modern styling.
- **HIDEUI** `<level>`: Controls the visibility of default engine elements. Level 4 hides all standard UI components for a clean, custom look.
- **DEBUG** `<level>, <width>, <border>`: Sets the default behavior of the debug console.
 - Example: `DEBUG 0, 200, 50` initializes the log window with a 200px width and a 50px border, keeping it hidden (0) until called by a script.

Developer's Note: The Power of Defaults

By defining your `TEMPERAMENT` folder and `SETFONT` in the `settings.ini`, you ensure a consistent brand and musical feel across all your instruments. You don't need to re-type these definitions in every `.txt` file; Crescendo loads them globally, keeping your individual instrument scripts lean and focused.

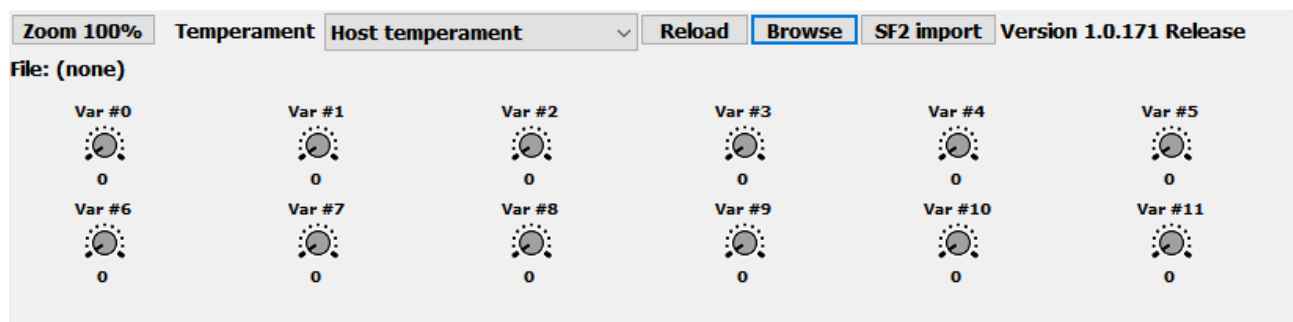
User Interface

The VST protocol supports plugin without a custom interface. The DAW will draw one under its control allowing the user to change VST parameters or loading and saving presets.

Crescendo **does** have a custom interface. This is called Editor in VST terminology. DAWs have usually a button or menu option to open the custom interface. E.g. Ableton has a little button with a wrench symbol, to open the editor of a VST plugin.

After including Crescendo in your DAW project, it is loaded and initialized. After that you can move the knobs or sliders in the DAW interface, but the plugin does not have an instrument file loaded yet, so you should open the Crescendo editor to begin with and load an instrument file.

This is the default appearance of the Crescendo plugin on a 96 dpi monitor, with 100% zoom factor:



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. The drop downs can be changed also with the mouse wheel or the cursor keys on the keyboard.
 - **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 and also allowing to see the current selected option.
- **Buttons:** Initiate specific actions within the plugin.
 - **Browse:** Opens a file dialog to select and load instrument files. During the loading process of a new instrument, the navigation button in the Browser Bar functions as a **STOP** button. This allows you to interrupt the loading sequence at any time, which is particularly useful when dealing with very large files.
 - **The STOP Function:**
 - **Responsiveness:** The system polls for the STOP command every **redraw1** milliseconds (default is 1000ms). Depending on your settings, there may be a slight delay between clicking the button and the actual interruption of the process. **Closing the editor is like clicking STOP.**
 - **Customization (UIMOD 5):** The frequency of this check is controlled by the first parameter of the **UIMOD 5** command, see below. By reducing the **redraw1** value, you can achieve a more responsive STOP button and more

frequent UI updates during loading, at the cost of slightly higher CPU overhead for the interface logic.

- **Post-Interrupt State:** To ensure system stability and immediate memory deallocation, clicking STOP does not revert to the previously loaded instrument. Instead, the engine performs a full reset: After a STOP command, the plugin will clear all samples from RAM and reload only the global defaults defined in the `settings.ini` file. This leaves the plugin in a "clean slate" state, ready for a new selection.
- **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 or just clicking it.
- **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:** Display the name and number of the currently loaded program and bank. This information can be hidden using the `HIDEUI` instruction (by default they are hidden).
 - **Time Signature and BPM:** Show 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` (by default they are hidden).
 - **Polyphony and POST step:** Show the current polyphony and the presence of a POST step. Like the program and bank information, these can be hidden using `HIDEUI` (by default they are hidden).
 - **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.
 - **Other labels:** The version text responds to mouse wheel, changing the current debug level. The file text shows the current loaded file and responds to mouse wheel, changing the current hidden UI elements.
- **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. If the instrument file contains some error, the debug log window is automatically activated, showing the errors.

UI Features and Customization Options

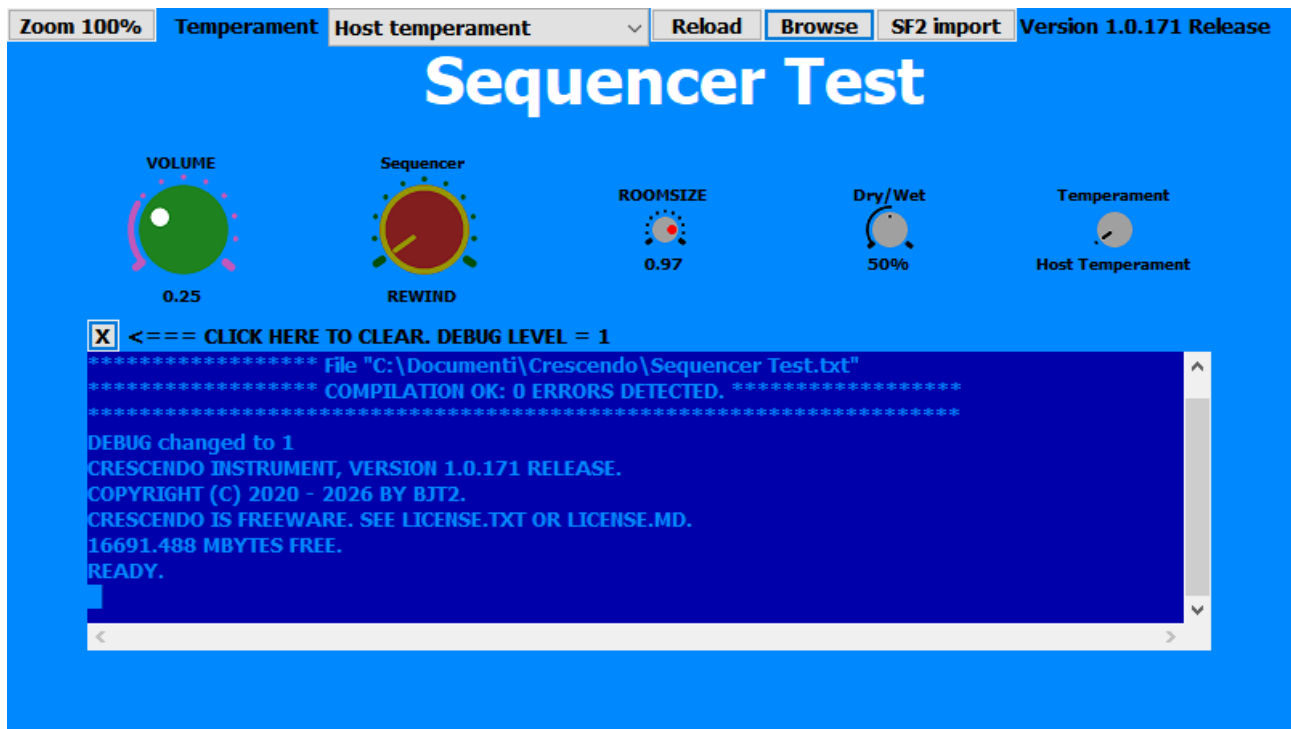
- **Textual VST Parameters:** The VST parameters can be of various types, depending on the `VSTVAR` configuration instruction. One of the types of a VST parameter is textual values, as an alternative to `KEYSWITCHES`. These have the Knob shape and behavior, but can be used as the `KEYSWITCHES` to select discrete options, without occupying a Note key.
- **Custom Tooltips for VST VARs / Parameters:** Allows the developer to better explain to the final user the meaning of a Knob.
- **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 if the display is 96 DPI and the Windows zoom level is set to 100%.
 - **Zoom Button:** Provides a manual way to adjust the overall scaling factor of the interface, independent of system DPI and zoom 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:
 - **Branding:** see the `BRANDING` section and instruction in the Initialization instructions section below.
 - **Skining:** You can change the diameter, border thickness, color and appearance of knobs and the window background color or image using the `UIMOD` instruction.
 - **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. The upper bar can be also hidden to have a cleaner interface, like the Simple Reverb example file, where window and knob skinning is shown:



This, instead, is what the editor looks like with the debug log window visible:



User Interaction: Keyboard and Mouse Gestures

- **Keyboard Navigation:**
 - **Tab:** Moves focus between interactive elements (knobs, buttons, and 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. Audio file should be dragged only in SAMPLEUI controls, otherwise they are tried to be opened as instrument files.

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, tooltips and VST var strings. However UTF-8 may not be supported in your DAW: 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.

SoundFont Import

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 the button labeled **"SF2 Bulk Import."** Or drag and dropping a SoundFont file into the Crescendo UI.

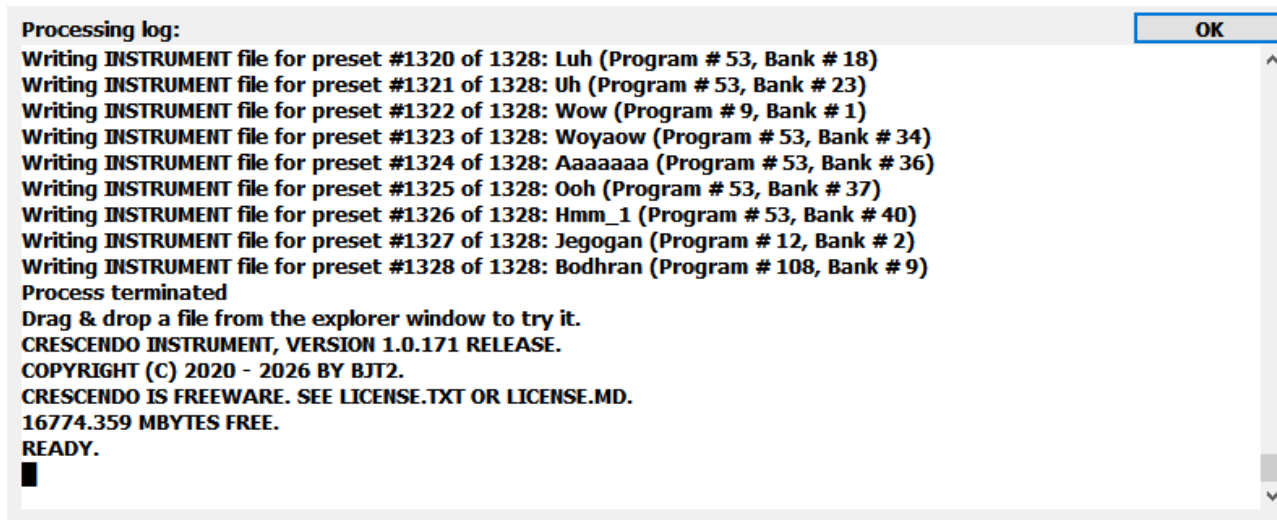
Import Process: The import process creates a series of files and directories to store the converted data:

- **"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. With SoundFont import only Settings.ini is loaded, so to change the `DEBUG` level you could include a `DEBUG` instruction there.

- **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.

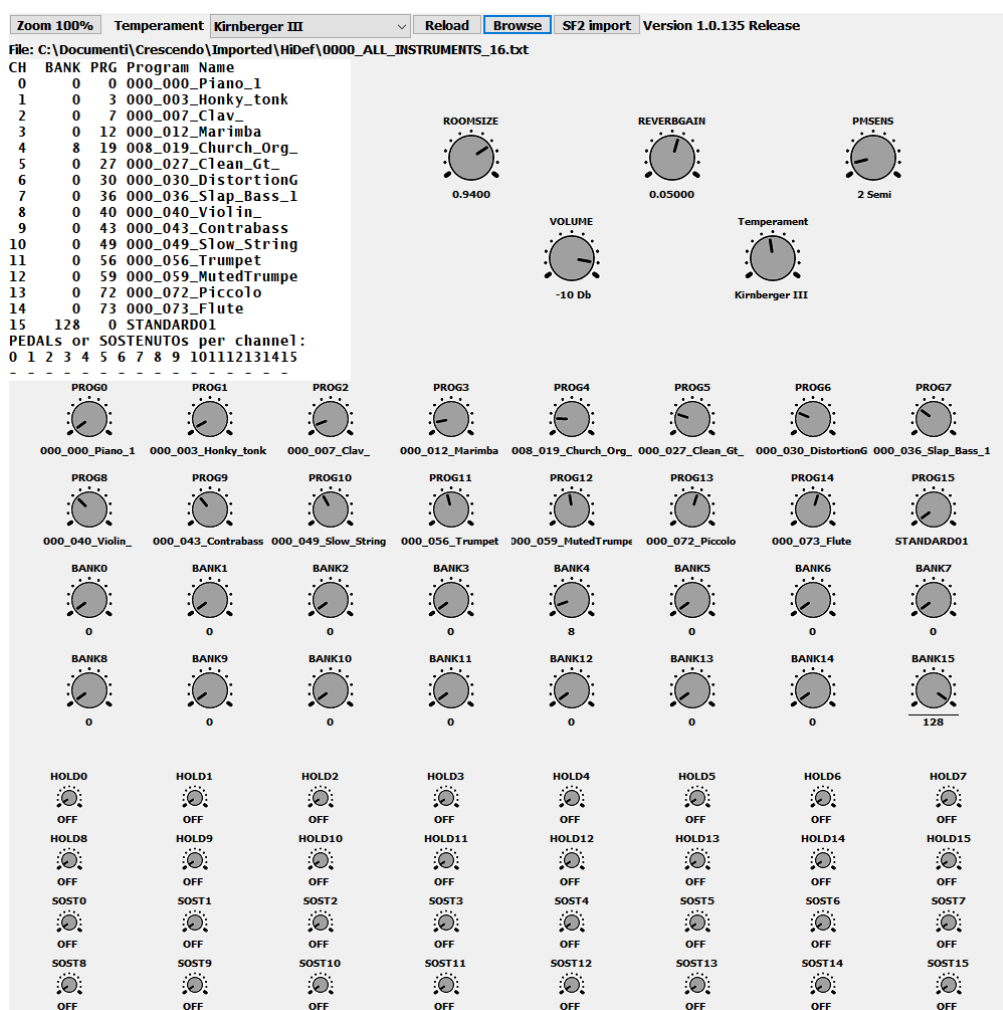
This is what the editor looks like after importing a Soundfont file with the default debug level:



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 or drag and dropping them.

After importing the file, Crescendo opens a Windows Explorer window pointing to the folder where it put the instrument files, encouraging the user to drag and drop one of them into the interface to try it. This will open the selected file, allowing the user to play with it.

This is an example of the interface of the 00000_ALL_INSTRUMENTS_16.txt with some knobs already personalized. The source SoundFont was a 4GB GM SoundFont, with 1300+ presets.



Importing a Soundfont file is a fast way to getting productive, but Crescendo is more powerful than this. Give a look to the tutorials interspersed below for examples of instrument files.

Instrument Files

Crescendo instrument files are plain text files, typically with a `.txt` extension, and utilize a structured programming language to define the behavior and characteristics of instruments and audio/MIDI effects.

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, user interface texts 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 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 compiler.
- **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 characters, like spaces and tabs, are handled flexibly in Crescendo instrument files. Leading and trailing spaces are ignored, allowing for indentation. Multiple consecutive spaces and tabs are treated as a single element. This flexibility promotes clean and readable code formatting.

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 (so this includes spaces) 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.

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. See below for details.

Key Syntax Elements

- **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.
- **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, any MIDI CC or Keyword or any output variable (`OUTnn` or `SENDS<i>`). The only exceptions are the `INxx` input variables: they cannot be modified. This instruction modifies the local storage only, being it a local variable or a VST parameter or a MIDI CC or the audio output placeholders. To output a value on the MIDI Output, use the `MIDIOUT` instruction.
- The `<expression>` can be any valid Crescendo expression. The expression is evaluated sample by sample (except when `HOLD` keyword is used) and assigned to the variable or MIDI CC.
- **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.

Variable Names

- **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 function names or `INxx` keywords 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.

Types of Variables

- **User Variables:** Standard user defined variables, for storing intermediate values, much like a conventional programming language.
- **Internal Variables:** Crescendo relies on numerous internal variables to track various aspects of the instrument's state and behavior. These include:
 - **VST Variables:** These are the 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.

- **MIDI CC Values:** These are accessed using the `VAR`, `MCC`, `MCC2` and `MCC3` functions or the `MCCnnnn` and `VARnnn` keywords, 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 and can be used on the right of an assignment to retrieve its current value.

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. The expression is the same in all Layers, but the value may be different based on the actual values of the items included.
- **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.

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, because 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, by not importing the assignment from the COMMON 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. They are generally calculated sample by sample.

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.
- **Keywords:** Reserved words with specific meanings and functionalities within the language. Examples include KEY, ONVEL, OUT, SENDS<i>, VARnnn, MCCnnnn.
- **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), VST Vars and MIDI CC access (VAR, MCC, MCC2 and MCC3), and various mathematical and transcendental operations.

Expressions can be used for a variety of purposes:

- **Automating Parameters:** By assigning an expression to a function parameter, you can dynamically control it 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 penalty. 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).
- Assignment of the value to a variable or another MIDI CC or outputting it on the MIDI Output.
- **Using Expressions in conditional Logic:** The IF...GOTO/EXIT construct supports any expression, including plain variables, keywords or a constant for the comparison.

Example Expressions and Assignments:

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. **MCC(144) = 0:** This assignment writes 0 in the Temperament MIDI CC.
4. **OUT = OSCG("Sgv00PS", 1, F, P, 0.1, 0.2, 0.8, 0.5):** This assigns the output of a slotted oscillator to the OUT variable, using specific parameters for frequency, phase, and envelope.

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.
- You can also modify in real-time almost any MIDI CC or VST parameter.
- The expression is evaluated for each sample (if not prefixed with the HOLD keyword).
- **Order of Operations:** Mathematical operators follow standard precedence rules. Use round parenthesis to control the order of evaluation. Other parenthesis are not supported.

More in details:

○ OPERATOR	Precedence	Associativity
○ ^	3 (Highest)	Right
○ *	2	Left
○ /	2	Left
○ +	1 (Lowest)	Left
○ -	1 (Lowest)	Left

- **Variable Scope:** The scope of variables is always local, meaning that each LAYER or POST has a different value for a variable with the same name. If a variable is defined in the COMMON section, the expression defining it is global, meaning that the expression is the same everywhere, but the value may differ.
- **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, including MIDI CC assignment. This can happen when a global expression, which is automatically imported in all layers and in the POST step, is not actually used. Also expressions after the last MIDI CC assignment (including to OUTnn keywords) are ignored, since they do not contribute to any output.

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 index of the temperament currently in effect.
- **IN**: Represents the audio signal coming from VST input #2 (stereo), often used for sidechaining.
- **INZ**: Represents the audio signal coming from VST input #1 (stereo), often used for sidechaining.
- **IN0-IN99**: Represent audio signals from additional VST inputs (#2 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 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.
- **Polyphonic aftertouch (200-327)**: Aftertouch for each key of the MIDI keyboard.
- **Keyswitch Values (400-527 or 1000-1007)**: 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.

Crescendo supports Program and Bank messages, with dedicated store for each channel. If you save the DAW project file, the last program and bank seen for each channel is saved too, so to restore a known state upon project reload.

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).
- The global expressions (defined in the COMMON section) 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

Given a MIDI CC number, the user can access its value in many ways:

- 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 `VAR` function: It retrieves the smoothed value of the specified VST VAR. It is an alternate way to access them, with both 0-127 and 600-727 numbering supported. This function is the fastest to retrieve a VST VAR value as usually does not consume any CPU time, since it is translated to a MIDI CC number in the operation using it.
 - **Syntax:** `VAR (<num>)`
 - **Parameter:** `<num>` is a numeric constant representing the VST VAR number.
- The `MCCnnnn` keywords: they retrieve the smoothed raw, unscaled value of the specified control parameter. These keywords are the fastest way 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:** `MCCnnnn`

- **Parameter:** `nnnn` is a numeric constant representing the MIDI CC number or control parameter identifier.
- The `VARnnn` keywords: they retrieve the smoothed value of the specified VST VAR. It is an alternate way to access them, with both 0-127 and 600-727 numbering supported. These keywords are the fastest way to retrieve a VST VAR value as usually does not consume any CPU time, since it is translated to a MIDI CC number in the operation using it.
 - **Syntax:** `VARnnn`
 - **Parameter:** `nnn` is a numeric constant representing the VST VAR number.
- 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.

The first four syntaxes can be used also on the left of the equal sign, to produce assignment instructions that modify in real-time almost any MIDI CC. The conversion to MONO is automatically performed if the MIDI CC is MONO (e.g. a VST VAR).

Note: assignments to MIDI CCs **DO NOT OUTPUT ANY MIDI CC message** on the MIDI Output, but modify only the internal storage as if a MIDI CC message were arrived from the MIDI Input. Use the **MIDIOUT** instruction for this purpose (see below).

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. Exceptions are the `OUTnn` that are treated differently.
They are: `KEY/KEYI`, `KEYF`, `ONVEL`, `OFFVEL`, `RANDOM`, `NOISE`, `GAIN`, `FREQ`, `TIME`, `DEFAULTFC`, `DURATION`, `GAIN0`, `FREQ0` and `OUTnn`.
- 2) **MIDI CC with different value per channel:** each channel has its value. `MCC`, `MCC2` and Keywords access the value of the default channel (see above for a definition of default channel). `MCC3` allows specifying 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, 1000-1007), `VST VARS` (600-727), `INZ`, `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, then it is this expression that is shared by the LAYERs and the POST step that use that variable, but the values 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), so the value would be different.

MIDI CC messages details

Crescendo has a very flexible MIDI CC processing engine. Here we outline its internals.

Definitions:

- [LOCAL] is a wide category of MIDI CC messages. It applies also to Note ON/OFF messages. It includes:
 - Every message generated through the Crescendo or DAW GUI, including user modifies with the mouse or keyboard or DAW automations to the VST VARs, and Temperament or Keyswitches combo boxes.
 - Every message generated programmatically, namely the assignments of the type `MCCnnnn/VARnnn/KEYWORD = <expression>`
The writes on MIDI CC number below 0, 128-132, 136-143, 145-154, 156-159, 328-599, 728-999, and above 1007 are ignored.
 - Every message generated to fulfill the MIDI CC linking feature (see `TEMPERAMENTCC`, `MIDICC` and `LINKCC`) even caused by a [REMOTE] message (see below).
 - Every MIDI CC message generated by the sequencer or the arpeggiator or the chord feature.
 - `LINKCC`, `MIDICC`, `KEYSWITCH`, `TEMPERAMENT`, and `TEMPERAMENTCC` at initialization: they generate a [LOCAL] message to update the related MIDI CC at instrument compiling.
- [REMOTE] includes all MIDI messages (MIDI CC and NOTE ON/OFF) coming from the MIDI Input, however they are generated. This includes also MIDI automations from the DAW, that are sent as MIDI CC messages and MIDI messages from MIDI files, MIDI clip in the DAW or keyboards or external controllers connected to the PC.
- [OUT] includes all messages potentially directed to the MIDI Output. It includes all messages that modify somehow the local MIDI CC storage or produce some notes, excluding the assignments that don't go on the MIDI Output. It includes also messages generated by the `MIDIOUT` instruction, which are intended for the MIDI Output only, namely they do not modify the local storage.
- Local storage is a term used for the internal storage space accessible with the `MCCnnnn/MCC (nnn) /VARnnn/VAR (nnn) /KEYWORD` syntax or the assignments.

Message types:

These are the types of messages that can be sent ([OUT] category) or received ([REMOTE] category) through the MIDI Input and Output pins:

- Standard MIDI CC messages: they are the standard MIDI CC messages, that are accessible in the following ranges:
 - 0-127: the standard MIDI CC messages (status 0xB0).
 - 133: the standard aftertouch/channel pressure MIDI CC message (status 0xD0).

- 134: the standard program change MIDI CC message (status 0xC0).
- 135: the standard pitch bend MIDI CC message (status 0xE0).
- 132 and 200-327: the standard polyphonic aftertouch MIDI CC message (status 0xA0).
- 1100-1227: the standard NOTE ON MIDI CC message (status 0x90).
- 1300-1427: the standard NOTE OFF MIDI CC message (status 0x80).
- RPN/NRPN messages: these are supported only in INPUT and so they are NEVER retransmitted in the MIDI Output. They are put in the RPN/NRPN local storage, accessible with the RPN and RPN2 instructions. If linked to a MIDI CC with the RPNLINK instruction, they are transformed in a [REMOTE] message and treated as such, including filtering. The RPN/NRPN feature can be completely disabled, causing the messages to be ignored, if any of the RPN/NRPN MIDI CCs (6, 38, 96-101) is disabled in the input/output enable feature (see MCCCTL below).
- Channel mode messages: these are supported only in INPUT and so they are NEVER retransmitted in the MIDI Output. They can be singularly disabled (see MCCCTL below).
- Extended MIDI CC. MIDI CC in the range 128-199 not included above, VST Vars (600-727) and Keyswitches (1000-1007) . They are transmitted and received, in full floating point with a series of standard MIDI CC messages in the range 102-108. If you have controllers that uses one of these MIDI CC, you should not use it with Crescendo or at least disable those messages. The Message number 108 signals the end of the train. 102 and 103 hold the MIDI CC number and 104-108 the floating point value (32 bits).
- Note ON and OFF messages. If coming from the MIDI Input they are classified as [REMOTE] and so subject to input filtering (see MCCCTL below). If coming from the sequencer they are classified as [LOCAL] and so they are always played. The temperament linking with a note key or the keyswitches require that the related Note ON messages are enabled (since they are of the [REMOTE] type) to work, otherwise only with the GUI or an enabled extended message or a local assignments they can be changed. TAPE Note ON messages cannot modify temperament or keyswitches even if they are in the range.

Details of the MIDI processing:

Here follows the detailed description of the treatment of the MIDI messages, from the MIDI Input (or from internal sources, like GUI, sequencer or assignments) to the MIDI Output.

- Input stage:
 - [LOCAL] messages are ALWAYS enabled, either modifying the local storage or producing some note. All [LOCAL] messages not coming from an assignment instruction are eligible to be output on the MIDI Output.
 - [REMOTE] messages modify the local storage or are played or are eligible for MIDI Output only if the related input bit (see MCCCTL below) is enabled and the sequencer is not in SEQ.ONLY mode.
- Processing stage. If the input message is enabled and the new value, after rounding and clipping, is different from the one in the local storage, the following processing actions are triggered:
 - Local storage update and GUI and DAW update. The DAW is informed if a VST VAR is modified so it can update its GUI and eventually disable its automation.
 - If the MIDI CC is linked to another MIDI CC (via TEMPERAMENTCC, MIDICC or LINKCC), then a new [LOCAL] message is generated recursively. MIDI CC loops are not detected, because a MIDI CC can be linked with only one other MIDI CC, so loops are impossible. See TEMPERAMENTCC, MIDICC and LINKCC.

- If the target MIDI CC is a VST Var, a check is performed to see if it is the sequencer MIDI CC or the smoothing frequency MIDI CC and these features are updated consequently.
- Check if it is associated to a hold or SOSTENUTO pedal and modify of the relative state.
- Check if a NOTE OFF trigger is associated with that MIDI CC and check of its value.
- Saving of the MIDI CC value as a new message in the TAPE if the sequencer is in MERGE mode.
- Output stage. The message can be sent on the MIDI Output. For standard messages a standard MIDI message is sent. For Extended messages the extended messages described above are sent. For RPN/NRPN messages linked with a MIDI CC a message is sent depending of the target MIDI CC number as if the message was a standard [REMOTE] message. A message is **NOT** sent on the MIDI Output if any of these conditions are satisfied:
 - If the message is [REMOTE] and the input bit is disabled (see MCCCTL below). As described above, all the local storage modify and check are not performed.
 - If the new value, after rounding and clipping, is equal to the one in the local storage. As described above, all the local storage modify and check are not performed.
 - If the relative output bit is disabled (see MCCCTL below), excluding sequencer messages and synthetic messages created with MIDIOUT: those messages bypass the output bit check.
 - If the sequencer is in SEQ.ONLY mode and the message does not come from the TAPE.
 - If the message come from an assignment instruction. To explicitly output a MIDI message (if the sequencer is not in SEQ.ONLY mode), use the instruction MIDIOUT (see below).

Overview of other MIDI related instructions:

There are some instructions that modify the MIDI preprocessing step. For more details, see below.

- RPNLINK. This instruction links an RPN/RPN message to a specific MIDI CC. The message is translated in a [REMOTE] message and treated consequently.
- TEMPERAMENTCC. This instruction links the TEMPERAMENT keyword, the MIDI CC number 144 and the temperament combo box to a specific MIDI CC. Each time one of them is modified, all the other are updated in real time.
- MIDICC. This instruction is used to initialize the local storage of a MIDI CC (excluding temperament and others) and optionally link it to a VST Var. If one of them is linked to some other object, the old link is broken before forming the new link. The VST Var is updated with a [LOCAL] message to the MIDI CC value, before establishing the new link.
- LINKCC. This instruction links any MIDI CC to any other MIDI CC (excluding the temperament, which is linked with TemperamentCC). If one of them is linked to some other object, the old link is broken before forming the new link. The second MIDI CC is updated with a [LOCAL] message to the first MIDI CC value, before establishing the new link.

How the messages are filtered:

There is an instruction, MCCCTL, which can enable or disable the MIDI Input for [REMOTE] messages and/or the MIDI Output for all messages excluding the sequencer and MIDIOUT ones (that are always enabled), MIDI CC for MIDI CC. See below for details.

Advanced modifies to the MIDI Note ON/OFF messages:

These processing steps are applied in order to MIDI Note ON and OFF messages.

- Arpeggio or Chord processing.
- Times, velocity and pitch processing.
- MIDIPOST program for advanced MIDI processing.
- Temperaments.

In conclusion, there are many options for MIDI processing. The more powerful are the assignments, the MIDIOUT and MIDIPOST instructions. See below for details.

RPN and NRPN messages:

Crescendo supports RPN and NRPN messages with a state machine for each of the 16 channels. RPN and NRPN messages are processed and the received values are stored in internal variables, per channel.

RPN and RPN2 allow accessing them with the default channel or a specific channel.

RPNLINK allows linking an RPN or NRPN message to a MIDI CC or VST variable: as soon as a linked RPN messages arrives, the value is translated in an equivalent [REMOTE] message and treated accordingly, e.g. filtered in input, etc.

Host Engagement Rules

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

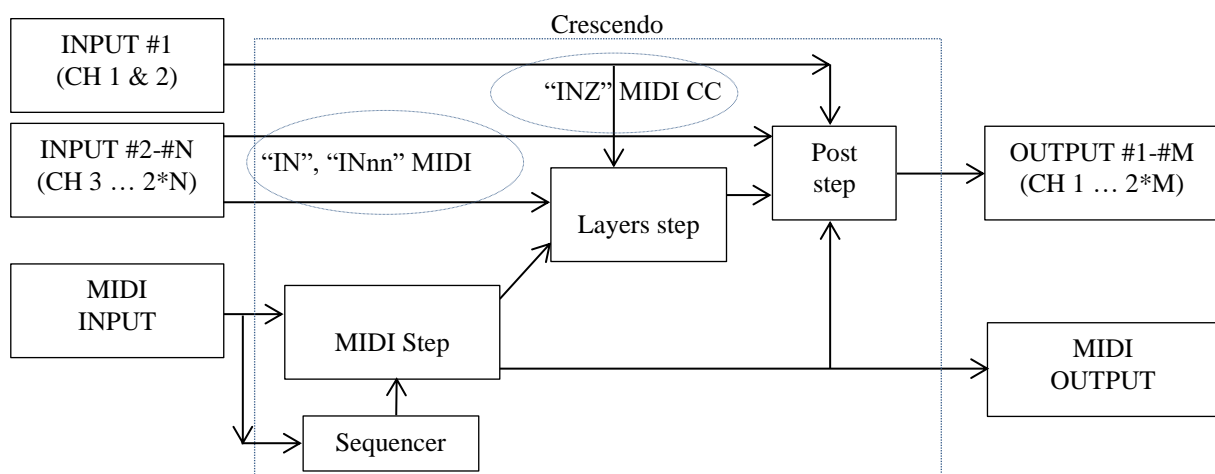
- **Audio and MIDI I/O:** The host DAW handles the audio and MIDI input and output for the plugin, routing signals to and from Crescendo. The host provides the audio stream that Crescendo processes and receives the processed audio back from the plugin. It also manages the MIDI messages that trigger notes 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.

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 for some DAWs as it determines how the host DAW handles the plugin's audio routing and MIDI connections.

- **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 allows to control VST parameters with a MIDI CC.
- **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.
- **Multi I/O (Multiple Input/Output):** 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. 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 channels can be filtered out or you can allow the triggering of multiple notes from different channels. The VST 2.4 standard allows a single MIDI Input and a single MIDI Output, leading to a limit of 16 channels in input and 16 in output.
- **Multi Instrument:** The MIDI standard includes three MIDI CCs, the program number and the bank number low and high, 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 per channel. The SoundFont import feature produces such a file: if the source SoundFont file contains more than one preset, then they can be assigned to different channels and played together.

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, available through the `INZ` variable. 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 first 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.
- **Crescendo-MIDI.dll:** This version defaults to zero inputs and zero outputs and so the use of those variables does not have effect. Also some DAWs (notably VSTHost), treats such a VST as a pure MIDI effect. Note that in all versions you can use the `IO` instruction to change the number of inputs and outputs and so you can emulate this behavior, but not all DAWs support dynamic changing the number of inputs and outputs. Notably Reaper and VSTHost do support dynamic changing and Ableton does not support dynamic changing. Check with your DAW if this feature is supported. Also please note that Ableton does not allow putting another instrument VST after any other VST in the chain: the first VST will receive the track MIDI input. Other VSTs on the track can receive MIDI data only with MIDI redirection (see below). VSTHost does not have this limitation.
- **MIDI INPUT** box represents MIDI messages from the host DAW, supporting up to 16 MIDI controllers. These messages can include note data, control changes, program changes, and other MIDI events. The messages can be filtered per channel or singularly, including single Note ON and OFF messages.
- **MIDI OUTPUT** box represents MIDI messages generated or modified by the plugin and sent to the host DAW, that eventually can route to other tracks or VSTs or other connected MIDI devices.
- **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 included in "MIDI CCs" as they can be accessed with the same functions used for the MIDI CCs. The VST parameter automations are NEVER filtered in input. They can be, and by default are, filtered in output.
- **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). It is included in "MIDI CC" as it can be accessed with the same functions used for the MIDI CCs. This control is NEVER filtered in input. It can be, and by default is, filtered in output.

- **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. They are included in "MIDI CCs" as they can be accessed with the same functions used for the MIDI CCs. These controls are NEVER filtered in input. They can be, and by default are, filtered in output.

All the MIDI CCs are accessible inside the instrument file to be modified, to be used in the expressions to calculate new values or to be outputted in the MIDI Output.

Inside the Crescendo box:

The diagram depicts the following signal processing:

1. **Input step:** Audio and MIDI signals enter the plugin through the respective inputs and are used by the various elements of Crescendo.
2. **MIDI step:** Incoming MIDI messages undergo processing, including filtering using the `MIDICH` instruction or the `MCCCTL` instruction and their MIDI CC values are stored in the local storage. New incoming MIDI CC messages may be created with the assignments or the sequencer, moreover note messages are potentially processed or created by the arpeggiator, chord generator or the sequencer, depending on the instrument's configuration. Processed MIDI note messages may undergo further modifications using instructions like `MAP`, `TIMING`, `PITCH`, `VELOCITY` and `MIDIPOST`. The selected temperament is also applied at this stage.
3. **Layers step:** Processed MIDI data triggers the activation of layers, each generating audio signals based on their defined parameters and processing.
4. **Post step:** 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.
5. **Output step:** 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 filtered and 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).

General Output Routing Concepts

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.

- **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** (equivalent to `INZ`, if connected). The **POST section** can further manipulate this combined signal. If you need only the layers sum, you can subtract `INZ` from `OUT` as a first step and then use the new `OUT` and `INZ` as you wish, for example as a sidechain. The automatic sum of `INZ` to `OUT` is performed so that an empty instrument file acts as a simple repeater of **INPUT #1**. Without this automatic routing an empty file will be silent independently from the input signals.
 - `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.

Ableton Live and Crescendo

Audio Tracks:

- **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:

- **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. Please note that a Crescendo-MIDI.dll is treated like a Crescendo.dll and will eat the MIDI input. You can only use it as a MIDI effect with MIDI redirection in another track (see below), because it will not have audio inputs and outputs.

Return Tracks:

- **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 9.7.5 supports VSTs with a maximum of 4 inputs and 16 outputs. This can restrict complex routing scenarios involving multiple audio sources and destinations. Also it does not support dynamic changing of number of inputs and outputs.
- The treatment of the inputs on MIDI and Audio tracks by Ableton are the main reason for the default configuration of Crescendo of 2 inputs and 1 outputs: with this setting it is always possible to perform sidechaining.

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 on the far right or down, depending on the view options. 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 below, 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.
- **Ableton's Input/Output Limits:** 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.

General MIDI Redirection Concepts

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. In particular, VSTHost, while not supporting natively temperaments, it does support preserving and sending detune values.

MIDI Redirection in Ableton Live 9.7.5

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.

- **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-plugin>

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/or preserves detune values, like VSTHost.

- **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 Sidechaining Concepts

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.

- **INZ and INxx keywords:** Crescendo uses specialized keywords, denoted as `INxx`, to represent the values of its various audio inputs. These keywords act as conduits for external audio signals routed from other tracks within the host DAW, enabling sidechain functionality.
- **Input Numbering:** The specific `INxx` keyword 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 keyword `IN` (or its aliases `IN00` or `IN0`) to access that signal. In some cases the `INPUT #1` is available for sidechaining. In these cases you access it using the `INZ` keyword.
- **Flexibility in Processing:** You have the flexibility to utilize `INxx` keywords in either the `LAYER` or `POST` sections of your Crescendo instrument file, giving you control over where sidechaining is applied in the signal flow.

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.

Sidechaining with Crescendo in Ableton Live 9.7.5

- **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. In particular, `INPUT #1` is available for sidechaining and you should use the `INZ` keyword to access it.
- **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 **Ableton's I/O Section** (see above) of your sidechain source track. 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 INx Variables:** In your Crescendo instrument file, use the corresponding `INxx` variable (or `INZ`) 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 (`INZ` or `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 amplitude of the `IN` variable to achieve a ring modulation effect.

For additional details, please consult the relevant Ableton KB page:

<https://help.ableton.com/hc/en-us/articles/209775325-Sidechaining-a-third-party-plugin-in>

Other features related to Host/VST interactions

- **Bank Loading and Saving** The plugin fully implements VST bank load and save protocols. DAWs utilize these functions to store and restore the plugin's entire state within the project file.

The stored information includes:

- Full instrument file paths.
- All UI control positions (knobs, drop-down menus, VST VARs).
- Keyswitch states and the currently selected temperament.
- Program and Bank numbers for each of the 16 MIDI channels.
- Sequencer status and TAPE file paths.
- Data entered in the SAMPLEUI controls.

When a project is loaded, Crescendo restores all variables to their saved states, then reloads and recompiles the instrument file.

NOTE: The VST assumes the underlying instrument file has not been modified since the project was last saved. If the file has been edited externally, the restored UI parameters may no longer align with the new instrument structure, leading to inconsistent results.

- **Error Handling & Diagnostic Feedback during Recall** When a project is recalled, Crescendo initiates a deep-loading sequence to restore all engine parameters, sample pointers, and layer configurations. To ensure transparency, a **Progress Monitor** is deployed. The Progress Monitor is designed to be informative rather than interruptive, providing real-time diagnostic data:
 - **Non-Blocking Error Reporting:** If the plugin encounters missing samples (e.g., due to moved folders or renamed assets), it will not trigger a system hang or a modal error dialogue.
 - **The Error Counter:** If the engine detects a missing resource, an "Errors Detected" counter will appear within the progress window.
 - **Actionable Info:** This allows the user to immediately identify if a project is incomplete while the loading is still in progress.
 - **Engine Integrity:** A non-zero error count indicates that the instrument will still be playable, but some audio assets will be silent. This ensures you can always access your project and resolve path issues later.

- **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 performances. 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. It would be prudent to check the documentation or support resources for your specific DAW to understand its multithreading behavior with VST plugins.
- **MIDI channel mode messages:** The messages number 120, 121 and 122 cause a suspend/resume cycle, that stops all notes, delay and reverbs and resets the sequencer. The other MIDI channel mode messages put in release all active layers, without performing the MIDI POST step. These messages can be disabled, disabling the relative MIDI CC with MCCCTL. They are NEVER retransmitted in the MIDI Output. Regarding other MIDI features, the OMNI mode is always ON and the POLY mode depends on the POLY instruction setting, so the relative channel mode messages are ignored.
- **MIDI system messages:** They are ignored.
- **MIDI RPN/NRPN messages:** They are processed, per channel and they are accessible with the related functions that are RPN, RPN2 and RPNLINK. See below.
They are accessible, but they do not by default change the behavior of the VST, e.g. RPN message 6 (master tuning) changes the storage of the internal variable associated with it, but it's up to the instrument developer read it and apply it, e.g. changing the base frequency. These messages can be disabled, disabling the relative MIDI CC with MCCCTL. They are NEVER retransmitted in the MIDI Output.
- **MIDI messages routing:** By default all enabled input MIDI messages of the enabled channels are processed for modification of the local storage and routed to the MIDI Output, if enabled for the specific MIDI CC. NOTE ON and OFF are processed and changed before sending them on the MIDI Output and may be further created or filtered by the SEQUENCER, arpeggiator or chord generator.

Files and Directories

- **Main Directory:** 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.
- **Cache Directory:** 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:** Crescendo's "SF2 Import" feature, which converts SoundFont 2.04 files, creates a dedicated directory, located in `<My Documents>\Crescendo\Imported`. This directory houses subdirectories and files related to imported SoundFonts.
- **“Temperaments” Directory:** The default `Setting.ini` created when there isn't one contains an instruction to import all SCALA and TUN temperaments found in the folder located in `<My Documents>\Crescendo\Temperaments`. This directory is not set in stone in the code. To change the directory or disable the importing, just edit the `Settings.ini` and delete or modify the row `TEMPERAMENT "Temperaments"`.
- **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.
- **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:** They define how audio is processed, including layer specifications, sound generation, and post-processing.
 All instrument files should reside in `<My Documents>\Crescendo`, or some custom subdirectory 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).
 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.
 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 allows the user to use their preferred text editor.
 These instrument files contain code written in Crescendo's custom 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.
 - **Audio Files:** The paths to sample files, referenced in the instrument files using the `SAMPLE` or `SAMPLES` instructions, 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.
 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.
- **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.

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.

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

- **SoundFont Files:** Crescendo supports importing SoundFont 2.04 files (.sf2 extension), allowing users to integrate pre-existing sound libraries into the Crescendo environment. See above in the UI section for details on the importing process.
- **Tuning Files:** 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:** The sequencer within Crescendo can store and load sequences of MIDI notes or messages using TAPE files (.tap extension). These files utilize the CSV (comma-separated value) format, making them easily editable and shareable. Only ANSI coding is supported, since it is sufficient for numerical values.
- **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:** 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:** 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, add it to the list of available temperaments and make it the active temperament. This is only to test it though. To permanently add it to the instrument file, even when the DAW project is reloaded, it must be added to the instrument file. For live performances, drag and drop may be sufficient.
 - **Instrument and SoundFont Files:** 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. 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.

Architectural Philosophy: Why VST 2.4?

Users often ask why Crescendo is built on the **VST 2.4** standard rather than the newer **VST3**. This is a deliberate design choice based on one of Crescendo's core strengths: **Dynamic Identity**.

The "Shapeshifter" Problem

Most modern plugin standards, particularly **VST 3.0**, require a plugin to declare its "identity" (Type) the moment it is instantiated. You must tell the DAW exactly what you are:

- A MIDI Effect
- A Virtual Instrument
- An Audio Processor

Crescendo refuses to be put in a box. Because Crescendo is a programmable environment, it can change its identity at runtime based on the script you load.

Dynamic Runtime Identity

In Crescendo, you might start a session as a simple synthesizer (Instrument), but then decide to load a script that acts as a MIDI Sequencer and a Sidechain Compressor simultaneously.

- **The VST 2.4 Advantage:** This older but more flexible standard allows the plugin to remain "agnostic." It can accept MIDI, output MIDI, and process audio all at once without the DAW forcing it into a specific category.
- **The VST 3.0 Limitation:** VST3 uses a strict "typing" system. If a plugin is registered as an "Instrument," it often struggles to function as a pure MIDI Effect in many hosts because the standard enforces rigid input/output rules.

One of the most restrictive elements of VST3 is its handling of **Bus Management**. In VST3, changing the number of audio inputs or outputs typically requires the host to "reconstruct" the plugin instance. This process often leads to audio dropouts, CPU spikes, or the loss of volatile script states.

Crescendo utilizes the VST 2.4 `ioChanged()` mechanism. When a user loads an instrument script that requires a specific I/O configuration (e.g., shifting from a stereo synth to a 4-channel surround processor using the `IO` instruction), Crescendo signals the host to update the bus count **at runtime**.

- **Host Engagement Rule:** For hosts that do not support dynamic I/O changes, Crescendo is engineered to read the `IO` instruction during the initial handshake, ensuring compatibility while maintaining the possibility of "on-the-fly" adaptation in advanced DAWs.

Host Behavior: Ableton vs. Reaper

The freedom of VST 2.4 also highlights how different DAWs handle plugin communication:

- **Ableton Live:** Tends to treat VST 2.4 with a structured approach, often trying to guess the plugin's intent.
- **Reaper:** This host embraces the full flexibility of the 2.4 standard, leaving the plugin free to route MIDI and audio in any configuration the script demands.

Three Plugins in One

In many DAWs, a single instance of Crescendo can effectively function as **three plugins at once at the same time**:

1. **MIDI Effect:** It can generate or transform MIDI notes (Post-processing).
2. **Instrument:** It can synthesize sound via its oscillators and layers.
3. **Audio Effect:** It can process external audio (plus its internal instrument audio) via the `POST` section and sidechain inputs.

Direct Parameter Transparency

VST3 mandates a strict separation between the **Processor** (the audio engine) and the **Edit Controller** (the UI). While conceptually clean, this adds significant "bureaucratic" overhead when a plugin needs to dynamically generate and automate many parameters based on a user-written formula.

The VST 2.4 standard allows Crescendo a direct, low-latency path to the host's automation buffer. This ensures that even the most complex, recursive formulas remain **sample-accurate** and highly responsive to DAW automation without the translation layers required by VST3.

Why Not VST3? A Summary of Constraints

To port Crescendo to VST3 would mean accepting a "downgrade" in creative freedom:

1. **Identity Fragmentation:** We would be forced to release three separate binaries (*Crescendo_Synth*, *Crescendo_FX*, *Crescendo_MIDI*) just to satisfy the VST3 category requirement. Now it is done only for compatibility.
2. **Host Silencing:** Many VST3 hosts "suspend" a plugin if it is labeled as an instrument but no MIDI is present on the track. VST 2.4 ensures that Crescendo's internal clocks, sequencers, and LFOs keep running regardless of input presence.
3. **Port Filtering:** VST3 DAWs often hide or disable MIDI Output ports for plugins labeled as "Audio Effects." Crescendo refuses to be silenced; if you want a distortion effect that also generates MIDI data, VST 2.4 is the only standard that guarantees this autonomy.

By choosing VST 2.4, Crescendo ensures that you never have to reload a plugin just because you changed your mind about what it should do and you can create a plugin that does what other plugin don't: three things a time.

VST3 was designed to simplify things for the DAW, but it did so by taking power away from the developer. For an environment as dynamic as Crescendo, where the code determines the routing, VST 2.4 remains the superior choice for unrestricted creativity.

The "Instant Effect" Architecture: Why `IN` is already in `OUT`

In most modular environments, you must manually route an input to an output. In Crescendo, the audio engine is designed with a "Pass-Through" philosophy. When the `POST` section begins its evaluation, the variable `OUT` is already pre-loaded with the incoming audio signal (`INZ`).

Audio Effects: "For Free"

Because `OUT` starts as a copy of the input, every Instrument you build is also an Audio Effect by default. You don't need to write a single line of code to hear your external audio passing through the plugin.

- **Zero-Effort Routing:** If you load a synthesizer script and place it on an audio track, the track's audio will pass through untouched until you decide to "multiply" or "process" that `OUT` variable.
- **Hybrid Power:** You can play a MIDI melody (using `LAYER`) while simultaneously applying a filter or a delay to the audio coming from your DAW using the same script.

Simplified Scripting Logic

This design choice eliminates "boilerplate" code. Compare these two approaches for creating a simple volume control for an external signal:

- **Traditional Approach:** `OUT = INZ * VOLUME`
- **Crescendo Approach:** `OUT = OUT * VOLUME`

In the second example, `OUT` already contains the audio data. This allows you to chain effects naturally:

1. `OUT = FILT(0, OUT, 1000, 0.5)` (Filter the incoming audio)
2. `OUT = OUT * 0.5` (Lower the volume of the filtered result)

The Instrument + Effect Synergy

This architecture is particularly powerful for creating "Vocoder-style" or "Sidechain-aware" instruments. Since the external audio is already sitting in the `OUT` buffer at the start of the `POST` step, you can use the instrument's internal oscillators to modulate the external audio instantly.

Developer's Note: The "Clean Slate" Fallback

If for some reason you want to ignore the external audio entirely and only hear the internal synthesizer, you can simply 'reset' the buffer at the top of your `POST` section using `OUT = OUT - INZ`. However, leaving it active allows for a 'Serial' processing chain where Crescendo acts as both the sound source and the final processor in your signal path.

Error Handling and Debugging

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 messages 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

Crescendo employs high-level engineering strategies to ensure smooth operation and prevent CPU bottlenecks, even when running high-voice-count instruments or complex effects chains.

Pre-Compilation Stage

Instruments are pre-compiled into an efficiently interpreted intermediate language. Furthermore, the pre-filtering of sample slots is performed at this stage. By completing these operations beforehand, Crescendo avoids redundant calculations during real-time runtime, improving overall efficiency.

Aggressive Voice Release

To manage polyphony and avoid excessive CPU load, Crescendo uses an aggressive voice strategy: inactive voices (notes 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 & OSCG Options

- **Core Optimization:** Oscillator functions are specifically designed for efficient execution, minimizing CPU usage while supporting default modulations.
- **Mono/Single Efficiency:** The OSCG function supports mono output and single-phase/frequency/amplitude options, bypassing unnecessary stereo calculations when they are not required by the patch.

Hardware-Level Denormal Protection (Flush-to-Zero)

Crescendo implements robust hardware-level protection against "denormal" numbers—infinitesimally small values near zero that can cause massive CPU spikes as processors struggle to calculate them.

- **Automatic Management:** The audio engine enforces **Flush-to-Zero (FZ)** and **Denormals-Are-Zero (DAZ)** flags at the processor level.

NaN and "Overload" Protection

Crescendo includes measures to protect against **NaN** (Not a Number) values or signal "overloads" (where the absolute value of any output exceeds 100,000), which can disrupt audio processing or damage equipment.

- **Emergency Reset:** If protection is triggered, Crescendo automatically resets the internal memory buffers (stores) of reverbs and delays in the POST step and terminates all active LAYERS.
- **Housekeeping:** Stores and LAYERS (or the POST step) are also reset whenever the VST is suspended or resumed.
- **Logging:** These events are recorded in the debug log, if enabled, to assist in troubleshooting, preventing invalid values from propagating through the audio chain.

FFMPEG Caching

If FFMPEG is installed, Crescendo caches converted files in the <My Documents>\Crescendo\Cache directory. This mechanism uses file size and modification timestamps to identify and reuse previously converted files, drastically speeding up the loading process.

AVX Versions of Crescendo

Crescendo is distributed in multiple versions compiled to utilize specific **AVX** instruction sets (SIMD), allowing the processor to handle multiple data points simultaneously for significant performance gains.

- **Vanilla:** Functions on all x64 CPUs without specific requirements.
- **AVX / AVX2 / AVX512:** Optimized for modern high-performance architectures.
- **AVX10.1 / AVX10.2:** The latest standard, compiled with 512-bit operation sizes (as supported by Visual Studio 18.3 and later).
- *Users should select the version that aligns with the highest instruction set supported by their CPU to maximize efficiency.*

Current implementation limits

Here we list the maximum size of some structures, that limits the maximum complexity of the instrument:

- The current row length limit is fixed at 8192 characters.
- The current implementation supports maximum 32768 program name entries.
- A maximum of 16384 temperaments can be defined.
- The assignments are not really unlimited. You can perform 8192 operations and have 8192 temporary slots, including custom variables, per layer or post step. Giving current and near future CPUs this is not going to be a limitation.
- Maximum 1024 arpeggio notes can be specified.
- The tape is 524288 slots in the current implementation.
- Maximum 8192 sequencer instructions can be defined in the "Program".
- Maximum of 256 curves of 2048 points are supported.
- Maximum 256 names are allowed, per each textual VST variable, all sharing 524288 bytes for text storage.
- Maximum 64 modulations per layer per each of GAIN, FREQ, DEFAULTFC are supported.

Programming the Crescendo Engine.

This section transitions from abstract architectural theory to a hands-on laboratory, providing you with "How-To" guides and "Hello World" style samples designed to help you master the plugin's unique programming language. You will find more comprehensive tutorials in the last sections. The tutorials seen here are targeted to show the functionality of the feature being described.

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 3 Declaration Rows
...
...
LAYER
// Layer n Declaration Rows
...
...
// End File
```

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.

- **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.

- ***MOD, *ENV, *LFO, BASEG, BASEF, KEYTRACK, KEYCENTER, VOLTRACK:** Declares default formulas, modulations, envelopes and LFO for GAIN, oscillator frequency and filter cutoff.
- **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.
 - **LINKCC:** Links any MIDI CC with any other MIDI CC.
 - **MIDIOUT:** Put on the MIDI Output, if enabled, a custom expression.
 - **SEQUENCER[I]:** Activates and configures Crescendo's built-in sequencer, enabling the generation of MIDI note data based on patterns and sequences.
 - **MAP, VELOCITY, PITCH, TIMING, and MIDIPOST:** 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.
 - **TEMPERAMENT:** Allow the definition or loading of temperaments to apply to the notes to be played.
 - **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 assignments:** Establishing default values for parameters that might be used by multiple layers or assignments to MIDICC that should be processed by any layer.
 - **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 or when the file terminates.

Note that most global settings can be overwritten into the **LAYERs** or **POST** step. In the descriptions that follow, it's stated where an instruction or function can appear. If it can appear in the **COMMON** section and also in the **LAYER** or **POST** section, then the feature can be overwritten from its global behavior.

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.

Signal Manipulation and Routing

- **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>i</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.

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.

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. If you need only the layers sum, you can subtract INZ from OUT as a first step and then use the new OUT and INZ as you wish, for example as a sidechain.
 - 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, INZ 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.
 - INZ: This variable represents the signal received on INPUT #1, often used as main input of an effect. In some cases (Instrument in a MIDI track) is free to be used for sidechaining, both in the LAYERS and POST step.
 - 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

- **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.
- **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.

Final Output

- **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.

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

NOTE: input and output variables may not be defined if using the Crescendo-MIDI.dll version or if you set zero as number of inputs or outputs in the IO instruction.

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 features 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. All the defined triggers must be satisfied for the note to be triggered (AND mode). 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.
 - Previous note states: legato and round robin is implemented with keywords allowing to trigger a layer only in determined conditions.
- **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 nonlinear 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, but it is sufficient that one is satisfied to trigger the release stage (OR mode). They are:
 - **NOTEOFF:** When the corresponding MIDI note is released. This trigger is always active and does not need to be specified.
 - **GROUP:** When a layer of a given group is triggered.
 - **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 (excluding NOTE OFF, that is implicit), can also include a **forced release parameter**, which overrides the normal release time of all envelopes, enabling quick and abrupt sound termination under the specified conditions.

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

It is important to note that the LAYER subdivision and other features (like the crossfades) are supported to ease the porting of instrument from other plugin. Since Crescendo supports unlimited oscillators, envelopes, etc., per layer, it is not necessary to have split layers: you can merge multiple layers into one, just summing the oscillators. Actually even the SoundFont import feature works like that: if an instrument is multilayered, it tries to merge multiple layers into one, summing the various components. It's more efficient to sum multiple oscillators inside a single layer than having separate layers. Also with split layers you can reach more easily the polyphony limit. So use multiple layers only when strictly necessary, e.g. when splitting for notes or velocity ranges or program and bank number.

Learning Through Example

To facilitate your development process, this chapter intersperses technical documentation with heavily commented tutorials that serve as functional "Hello World" programs. By following these practical implementations, you will learn to use the engine's assignment system to programmatically modify audio and MIDI data at a sample-accurate level.

The tutorial samples highlights the instruction or feature currently discussed and not necessarily are a complete fully functional instrument. For complete and fully functional tutorials, there is another chapter below.

Initializations Instructions

Initialization instructions serve as the foundational configuration layer for every Crescendo instrument file. These commands establish the global environment in which your synthesis and processing logic will operate, defining critical parameters such as the number of available audio ports, the complexity of user-facing controls, and the overall visual layout of the plugin interface.

While these instructions are primarily located within the **COMMON** section of an instrument script, they can also be defined globally within the `Settings.ini` file located in your `<My Documents>\Crescendo` directory. The Crescendo engine processes these settings in two distinct stages: Stage 1 occurs during the initial loading of the DLL to satisfy host requirements for I/O and parameter counts, while Stage 2 occurs every time an instrument file is loaded or reloaded, allowing for instrument-specific overrides.

Core Configuration Categories

The initialization suite is divided into several functional categories to ensure a modular and organized development workflow:

- **Audio and Host Connectivity (`IO`, `VSTVARS`, `VSTVAR`):** These instructions define how the plugin communicates with the host DAW. The `IO` command sets the physical count of audio inputs and outputs, which is essential for sidechaining and multi-output routing. The `VSTVARS` and `VSTVAR` instructions manage the exposure of automatable parameters, turning raw internal variables into user-controllable knobs on the GUI.
- **Engine Performance and Accuracy (`QUALITY`, `MAXDLY`):** To balance CPU efficiency with audio fidelity, the `QUALITY` instruction allows you to fine-tune the sample interpolation window and oversampling factors. Additionally, the `MAXDLY` command defines the memory size allocated for delay buffers, ensuring that time-based effects have the necessary resources during real-time processing.
- **User Interface and Aesthetics (`INTERFACE`, `UIMOD`, `SETFONT`, `HIDEUI`):** The look and feel of the instrument are highly customizable. `INTERFACE` sets the window dimensions, while `UIMOD` provides granular control over the position, size, and color of knobs and dropdown menus. For further personalization, `SETFONT` allows for the integration of system TrueType fonts, and `HIDEUI` enables developers to declutter the workspace by removing non-essential information like time signatures or bank numbers.
- **Project Management (`INCLUDE`):** To support complex instrument designs, the `INCLUDE` instruction facilitates a modular approach by allowing you to merge external text files into a single master script, promoting code reuse across multiple projects.

Proper utilization of these instructions ensures that your instrument is not only functional but also optimized for the specific constraints of your host DAW and the needs of the end user. The following detailed technical references provide the exact syntax and operational ranges for each initialization command.

INTERFACE <size x>;<size y>

The `INTERFACE` instruction defines the dimensions of the instrument's graphical user interface. While typically set in the global configuration, it can be declared within an instrument file to

dynamically expand the workspace for additional controls, such as knobs, drop-down menus, or custom meters.

1. Technical Syntax

```
INTERFACE <size_x>, <size_y>
```

- **<size_x> (Width):** The width of the GUI in pixels.
- **<size_y> (Height):** The height of the GUI in pixels. (Optional; if omitted, the current height is preserved).

2. Operational Logic

- **Declarative Priority:** This instruction is declarative. If multiple `INTERFACE` calls exist, the **last specification** encountered overrides all previous ones. Use the `FEND` prefix to ensure your declaration takes absolute priority over global settings.
- **Minimum Size Logic:**
 - If a value is **positive**, the dimension is set exactly to that value.
 - If a value is **negative**, the engine treats it as a **minimum constraint**. The dimension will only be updated if the absolute value (`|size|`) is greater than the current setting.
- **Automatic Expansion:** There is no hard lower limit; however, the engine will automatically increase dimensions if the automatically placed GUI elements (knobs, etc.) do not fit within the defined area. Knobs placed with `UIMOD` do not participate in the minimum dimensions calculation.
- **Display Limits:** The maximum size is restricted by the user's screen resolution, minus 50 pixels for the OS title/status bars. These limits are calculated at runtime based on the current **Zoom Factor and DPI/FONT scaling**.
- **DPI and Font Scaling:** All pixel values assume a baseline of **96 DPI** and a **16 DLP (8pt) primary font**. Crescendo scales these values proportionally for High-DPI displays or larger font settings (e.g., at 192 DPI and 32 DLP font, values are multiplied by 4).

3. Practical Examples

Example A: Fixed Window Setup

Sets the interface to a specific, static size.

```
// Force the window to exactly 800x200 pixels
INTERFACE 800, 200
```

Example B: Setting Minimum Constraints

Ensures the window is at least a certain size without shrinking it if it was already larger.

```
// Ensure the width is at least 800 and height is at least 200
INTERFACE -800, -200
```

Example C: Height-Only Adjustment

Useful when adding an extra row of controls without wanting to risk messing with the width.

```
// Keep current width (-1) but ensure the height is at least 200 pixels
INTERFACE -1, -200
```

4. Developer's Notes

- **LMMS Constraint:** Some DAWs, notably LMMS (tested on v1.2.2), lock the VST window size upon the first load of the plugin. If you change the interface size while the plugin is active, you may see clipping or blank space. Reload the VST to force the DAW to recognize the new dimensions.
- **Clarity:** Always pair `INTERFACE` with `FEND` if you are writing a specialized instrument that requires a specific layout to be visible regardless of the user's global `.ini` settings.

IO <numInputs>, <numOutputs>

The `IO` instruction defines the number of physical stereo audio channels (inputs and outputs) the VST exposes to the DAW (Host). This allows for complex routing setups, such as sidechaining or multi-output instruments where different layers (drums, synths, etc.) are sent to separate mixer tracks.

1. Technical Syntax

```
IO <numInputs>, <numOutputs>
```

- **<numInputs> (Stereo Pairs):** The number of stereo input pairs available for sidechaining or effect processing. (Range: 0 to 101).
- **<numOutputs> (Stereo Pairs):** The number of stereo output pairs available for multi-out routing. (Range: 0 to 100).

2. Operational Logic

- **Section Constraint:** This instruction must be placed exclusively in the **COMMON** section.
- **Default Values:**
 - **Crescendo.dll / Crescendo-effect.dll:** 2 Inputs, 1 Output pair.
 - **Crescendo-MIDI.dll:** 0 Inputs, 0 Outputs (Defaulted for pure MIDI/Control use).
- **Host Compatibility:** When this instruction is called, Crescendo requests the change from the DAW.
 - If the Host supports dynamic I/O (returns `true`), the new configuration is applied immediately.
 - If the Host denies the change (returns `false`), Crescendo reverts to the previous stable configuration.
- **Initialization Note:** `Crescendo-MIDI.dll` ignores the `IO` instruction during the initial handshake. If your DAW does not support dynamic I/O changes, you will be unable to add audio ports to the MIDI version of the plugin after it has loaded.

3. Practical Implementation & Nuances

Global vs. Local Config

If your DAW (like some versions of Ableton Live or Logic) struggles with dynamic I/O, it is highly recommended to place the `IO` instruction in the `Settings.ini` file rather than the instrument file. This forces the VST to initialize with the desired number of ports every time it is loaded.

Performance Overhead

Having "too many" inputs or outputs carries a **negligible CPU cost**, provided those channels are not actively being processed or connected within your code. It is safer to over-provision ports in `Settings.ini` than to lack them during a complex production.

4. Examples

Example A: Sidechain/Vocoder Setup

Configures the VST to receive two stereo external inputs (e.g., a Kick for sidechaining and a Voice for vocoding) and one main output.

```
// --- COMMON SECTION ---
// 2 Stereo Inputs, 1 Stereo Output
IO 2, 1
```

Example B: Multi-Output Drum Machine

Configures the VST with no inputs and 8 stereo outputs, allowing you to route Kick, Snare, Hats, etc., to separate DAW mixer tracks.

```
// --- COMMON SECTION ---
// 0 Inputs, 8 Stereo Outputs
IO 0, 8
```

5. Developer's Note

When using multiple outputs, remember that `OUT` defaults to the first output pair (Output 0). To route audio to additional ports, you must specify the output index in your `LAYER` or `POST` code (e.g., `OUT1`, `OUT2`, etc.).

VSTVARS <number of vstvars>

The `VSTVARS` instruction defines the quantity of "VST Variables" (parameters) that Crescendo exposes to the DAW. These variables act as the bridge between your instrument's internal logic and the host's automation lanes, GUI knobs, and MIDI mapping systems.

1. Technical Syntax

`VSTVARS <number>`

- **<number> (Quantity):** The total number of parameters to be registered with the Host DAW.
 - **Range:** 1 to 128 (Current version limit).
 - **Default:** 16 (Assumed if the instruction is missing or invalid).

2. Operational Logic

- **Section Constraint:** This instruction must be placed exclusively in the **COMMON** section.
- **Declarative Priority:** If specified multiple times, the **last** declaration overrides all previous ones.
- **Visible vs. Hidden Vars:**
 - **Visible ($\text{Index} < \text{VSTVARS}$):** These are "Public" parameters. They appear in the DAW's automation list and can be manipulated manually via the VST's generic interface or assigned to custom GUI knobs.
 - **Hidden ($\text{Index} \geq \text{VSTVARS}$):** You can still access higher-indexed variables (up to the internal engine limit) within your instrument code. However, they are "Private"—they cannot be automated by the DAW or seen by the user unless they are explicitly linked to a MIDI CC or Temperament setting.
- **Dynamic Range:** If a value greater than 128 is specified, the engine clips it to 128 without throwing an error.

3. DAW Constraints & Persistence

Host Limits

Be aware that different DAWs have varying limits on parameter counts. For example, some versions of **Ableton Live** may only display up to 64 parameters. If you define 128 but only see 64, it is likely a Host-side limitation.

Caching Issues

Most DAWs (like Ableton) read the parameter count **only when the VST is first instantiated**. Furthermore, the DAW often caches the parameter list within the project file.

- **The Problem:** If you increase `VSTVARS` in an instrument file but load that instrument into an *existing* project, the DAW may still show the old parameter count.
- **The Solution:** You must delete the VST instance from the track and re-insert it to force the DAW to refresh the parameter map.

4. Practical Examples

Example A: Minimal Interface

Allocates only 4 parameters to keep the DAW automation list clean and uncluttered.

```
// --- COMMON SECTION ---  
// Expose only 4 variables to the DAW  
VSTVARS 4
```

Example B: Utilizing Hidden Constants

Allocates 16 visible parameters for automation, but uses `VAR(20)` (which is hidden) to store a fixed internal constant or a value scaled by a MIDI CC.

```
// --- COMMON SECTION ---
```

```
// Inside a LAYER, VAR(20) can still be used as a private storage slot
LAYER OUT = OSCG("sine") * VAR(20)
```

Developer's Note

Use hidden VST variables (indices higher than your VSTVARS count) as "scratchpad" memory for complex logic or to store values from MIDICC links that you don't want the user to accidentally automate or overwrite via the Host GUI.

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

The VSTVAR instruction configures a specific automation parameter. It defines its initial state, its visual label, its range, and—most importantly—how the movement of a knob translates into mathematical values for the synthesis engine.

1. Technical Syntax

```
VSTVAR <index>, <initial>, "Name", "Unit", <min>, <max>, <scale_type>,  
[;<names>...]
```

- **<index>**: The ID of the variable (0 to 127). Indices below the VSTVARS count appear as GUI knobs.
- **<initial>**: The starting value when the instrument file is loaded.
- **"Name"**: The label shown on the GUI knob and in the DAW automation list.
- **"Unit"**: The measurement unit (e.g., "Hz", "dB", "ms") shown next to the value.
- **<min> / <max>**: The values mapped to the minimum (0%) and maximum (100%) knob positions.
- **<scale_type>**: Defines the mathematical curve and behavior:
 - **0 (Logarithmic)**: Best for Frequency (Hz) or Time (ms). `min` and `max` must be > 0 .
 - **1 (Linear)**: Standard linear mapping.
 - **2 (Integer)**: Values are rounded to the nearest whole number.
 - **3 (Beats)**: Quantizes values to musical subdivisions (e.g., 1/16, 1/4, 2). `min` and `max` must be > 0 .
 - **4 (Integer with Names)**: An integer scale where specific values display a text label (e.g., "Sine", "Square") instead of a number.
- **[;<names>]**: Optional list of strings for Scale Type 4 (max 256 names, 19 characters each).

2. Operational Logic & Constraints

- **Section Constraint**: Must be placed in the **COMMON** section.
- **Visibility**: Knobs are only drawn for variables with an index less than the value set in VSTVARS. Variables with higher indices are "hidden" but can still be updated via MIDI CC or internal logic.

- **Inverted Scales:** You can set <min> higher than <max> to create inverted knobs (e.g., 100% position results in the minimum value).
- **Automation Override:** If a user moves a knob while the Host DAW has automation active, Crescendo sends a notification to the host to temporarily pause or override that automation.
- **Storage Limits:** Accessing variables 0–127 is standard. Variables 128–199 return always zero. Indices above 199 are reserved for other internal system mapping (INs, OUTs, KEYSWITCHES).
- **System Initialization:** Upon loading, a [LOCAL] message is triggered for every VSTVAR, ensuring that all linked MIDI CCs, storage updates, and UI reflections are synchronized immediately.

3. Scaling Types in Detail

Scale Type	Name	Best Used For	Behavior
0	Logarithmic	Filters, Frequencies	Fine control at lower values; fast jumps at higher values.
1	Linear	Volume, Dry/Wet	1:1 movement between knob and value.
2	Integer	Polyphony, Step count	Snaps to the nearest whole number.
3	Beats	LFO Sync, Delay Time	Snaps to 1/32, 1/16, 1/8, 1/4, 1/2, 1, 2, etc.
4	Named	Waveforms, Modes	Displays "Sine" at value 0, "Square" at value 1, etc.

4. Practical Examples

Example A: Standard Oscillator Controls

Configures a waveform selector with names and a standard linear volume knob.

```
// --- COMMON SECTION ---
VSTVARS 2

// Waveform selector (Integer with Names)
VSTVAR 0, 0, "Waveform", "", 0, 4, 4, "Sine", "Square", "Saw", "Triangle",
"Noise"

// Volume control (Linear 0.0 to 1.0)
VSTVAR 1, 0.5, "Volume", "", 0, 1, 1
```

Example B: Synced Modulation

Configures a retrigger rate that snaps to musical beats and an envelope with logarithmic time.

```
// --- COMMON SECTION ---
VSTVARs 2

// Retrig snaps to bar divisions (0.008 is approx 1/128, 5 is 5 bars)
VSTVAR 0, 0.25, "Retrig", "Bars", 0.008, 5, 3

// Logarithmic attack time for fine control in the 'fast' range
VSTVAR 1, 0.01, "Attack", "ms", 0.001, 10.0, 0
```

Developer's Note

When using **Scale Type 4 (Named)**, the underlying variable is still numeric. In your synthesis code, simply use `VAR(n)` or equivalent to determine which mode is selected. If a name is not specified for a specific integer value, the GUI will default to showing "Option #n".

TOOLTIP "String", <VST VAR num>

The `TOOLTIP` instruction assigns a descriptive text block to a specific VST variable. This text appears automatically when the user hovers their mouse over the corresponding knob in the Crescendo GUI.

1. Technical Syntax

```
TOOLTIP "String", <index>
```

- **"String"**: The descriptive text to display.
 - Supports ANSI and UTF-8 encoding.
 - Supports line breaks via the `\n` character.
 - **Maximum Length**: 511 bytes.
- **<index>**: The ID of the VST variable.
 - Supports the standard range (**0–127**).
 - Supports the internal mapped range (**600–727**) for consistency with MIDI CC logic.

2. Operational Logic

- **Section Constraint**: Must be placed exclusively in the **COMMON** section.
- **Sequencer Integration**: If a `VSTVAR` is linked to a Sequencer/Tape control, Crescendo intelligently prepends your custom tooltip string to the name of the currently loaded `.TAPE` file.
- **Removal**: To delete a tooltip and revert to the default display, call the instruction with an empty string: `TOOLTIP "", <index>`.

3. Practical Example

Example: Detailed Multi-line Explanation

Use `\n` to organize information, making it easier for the user to read technical details.

```
// --- COMMON SECTION ---
VSTVAR 2, 500, "Cutoff", "Hz", 20, 20000, 0
TOOLTIP "Adjusts the Filter Cutoff.\nLower values create a darker sound,\nhigher values are brighter.", 2
```

4. Developer's Notes

- **Conciseness:** While 511 bytes is plenty of space, aim for brevity. Users typically want to know *what* it does and *how* to use it at a glance.
- **Formatting:** Use `\n` strategically to separate the "Action" (e.g., "Controls volume") from the "Warning" (e.g., "Caution: High values may clip").

DPIAWARE <flag>

The `DPIAWARE` instruction determines how the VST interacts with the Operating System's scaling engine. Enabling this ensures that text and graphics remain crisp and high-definition on modern high-resolution (4K/5K) displays, rather than being "pixel-stretched" and blurred by the OS.

1. Technical Syntax

`DPIAWARE <flag>`

- **<flag>:**
 - **0: Disabled.** The OS handles scaling. The interface may appear blurry on high-DPI monitors as it is digitally upscaled.
 - **Non-Zero (1): Enabled.** Crescendo forces the Host process (the DAW) to declare itself as "DPI Aware" to the Windows API.

2. Operational Logic

- **Section Constraint:** This instruction must be placed exclusively in the **COMMON** section.
- **Initialization Only:** This command is executed **only at the moment the VST is loaded**. Changes made within an instrument file will not take effect until the plugin instance is reloaded.
- **Process-Wide Impact:** This is a powerful system command. Because Windows handles DPI awareness at the **process level**, once a DAW process declares itself DPI aware (either natively or forced by Crescendo), **this declaration cannot be retracted** until the DAW is restarted.

3. Compatibility & Warnings

- **Handshake Logic:** Crescendo automatically detects if a Host is already natively DPI aware. If so, it will sync its scaling accordingly.
- **The "Ableton Risk" (Non-Native Scaling):** Some DAWs (e.g., **Ableton 9.7.5**) are not natively DPI aware but use their own internal bitmap scaling. Forcing `DPIAWARE 1` in these environments can "fool" the OS, resulting in a sharp plugin interface but a broken, unscaled DAW interface or offset mouse coordinates. **Do not activate this for Ableton 9.7.5.**
- **Irreversibility:** Remember that if you force awareness in a DAW that doesn't support it, you may need to restart the DAW to fix the layout of other plugins.

- **Recommended Usage:** Use `DPIAWARE 1` if your interface looks small or blurry. If the interface becomes "cut off" or mouse clicks don't land where the cursor is, revert to 0.
- **Standardization:** It is best practice to set this flag in the `Settings.ini` to ensure all Crescendo instances behave consistently across your projects.

4. Examples & Debugging

Example: Enabling High-Resolution Support

Ensures the GUI is rendered natively at the monitor's full resolution for maximum clarity.

```
// --- COMMON SECTION ---
// Enable High-DPI awareness for crisp graphics
DPIAWARE 1
```

Monitoring Status

To verify the current scaling state, you can increase the `DEBUG` level (see the `DEBUG` instruction). At higher levels, Crescendo will output the current DPI Awareness status of the Host process to the log/console.

Developer's Note

DPI awareness is a complex handshake. When enabled, Crescendo scales its internal **DLP (Device Logical Pixels)** based on the detected system scale. Ensure your `INTERFACE` dimensions are defined with this scaling in mind to avoid unexpectedly large or small windows.

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

This instruction controls the verbosity of the debug log and the visual layout of the log window. It is essential for verifying that your Reverse Polish Notation (RPN) expressions are being calculated as intended.

1. Technical Syntax

```
DEBUG <level>, <log_pos>, <log_border>
```

- **<level>:** Sets the verbosity of the messages. (Range: 0 and up).
- **<log_pos> (Optional):** Defines the horizontal split or orientation of the log window.
 - **20% to 80%:** Sets the log window as a vertical panel starting at that percentage of the window width.
 - **81 and higher:** Switches the log to a horizontal panel located underneath the knobs.
- **<log_border> (Optional):** Sets the padding/border of the log window in pixels (Max 60).

2. Debug Levels

Level Name	Description
0 Disabled	No log window is drawn. Only GUI controls (knobs, etc.) are visible.
1 Errors	Displays the log window and logs only critical error messages.
2 Informative	Logs quality settings, temperament changes, MIDI CC assignments, and RPN operation lists.
3 Parsing	Full parser details. Shows how the engine breaks down expressions into internal operations.
4+ Verbose MIDI	Detailed dumps of all incoming and outgoing MIDI data packets.

3. Operational Logic

- **Placement:** Can be placed anywhere in the instrument file.
- **Scope Control:** You can wrap specific sections of code with `DEBUG` instructions to isolate troubleshooting. Only the code between a high-level `DEBUG` and a lower-level `DEBUG` will produce verbose parsing logs.
- **Final Layout:** While the level can change throughout the file to toggle logging, the **last** `DEBUG` instruction in the file determines whether the log window is actually rendered in the GUI.
- **Runtime Updates:** In play mode, the log window captures real-time events like MIDI CC movements, mouse-wheel interactions, and keyswitch triggers.
- **On-the-fly Adjustment:** You can change the debug level at runtime by using the **mouse wheel** while hovering over the version string in the GUI.

4. Practical Examples

Example A: Troubleshooting an Expression

To see exactly how the engine calculates a complex filter mix, wrap the target lines in Level 3 debug.

```
DEBUG 3 // Enable verbose parsing
OUT=OSCG("Sgfa000",1)
IF MCC(608)<.1 EXIT
EXECIF 610,1,1 OUT=.01*((100-
MCC(608))*OUT+MCC(608)*MOOGG(VAR609,OUT,MCC605*FREQ,MCC606,MCC607))
EXECIF 610,0,0 OUT=.01*((100-
MCC(608))*OUT+MCC(608)*FILT(VAR609,OUT,MCC605*FREQ,MCC606))
DEBUG 1 // Revert to standard error logging
```

This will output the following, in the debug log:

```
***** Line 25, file C:\Documenti\Crescendo\MoogG.txt
Original instruction:
OUT = OSCG ( "Sgfa000" ; 1 )
Modified instruction:
$99857 = OSCG ( "Sgfa000" ; $00000 )
Actual operations to perform:
$99857=OSCG ("Sgfa000" $00000 )
***** End of instruction, line 25, file C:\Documenti\Crescendo\MoogG.txt

***** Line 26, file C:\Documenti\Crescendo\MoogG.txt
Original instruction:
IF MCC ( 608 ) < .1 EXIT

Parsing operand 1: "MCC ( 608 )"

Parsing operand 2: ".1"

Modified instruction:
IF $99391 < $00001 EXIT
***** End of instruction, line 26, file C:\Documenti\Crescendo\MoogG.txt

***** Line 27, file C:\Documenti\Crescendo\MoogG.txt
Original instruction:
EXECIF 610 ; 1 ; 1 OUT = .01 * ( ( 100 - MCC ( 608 ) ) * OUT + MCC ( 608 ) *
MOOGG ( VAR609 ; OUT ; MCC605 * FREQ ; MCC606 ; MCC607 ) )
Modified instruction:
$99857 = $00002 * ( ( $00003 - $99391 ) * $99857 + $99391 * MOOGG ( $99390 ;
$99857 ; $99394 * $99858 ; $99393 ; $99392 ) )
Actual operations to perform:
$00004=- ($00003 $99391 )
$00005=* ($00004 $99857 )
$00006=* ($99394 $99858 )
$00007=MOOGG ($99390 $99857 $00006 $99393 $99392 )
$00008=* ($99391 $00007 )
$00009=+ ($00005 $00008 )
$99857=* ($00002 $00009 )
***** End of instruction, line 27, file C:\Documenti\Crescendo\MoogG.txt

***** Line 28, file C:\Documenti\Crescendo\MoogG.txt
Original instruction:
EXECIF 610 ; 0 ; 0 OUT = .01 * ( ( 100 - MCC ( 608 ) ) * OUT + MCC ( 608 ) *
FILT ( VAR609 ; OUT ; MCC605 * FREQ ; MCC606 ) )
Modified instruction:
$99857 = $00002 * ( ( $00003 - $99391 ) * $99857 + $99391 * FILT ( $99390 ;
$99857 ; $99394 * $99858 ; $99393 ) )
Actual operations to perform:
$00010=- ($00003 $99391 )
$00011=* ($00010 $99857 )
$00012=* ($99394 $99858 )
$00013=FILT ($99390 $99857 $00012 $99393 )
$00014=* ($99391 $00013 )
$00015=+ ($00011 $00014 )
$99857=* ($00002 $00015 )
***** End of instruction, line 28, file C:\Documenti\Crescendo\MoogG.txt

*****
***** File "C:\Documenti\Crescendo\MoogG.txt"
***** COMPILATION OK: 0 ERRORS DETECTED. *****
*****
CRESCENDO INSTRUMENT, VERSION 1.0.171 RELEASE.
COPYRIGHT (C) 2020 - 2026 BY BJT2.
```

CRESCENDO IS FREEWARE. SEE LICENSE.TXT OR LICENSE.MD.
13775.383 MBYTES FREE.
READY.

Example B: Horizontal Log Layout

Forces the log window to appear below the controls, perfect for wide interfaces.

```
// Level 1, Horizontal layout (85), 10px border  
DEBUG 1, 85, 10
```

5. Developer's Notes

- **Performance:** High debug levels (3 and 4) generate a massive amount of text, which can slow down file loading. Use these levels only for specific code blocks.
- **Buffer Management:** The log window has a limited buffer. If you are debugging real-time MIDI, it is advisable to keep your debug sessions short or isolate one parameter at a time to prevent the window from overflowing.
- **RPN Verification:** Pay close attention to the "Actual operations to perform" section in the log; it shows the optimized order of operations Crescendo uses, which is vital for debugging nested parentheses.

QUALITY <samples>; <ww>; <type>;
<exponent>; <oversampling>;
<synthoversampling>, <filteroversampling>,
<sharpness>

The `QUALITY` instruction configures the windowed-sinc resampling engine. This system is responsible for avoiding aliasing and "metallic" artifacts when samples are played at different pitches. By adjusting these parameters, you can achieve anything from lo-fi "Nearest Neighbor" textures to laboratory-grade audio reconstruction.

1. Technical Syntax

```
QUALITY <samples>, <ww>, <type>, <exponent>, <oversampling>,  
<synthoversampling>, <filteroversampling>
```

- **<samples> (Table Resolution):** The number of points used to store the pre-calculated window.
 - **Default:** 4096 (Uses 16KB of L1 cache).
 - **Range:** 256 to 65536 (Should be a power of two).
- **<ww> (Window Width):** The number of sinc "lobes" to calculate.
 - Each output sample requires $2 \cdot ww + 1$ calculations.
 - **Higher values:** Better high-frequency retention and less aliasing, but higher CPU.
 - **Default: 7. Zero** means Nearest Neighbor (Lo-fi). This value is capped to $\text{<samples>} / 8 - 1$ to be sure that a lobe is at least 8 points, so to prevent aliasing in the table itself.

- **<type> (Window Function):** Selects the mathematical curve used to truncate the sinc function.
 - **0: Hamming (All-rounder) | 1: Hann | 2: Cosine | 3: Lanczos (Sharpest) | 4: Gauss | 5: None (Rectangular) | 6: Kaiser (Precision).**
 - Default: **Hamming (Type 0):** Recommended for most acoustic samples and musical instruments for its natural and smooth character.
 - The **Kaiser (Type 6)** is the High-precision filter. Best for pure synthetic tones, but may introduce "metallic" textures on percussive acoustic samples if not properly tuned.
- **<exponent> (Curve Warp):** Modifies the window shape.
 - **Higher (>1):** "Warmer" sound, less ringing, but loses some high-end.
 - **Lower (<1):** Brighter sound, but higher risk of ringing (Gibbs Phenomenon).
 - **If Type 6 (Kaiser):** This acts as the **Beta Coefficient**.
- **<oversampling>:** Number of sub-samples to average per output sample. Only for sampled data
 - **Range:** 1 to 20. (Not used by WAVETABLE, WAVESCAN, or GRAINSYNTH).
- **<synthoversampling>:** Number of sub-samples to average per output sample. Only for synth data (see Power oversampling below).
 - **Range:** 1 to 257. It is optional, default 10. The maximum value is not strictly necessary. Usually 16-32 it's enough.
- **<filteroversampling>:** Sets the internal oversampling rate for all filter modules (Standard and Moog, including those integrated in OSCG).
 - *Default: 1. Range: 1 to 16.*
 - **NOTE:** the oversampling is **NOT applied to the off-line pre-processing filters**.
- **<sharpness>:** This setting controls the alignment and steepness of the **48dB/octave** anti-aliasing filter used during the filter upsampling process. It defines how the synth balances high-frequency "air" against digital artifacts (aliasing).

IMPORTANT: This filter is active **only** when **<filteroversampling>** is set to 2 or higher. If **<filteroversampling>** is 1 (default), this parameter has no effect on the audio signal, and the signal path remains bit-perfect and unfiltered to preserve CPU and the original phase response.

 - **0 = Eco (Vintage) [Default].** The recommended setting for everyday use. Optimized for a warm, rolled-off sound reminiscent of classic analog hardware. By using a critically damped alignment ($Q=0.5$), it aggressively eliminates ultrasonic images, resulting in a smooth top-end that is very easy to mix.
 - **1 = Natural (Balanced).** It provides a subtle lift in clarity compared to Eco while maintaining a safe margin against aliasing. It mimics the natural frequency response of high-end 90s studio gear.
 - **2 = Sharp (High Fidelity).** Uses a **Butterworth** alignment to achieve a "maximally flat" frequency response. This mode preserves the full harmonic content of the oscillators up to the limits of human hearing (~18–19 kHz at 2x oversampling) with surgical precision.
 - **3 = Ultra (Extreme).** Pushes the filter's cutoff to the absolute theoretical limit. Designed for users working at high oversampling rates (8x or 16x) who want zero compromise on transient response and spectral detail. *Note: At lower oversampling rates, this may introduce audible aliasing on extremely high notes.*

Pro Tip: If you find your leads sounding too "fizzy" when using heavy Drive, switch to **Natural** or **Eco** to soften the harmonic bite. For crystalline pads and FM-like bells, **Sharp** is your best ally.

2. The Mathematical Foundation

Crescendo performs interpolation using a **Windowed Sinc function**, defined as:

$$\text{WindowedSinc}(X) = \text{sinc}(X) \cdot \text{Window}(X)^{\text{exponent}}$$

Where the core functions are:

- **Normalized Sinc:** $\text{sinc}(X) = \frac{\sin(\pi X)}{\pi X}$
- **Lanczos Kernel:** $L(X) = \text{sinc}\left(\frac{X}{ww}\right) = \frac{\sin(\pi X/ww)}{\pi X/ww}$
- **Modified Bessel Function (I_0):** Used for the Kaiser window, approximated via hyperbolic cosines.

Window Type Formulas ($\text{Window}(X)$)

ID	Type	Formula
0	Hamming	$\frac{25}{46} - \frac{21}{46} \cos\left(\pi\left(1 + \frac{X}{ww}\right)\right)$
1	Hann	$0.5 - 0.5 \cos\left(\pi\left(1 + \frac{X}{ww}\right)\right)$
2	Cosine	$\cos\left(\frac{\pi X}{2 \cdot ww}\right)$
3	Lanczos	$\frac{\sin(\pi X/ww)}{\pi X/ww}$
4	Gauss	$e^{-\left(\frac{X}{ww} \cdot \text{exponent}\right)^2}$
5	None	1.0 (Rectangular/Pure Sinc)
6	Kaiser	$\frac{I_0\left(\text{exponent} \cdot \sqrt{1 - (X/ww)^2}\right)}{I_0(\text{exponent})}$

3. The Role of the Exponent

The <exponent> parameter is a powerful "spectral shaper." It modifies the steepness and width of the windowing curve before it is applied to the sinc function.

- **Exponent > 1.0 (Pinching):**

The window becomes narrower and tapers to zero more aggressively. This significantly reduces **ringing artifacts** (Gibbs Phenomenon) but acts as a natural low-pass filter. High values result in a "warmer," darker sound.

- **Exponent < 1.0 (Flattening):**

The window stays "open" longer, approaching a Rectangular window (Type 5). This preserves maximum high-frequency content and brightness but increases the risk of audible "ringing" or "halos" around transients.

- **The Beta Case (Type 6):**

For the **Kaiser** window, the exponent is repurposed as the **Beta (β) parameter**. It controls the trade-off between the main-lobe width and side-lobe attenuation.

To achieve a sideband attenuation (**A**) in dB, use: $\beta = 0.1102 \cdot (A - 8.7)$
(for **A** > 50 dB).

4. Advanced Operational Logic

Double Precision & Phase Accuracy

Crescendo utilizes **64-bit Double Precision** for phase accumulation. Standard 32-bit floats lose precision after approximately 16 million samples, causing pitch drift in long samples. Using doubles ensures nanosecond-level accuracy regardless of sample length.

The "fww" factor

The fractional phase is converted to an index for the sinc table. The engine uses a scaling factor **fww**:

$$fww = \frac{NWSIN}{ww + 1}$$

This ensures the window always tapers perfectly to zero at the exact boundaries of the interpolation window ($X = \pm ww$), preventing "boundary clicks" during high-speed resampling.

Cache Alignment

The default table size (4096 samples / 16KB) is specifically chosen to reside within the **L1 Data Cache** of modern CPUs. This allows the inner loop (performing $2 \cdot ww + 1$ multiplications per sample) to run at the processor's maximum theoretical speed without waiting for L2 or L3 memory fetches.

Modern CPUs have at least 32KB of L1 cache. Limiting the table size to 4096 leaves 16KB of L1 cache for other use.

Listening sessions didn't find appreciable differences between table size of 1024 and 4096. Try yourself if you can lower the table size at least to 2048.

Sub-sample Precision

By converting the fractional phase to an index for the 4096-sample sinc table, the engine achieves a reconstruction accuracy that eliminates the "metallic" artifacts common in simpler linear or cubic interpolators.

Advanced Sinc Interpolation & Windowing Logic (Technical Notes)

The **QUALITY** instruction controls the windowed-sinc resampling engine. While nearest-neighbor (NN) interpolation is available ($_{ww=0}$), the engine's power lies in its multi-point sinc reconstruction, which is critical for avoiding aliasing when samples are transposed.

Window Types & Spectral Characteristics

Each window type offers a different trade-off between the **Main Lobe** (frequency preservation) and **Side Lobes** (aliasing/ringing):

- **Hamming & Hann (Types 0, 1):** These are the "all-rounders." They offer excellent side-lobe rejection, ideal for high-quality musical playback with minimal artifacts.
- **Lanczos (Type 3):** A "sinc-windowed sinc." It is often considered the best compromise for preserving high-end sharpness. It is highly recommended for percussive samples or bright lead sounds.
- **Kaiser (Type 6):** The most sophisticated option. It uses a high-order approximation of the **0th-order modified Bessel function** (I_0), allowing for surgical control over side-band attenuation via the Beta coefficient (mapped to the `<exponent>` parameter).
- **Pre-calculated Gain:** The division by the oversampling factor (`invovs`) is pre-calculated into the sinc table. This removes redundant division operations from the inner rendering loop, maintaining high performance even on legacy processors.

Oversampling

The `<oversampling>` parameter (up to 20x) provides a "brute-force" quality increase by averaging multiple sub-samples.

Strategic Usage: Increasing `ww` (Window Width) is generally more efficient than increasing `oversampling`. Use oversampling primarily when dealing with extreme down-pitching of samples containing significant high-frequency energy.

Power Oversampling & FIR Filtering

While standard sampled data interpolation often uses a factor of 1 to preserve CPU, synthetic oscillators (OSCG, SUPERSAW) require much higher resolution to avoid aliasing and digital distortion, especially for notes above C4.

Crescendo implements a **Power Oversampling** system that goes beyond simple linear averaging. Instead of just "smoothing" the sub-samples, it applies a **Windowed Sinc FIR Filter** during the decimation process. This ensures maximum high-frequency transparency (up to 22kHz) while aggressively suppressing the aliasing "ghost" frequencies that typically ruin high-pitched synth leads. The window used is the same window of the sampled data, with the same parameters, except `ww`, because the number of lobes comprised in the FIR filter are directly determined by the `<synthoversampling>` value.

Note: power oversampling is ALWAYS enabled on all synth oscillators, including OSCG, SUPERSAW and RENDER. To have a classic NN interpolation, just use `<synthoversampling> = 1`.

Advantages of FIR vs. Moving Average:

1. **Linear Phase Filtering:** By using the same windowing function (Hamming, Kaiser, etc.) of the sampled data, the Power Oversampling maintains perfect phase alignment.
2. **Zero Pass-band Droop:** Unlike simple averaging (Moving Average) often used in standard oversampling code, which starts muffling the sound around 10kHz, the FIR implementation stays flat up to the Nyquist limit.

3. **Dynamic Adaptation:** The reconstruction filter automatically re-calculates its coefficients whenever you change the `QUALITY` parameters, allowing you to choose between "Vintage" softness or "Precision" sharpness.

Filter Oversampling & Reconstruction Logic

The seventh parameter, `<filteroversampling> (N)`, introduces a high-fidelity processing stage for both linear and non-linear filters. While linear filters benefit from increased phase precision near the Nyquist frequency, non-linear filters (like the Moog model) use this headroom to manage complex harmonic distortion.

The algorithm follows a "**Reconstruction-First**" approach:

1. **Linear Upsampling & Anti-Imaging:** When $N > 1$, the input signal is upsampled via linear interpolation. To prevent the "staircase" artifacts of interpolation from aliasing inside the filter, a dedicated **48dB/octave Reconstruction Filter** (Mode 3) is applied at the upsampled rate. Its cutoff frequency is dynamically tuned via the internal `k` coefficient to balance brightness and ultrasonic rejection.
2. **Recursive Processing:** The filter's variables of state are updated N times per sample. For non-linear models, this allows the internal saturation (`sat()`) to generate harmonics in an extended frequency range (up to $N * 24$ kHz for 48 KHz sample rate), preventing them from folding back into the audible spectrum.
3. **Linear Filter Benefits:** For standard linear filters, oversampling eliminates the "frequency cramping" effect, ensuring that the filter's resonance and slope remain mathematically accurate even when the cutoff is set very high (e.g., >15 kHz).
4. **Decimation:** The sub-samples are accumulated and averaged. This "Box Filter" acting at the end of the chain serves as the final anti-aliasing stage, discarding any residual noise before the signal returns to the main buffer.

By setting this value to **1**, all filters operate in "Fast Mode," bypassing the reconstruction stage to save CPU cycles on older hardware. Higher values (**4-16**) are recommended for studio-grade rendering, particularly when using high resonance or heavy MOSFET drive.

Strategies to have maximum quality

For live processing it is best to have `<samples> 4096` if the L1 cache of your CPU is 32KB. For modern CPUs that have 48KB, you can also try 8192. But going over is risky for live use. If you want to export your project in your DAW having the maximum quality for all the Crescendo instances, even if it will be slow, just put in the `settings.ini`:

```
FEND QUALITY 65536, 8191, 6, 20, 1, 257
```

Or your preferred setting.

5. Practical Examples

Maximum Fidelity (Lanczos 12-point)

Ideal for critical sample rendering or bright lead sounds.

```
// Wide Lanczos window for high sharpness
```

```
QUALITY 4096, 12, 3, 1, 1
```

Ultra-Clean "Surgical" Attenuation (Kaiser)

Provides extreme rejection of aliasing artifacts.

```
// Beta = 7.85 provides ~80dB of side-lobe rejection
QUALITY 8192, 16, 6, 7.85, 1
```

The "Warm" Sampler (High Exponent)

Emulates the slightly darker, rolled-off sound of vintage high-end hardware samplers.

```
// Hamming window with a 2.5 exponent for soft high-end roll-off
QUALITY 4096, 8, 0, 2.5, 1
```

Set the QUALITY to nearest neighbor.

```
QUALITY 256, 0, 5, 1, 1
```

Set the QUALITY around SAMPLES instructions.

```
QUALITY 4096,31,0,1,4 // High quality setting
SAMPLES ...
...
SAMPLES ...
QUALITY 4096,7,0,1 // Lower quality settings for the runtime
```

INCLUDE "filename"

Use `INCLUDE` to manage large projects, share common sample libraries across multiple instruments, or organize complex sequencer logic into separate, reusable files.

1. Technical Syntax

```
INCLUDE "filename"
```

- **"filename"**: The path to the text file you wish to include.
 - **Default Directory**: Crescendo looks in <My Documents>\Crescendo by default.
 - **Paths**: Supports both **absolute paths** (e.g., "C:\Samples\Config.txt") and **relative paths** (e.g., "../Subfolder/Logic.txt").

2. Operational Logic

- **Section Constraint**: This instruction is only valid within the **COMMON** section.
- **Recursive Limit**: To prevent infinite loops (where File A includes File B, which includes File A), Crescendo limits the depth of inclusions to **4 levels**.
- **Compilation**: Included files are processed at the moment the main instrument file is loaded. Any syntax errors within an included file will be reported in the `DEBUG` log as if they occurred in the main file.

3. Practical Implementation

Why use INCLUDE?

1. **Global Templates:** Keep your `TEMPERAMENTS`, `DPIAWARE`, `QUALITY`, and `VSTVARS` settings in a single file and include it in every new instrument you build.
2. **Shared Sample Maps:** If you have a massive multi-sampled piano (hundreds of `SAMPLE` lines), you can keep that list in a standalone `.txt` file. This allows multiple instrument variations (e.g., "Piano_Dry" and "Piano_Reverb") to reference the same data.
3. **Sequencer Libraries:** Store your custom rhythm patterns or mathematical sequences in a "Library" folder and call them only when needed.

4. Examples

Example A: Organizing a Project

This setup keeps the main file clean by outsourcing the hardware configuration and the sound library.

```
// --- COMMON SECTION ---
INCLUDE "Global_Config.txt"    // Sets IO, QUALITY, and DPIAWARE
INCLUDE "Drum_Samples.txt"    // Contains 50+ SAMPLE declarations

LAYER OUT = OSCG("sine") * VAR(0)
```

Example B: Relative Paths

Assuming your main file is in `Crescendo/Instruments`, you can access files in a subfolder.

```
// --- COMMON SECTION ---
INCLUDE "Libraries/Standard_Envelopes.txt"
```

Developer's Note

When debugging an instrument that uses `INCLUDE`, the `DEBUG` log will specify which file a line belongs to. If you see an error at "Line 25, file C:...\\Library.txt", you know exactly which external module needs fixing.

MAXDLY <seconds>

The `MAXDLY` instruction defines the maximum duration in **seconds** available for every individual delay-based effect (e.g., `DELAY`, `REVERB`, `MOOGG`) in your instrument.

1. Technical Syntax

```
MAXDLY <seconds>
```

- **<seconds>:** The requested time-capacity for each effect instance.
 - **Default:** 40 seconds.

- **Range:** 0.01 to 1,000,000 seconds.
- **Internal Capping:** The resulting sample count is strictly capped between **262,144** (min) and **1,073,741,824** (max) samples.

2. Operational Logic

- **Section Constraint:** Must be placed exclusively in the **COMMON** section.
- **Sample Rate Scaling:** The buffer is defined in seconds to ensure your delay times remain consistent across different project sample rates.
- **Power-of-Two Rounding:** For maximum DSP speed (using bitwise masking instead of modulo), the internal sample count is always rounded up to the next **power of two**.
 - **The "Bonus" Effect:** Because of this rounding, the *actual* duration is often higher than the *requested* duration.
 - *Example:* At 48kHz, the default **40s** requires 1,920,000 samples. The engine rounds this up to 2,097,152 samples, resulting in an actual limit of **~43.7 seconds**.
- **Instance-Based Allocation:** Crescendo allocates a **dedicated buffer for every instance** of a delay function. If you have 10 layers each using a `DELAY`, 10 independent buffers are created in RAM.

3. Memory & Resource Management

Buffers are stereo (32-bit floats), requiring **8 bytes per sample**. Total RAM footprint is calculated as:

Total RAM = (Number of Delay Instances) × (Samples Required × 8 bytes)

RAM Usage & Timing (at 48kHz)

Requested Seconds Actual Samples (2n) Actual Duration RAM per Instance

0.01 to 5.46s	262,144 (2 ¹⁸)	~5.46s	2 MB
5.47 to 10.92s	524,288 (2 ¹⁹)	~10.92s	4 MB
10.93 to 21.84s	1,048,576 (2 ²⁰)	~21.84s	8 MB
40s (Default)	2,097,152 (2 ²¹)	~43.69s	16 MB
100s	8,388,608 (2 ²³)	~174.76s	64 MB
~1000s	67,108,864 (2 ²⁶)	~1,398s	512 MB

Note: If you double the sample rate (e.g., to 96kHz), the number of samples required to maintain the same time doubles, effectively doubling the RAM usage per instance.

4. Practical Examples

Example A: RAM-Efficient Layout

Setting a low `MAXDLY` ensures that multiple layers of short echoes don't waste memory.

```
// --- COMMON SECTION ---  
// Requesting 1 second.  
// At 48kHz, it rounds to ~5.4s, using the minimum 2MB per instance.  
MAXDLY 1
```

Example B: Long Ambient Looping

Ensuring enough memory for long, evolving feedback paths.

```
// --- COMMON SECTION ---  
// Requesting 100 seconds per instance.  
// At 48kHz, this jumps to the 8.3M sample bracket.  
// Actual duration: ~174 seconds. Memory cost: 64MB per instance.  
MAXDLY 100
```

5. Developer's Notes

- **Verification:** Use `DEBUG 2` to see the exact "Actual number of samples" and the resulting duration limit for your current environment.
- **Stability:** At 192kHz, the default 40s (rounded to 43s) requires ~8.3 million samples, costing 64MB per instance. Keep an eye on your total layer count to prevent DAW memory exhaustion.
- **The "Jump" Threshold:** Because of power-of-two rounding, increasing your `MAXDLY` by just one second might occasionally double the RAM usage of your entire instrument if it crosses a power-of-two boundary.
- **Stereo Buffers:** Memory is always calculated as `Samples * 2 (Stereo) * 4 (Bytes per Float)`.
- The effects are usually put in the POST step, so there is a single instance of buffer for each delay etc. It is advised to not use `DELAYs` in `LAYERS` if it can be used in the POST step. If you must use them in the `LAYERS` (e.g. see EKS example in the tutorials), then it's advisable to set a low `MAXDLY`.
- Special exceptions: `REVERB3` allocates a fixed buffer of 2.5MB, the other `REVERBs` allocate a fixed buffer of 1.125MB, the `CHORUS` and `FLANGER` instructions allocate a fixed buffer of 128KB. `PHASER` does not require a buffer.

UIMOD <UI item number>; <P1>; ... <P9>

`UIMOD` allows you to move, resize, and recolor the interface. This instruction assumes a baseline of **96 DPI** and an **8pt (16 DLP) font**. If a user's system uses higher DPI or larger fonts, all values are scaled automatically to maintain the layout's relative proportions.

1. Technical Syntax

UIMOD <item_index>, <P1>, <P2>, ... <P9>

- **<item_index>**: The ID of the UI element or category to modify.
- **<P1>...<P9>**: Parameter values specific to the item being modified.

2. Modification Categories

Category 0: Global Knob Defaults

Sets the "base" look for all knobs. Individual knobs (400–527) inherit these unless overridden.

UIMOD 0, <diam>, <thick>, <outer_style>, <inner_style>, <bkRGB>, <lineRGB>, <extRGB>

- **<diam>**: Diameter in scaled pixels (Default: 30).
 - **Negative Values**: Activates "Compact Mode." Draws only the knobs (closer together) and moves the parameter text to the tooltip.
 - **Value < -1000**: Resets to default diameter but allows setting other parameters.
 - **Maximum value**: 200 pixels (scaled).
- **<thick>**: Border thickness (0.1 pixel precision). < 0 uses default.
- **<outer_style>**: 0 = 11 ticks | 1 = Fat arc | 2 = Arc with 2 fat ticks | 3 = Arc with 9 sub-ticks.
- **<inner_style>**: 0 = Line indicator | >= 1 Circle indicator. The size is proportional to the fourth square root of this value.
- **<bkRGB>, <lineRGB>, <extRGB>** Default knob inner color (background), Default knob border and inner line/circle color, Default knob outer lines/circles color.

Category 1: Themes

UIMOD 1, <flag> (0 = Disabled, >= 1 Enabled).

Category 2: Title & File Name Customization

UIMOD 2, <x>, <y>, <font_size>, <weight>, <R>, <G>,

- Strips the path and extension from the filename to use it as a title.
- **Auto-Hide**: The upper bar is automatically hidden unless the mouse hovers over it.

Category 3: Anti-Aliasing (AA)

UIMOD 3, <level>

- Sets the smoothness of GUI rendering. Supported values: 1X, 4X, 9X, 16X. Out of range values are capped to the nearest supported value. No error is given, but only a warning on the log.

Category 4: GUI Background Customization UIMOD 4, <tilemode>, "filename"

- **<tilemode>**: Defines how the background image is rendered.
 - **0**: Uniform Background. Ignores the filename and uses the default background color (`bkgRGB`).
 - **1**: Stretched. Scales the image to fit the entire GUI area.
 - **2**: Top-Left. Places the image at the top-left corner, adjusted by the **current Zoom, Font, and DPI scaling factor**.
 - **3**: Centered. Centers the image within the GUI, adjusted by the **current Zoom, Font, and DPI scaling factor**.
 - **4**: Tiled (Fixed). Repeats the image at its native resolution (1:1). Texture grain remains fine regardless of zoom.
 - **5**: Tiled (Scaled). Repeats the image, but scales each tile according to the **current Zoom, Font, and DPI scaling factor**.
- **"filename"**: The path to the BMP image. The system validates the path during initialization using the following priority:
 1. **Absolute Path**: Checks if the provided path is valid.
 2. **Instrument Directory**: If not found, appends the path to the current Instrument folder.
 3. **Documents Folder**: If still not found, appends the path to `<My Documents>/Crescendo`.
- **Fallback**: If the filename cannot be resolved after these checks, `tilemode` is automatically set to **0** (Uniform Background).
- **NOTE**: Only BMP supported.

To achieve a professional GUI look, choose the **<tilemode>** that best suits your graphical assets:

- **Mode 1: Stretched (Full Coverage)**
 - **Best for**: High-resolution photographs, complex artistic paintings, or abstract gradients designed specifically for the plugin's dimensions.
 - **Pro Tip**: Use an image with a resolution close to your GUI size to avoid "blurring" when the system scales it up.
- **Mode 2: Top-Left (Corner Branding)**
 - **Best for**: Small logos, manufacturer watermarks, or decorative "corner" art.
 - **Pro Tip**: Since the rest of the GUI is filled with the solid `bkgRGB` color, ensure your image background either matches `bkgRGB` exactly or uses a clean border.
- **Mode 3: Centered (Focus Art)**
 - **Best for**: Central badges, circular logos, or specific background illustrations that should remain the focal point regardless of the window width.
 - **Pro Tip**: Great for "Vintage" style skins where a metal plate or emblem sits behind the main controls.
- **Mode 4: Tiled Fixed (Fine Textures)**
 - **Best for**: Realistic material surfaces like **Brushed Aluminum**, **Fine Leather**, **Noise/Grain**, or **Paper**.
 - **Pro Tip**: Because it remains 1:1, the texture "grain" stays sharp and realistic on 4K monitors, preventing the surface from looking "plasticky" or zoomed-in.
- **Mode 5: Tiled Scaled (Graphic Patterns)**
 - **Best for**: Geometric patterns like **Carbon Fiber**, **Honeycomb Grids**, **Dot Matrixes**, or **Diagonal Stripes**.
 - **Pro Tip**: Use this when the pattern is part of the "design" rather than a "material." It ensures that a carbon fiber weave grows in size proportionally as you zoom in on the knobs.

If your `filename` is missing or the path is invalid, the system defaults to **Mode 0 (Uniform Background)**. This ensures the plugin remains functional and readable using your defined `bkRGB` color even if the skin assets are moved or deleted.

Category 5: GUI Performance

UIMOD 5, <gui_rate_ms>, <knob_rate_ms>

- **<gui_rate_ms>**: Sets how often the general interface updates (in milliseconds).
 - **Default:** 1000 ms.
 - **Low values (15-100):** Real-time visual feedback for logs and status.
 - **Use a high value** (1000 or more) to make file loading and SoundFont importing much faster.
- **<knob_rate_ms>**: Refresh limit for **GUI knobs and DAW refresh**.
 - **Purpose:** Prevents internal instructions (e.g., `VAR0 = x`) from overloading the GUI and the host during complex modulations.
 - **Default:** 100 ms.
 - **Low values (15-30):** Ultra-smooth, analog-style knob movement.
 - **Set it higher** to save CPU on very old computers.

Category 6: Zoom Factor

UIMOD 6, <value>

- **Range:** 50 to 400 (represents percentage).
- **Purpose:** Sets the manual scaling factor for the GUI.

When to use it:

- **High-DPI Fix:** If your DAW lacks native DPI scaling (e.g., older versions of Ableton) and you are using a high-resolution monitor, add `UIMOD 6, 200` (or your preferred value) to your `settings.ini` to force a global size.
- **Per-File Setting:** Use it within a specific preset file to ensure it always opens with a custom layout size.

How it behaves:

- **Persistence:** If not specified, the current zoom level is preserved when loading new files.
- **DAW Projects:** Restoring a project in your DAW will always override the zoom to the exact state it was in when saved.
- **Per-Instance:** Each instance of Crescendo maintains its own independent zoom factor.

Category 100: Global Photo-Realistic Knob Skin

Sets a bitmap image as the default skin for all knobs. `UIMOD 100, <style>, "filename"`

- **<style>**: Defines how the vector indicator (pointer) is drawn over the image.
 - **< -1:** Disables the bitmap and reverts to **Flat Mode** (uses Category 0 settings).
 - **-1: Image Only.** No vector indicator is drawn (best if the indicator is already part of the bitmap).
 - **0: Image + Line.** Draws a vector line pointer.

- **>= 1: Image + Circle.** Draws a vector circle pointer. The size is proportional to the fourth square root of this value.
 - **Default: -2 (Flat Mode).**
- **"filename":** The path to the BMP image. If the file is missing, the system reverts to default **Flat Mode**.

Category 101–228: Specific Knob Skin Override

Overrides the bitmap skin for a specific VSTVAR knob (ID 0–127). `UIMOD <101+ID>, <style>, "filename"`

- **<style>:**
 - **<= -1001: Force Flat Mode.** Specifically forces this knob to be flat, even if a global image is set in Category 100.
 - **-1000 to -2: Inherit Global.** Uses the style and bitmap defined in Category 100.
 - **-1: Image Only (Local Override).**
 - **0: Image + Line (Local Override).**
 - **>= 1: Image + Circle (Local Override).**
 - **Default: -2 (Inherit Global).**
- **"filename":**
 - **Empty (""):** If a global image (Cat 100) exists, it uses the global bitmap but applies the **local style** settings.
 - **Invalid/Missing:** If both local and global filenames are invalid, the knob reverts to **Flat Mode**.
 - **Specific Path:** Loads a unique bitmap for this specific knob.

Implementation Guide for Photo-Realistic Knobs

1. Texture Inheritance Logic

The engine follows a strict priority to ensure the UI never "breaks":

1. **Local Skin:** If a valid BMP is defined for the specific knob ID (101-228).
2. **Global Skin:** If the local filename is empty or missing, but Category 100 has a valid BMP.
3. **Flat Fallback:** If no valid bitmaps are found, the engine uses the vector settings from Category 0 or 400+.
4. **Only circular knobs supported. If the image has the indicator, it must be in 12 o' clock position and the borders of the knob must touch the image borders.**

2. Advanced Usage Examples

Example A: Shared Texture, Different Styles

Use the "Bakelite" texture for all knobs, but the Master Knob (ID 0) uses a line while others use a small circle.

```
// Use brushed aluminum background
UIMOD 4, 1, "bg_brushed_alum.bmp"
// Global: Aluminum skin, circle indicator
UIMOD 100, 10, "knob_solid_alum.bmp"
// Override Knob 0: Use global texture but change style to Line (0)
UIMOD 101, 0, ""
```



Example B: Forcing Flat on Specific Controls

You have a global aluminum skin, but you want the "Fine Tune" knob to remain a simple vector drawing for clarity.

```
// Use brushed aluminum background
UIMOD 4, 1, "bg_brushed_alum.bmp"
// Global: Aluminum skin, line indicator
UIMOD 100, 0, "knob_solid_alum.bmp"
// Knob 5: Force Flat mode regardless of global setting
UIMOD 106, -1001, ""
```



Example C: Forcing another different image on Specific Controls

You have a global aluminum skin, but you want the "Master" knob to be a different image.

```
// Use brushed aluminum background
UIMOD 4, 1, "bg_brushed_alum.bmp"
// Global: Aluminum skin, line indicator
UIMOD 100, 0, "knob_solid_alum.bmp"
// Knob 0: Force bakelite image without indicator regardless of global setting
UIMOD 101, -1, "knob_bakelite_alum_cap.bmp"
```



Example D: Simple Reverb with the skins



Developer's Note on Anti-Aliasing

When using Category 100+, it is highly recommended to set **UIMOD 3, 16** (16X Anti-Aliasing). This ensures that the vector indicators (lines and circles) are blended smoothly over the bitmap texture, preventing "pixel crawling" during rotation. If performance is a concern on older CPUs, **AA 4X** provides a sharper, "retro" look similar to 90s hardware workstations.

Category 400–527: Specific Knob Configuration

Override global settings for a specific `VSTVAR` knob (ID 0–127).

```
UIMOD <400+ID>, <x>, <y>, <diam>, <thick>, <outer>, <inner>, <bkRGB>, <lineRGB>, <extRGB>
```

Category 600–708: Drop-down Menus position and size

- **600–608:** Labels for Temperament (0) and Keyswitches (1–8).
- **700–708:** Drop-down boxes for Temperament (0) and Keyswitches (1–8).

```
UIMOD <ID>, <x>, <y>, <width>
```

Category 709–712: Global System Colors

- **709:** Main Background | **710:** Default Text | **711:** Log Background | **712:** Log Text.

```
UIMOD <ID>, <R>, <G>, <B>
```

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

3. Operational Logic

- **Section Constraint:** Must be placed in the **COMMON** section.
- **Color Logic:** RGB values range from 0–255. If any RGB value is **negative**, the engine reverts that specific color to the factory default.
- **Coordinates:** (0,0) is the top-left corner. Knob labels use the font set by `SETFONT`, which may affect auto-positioning.
- **Limits:** Buttons and drop-down menu *colors* cannot currently be changed via `UIMOD`.

4. Practical Examples

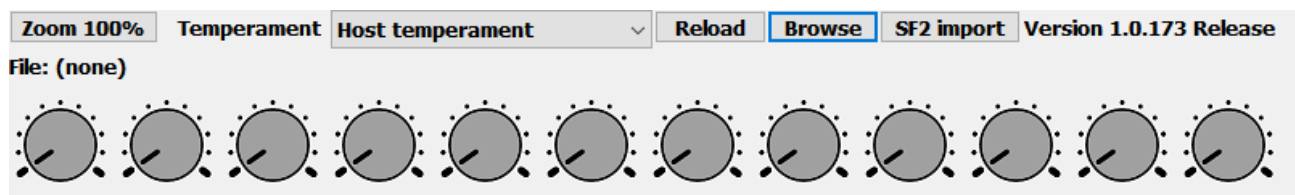
Example A: High-Definition Modern Interface

Sets a custom title, high anti-aliasing, and a blue-on-black background theme.

```
// --- COMMON SECTION ---  
// Customize Filename title: Pos(260,20), 48 DLP, Bold (900), White  
UIMOD 2, 260, 20, 48, 900, 255, 255, 255  
// Enable 16x Anti-Aliasing for smooth knob arcs  
UIMOD 3, 16  
// Custom Log Colors: Blue text on dark navy background  
UIMOD 711, 0, 0, 170  
UIMOD 712, 0, 136, 255
```

Example B: Activate Knob Compact mode

```
// --- COMMON SECTION ---  
// Compact mode, with 60 pixels Knob size.  
UIMOD 0, -60
```



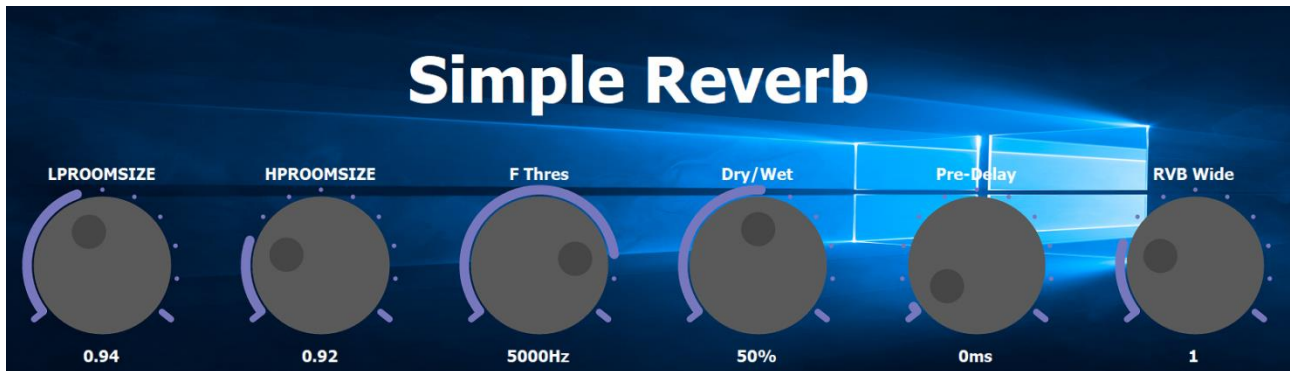
Example C: Custom Master vs. Utility Knobs

Overrides the look of Knob 0 to be a large "Master" while keeping Knob 2 as a tiny utility control.

```
// Master Knob (0): Pos(50,90), 70px diam, Fat Arc (3), Circle Ind (20)  
UIMOD 400, 50, 90, 70, 0, 3, 20, 30, 130, 30, 255, 255, 255  
// Utility Knob (2): Pos(350,110), 30px diam, 11 Ticks, Circle Ind (50)  
// Uses default BG colors (first 3 RGB values are -1)  
UIMOD 402, 350, 110, 30, 0, 0, 50, -1, -1, -1, 255, 0, 0
```

Example D: Custom background in SimpleReverb.txt


```
// Mode = 1: Stretch
UIMOD 4, 1, "<path to image>.bmp"
```



Example E: Customized UI.

```
// Set the default Knob diameter to 30 pixels
// Knob border 0 pixels; Scale style: continuos fat line with fat ticks
// Indicator style: circle with size 5
UIMOD 0,30,0,2,5
// File name customization at 260,20, 48 DLP, 900 weight, white.
UIMOD 2,260,20,48,900,255,255,255
// Set Antialiasing to 16X
UIMOD 3,16
// Knob 0, positioned at 50,90; diameter to 70 pixels; border 0 pixels
// Scale style: continuos fat line with fat ticks
// Indicator style: circle with size 20, Various RGB color
UIMOD 400,50,90,70,0,3,20,30,130,30,255,255,255,190,90,190
// Knob 1, positioned at 200,90; diameter to 70 pixels; border 2.5 pixels
// Scale style: 11 ticks; Indicator style: line, Various RGB color
UIMOD 401,200,90,70,3.5,0,0,130,30,30,150,150,0,0,80,0
// Knob 2, positioned at 250,110; diameter to 20 pixels; border 0 pixels
// Scale style: 11 ticks; Indicator style: circle with size 50, Various RGB
color, first and last at default
UIMOD 402,350,110,30,0,0,50,-1,-1,-1,255,0,0
// Knob 3, positioned at 490,110; diameter to 20 pixels; Knob border 0 pixels
// Scale style: continuos fat line with fat ticks
// Indicator style: circle with size 1, Default colors
UIMOD 403,490,110,30,0,2,1
// Knob 4, positioned at 620,110; diameter to 20 pixels; border 0 pixels
// Scale style: continuos fat line; Indicator style: line, Default colors
UIMOD 404,630,110,30,0,1,0
// Default colors
UIMOD 709,0,136,255
UIMOD 712,0,136,255
UIMOD 711,0,0,170
```

Developer's Note

If you use `INCLUDE` to pull in external scripts, they may contain their own `UIMOD` color settings. To ensure your main instrument looks correct, place a "Color Reset" at the end of your `COMMON` section using negative values: `UIMOD 709, -1, -1, -1`.

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

KNOB <knob number>; “format string”

Syntax

```
KNOB <index>, "[*]<parcel1>; <parcel2>; ..."
```

- **Index 0:** Global default for all knobs.
- **Index 1–128:** Local override for a specific control.
- **The String:** All configuration must be inside double quotes. Each instruction (parcel) must end with a semicolon ;. All parameters are separated by commas.

The `KNOB` instruction configures the visual appearance and positioning of the interface knobs. It can be used **Globally** (using index 0) to set defaults for all knobs, or **Locally** (using index 1–n) to override specific controls.

This command supersedes the `UIMOD` options 0, 100-128, 400-527. It is more flexible, supports floating point sizes, allows for independent color specifications and allows for chroma key transparency for knob images. These `UIMOD` options are still supported for compatibility and translated internally in the new algorithm.

Note: `KNOB` commands are not additive, unless the first character of the format string is an asterisk ‘*’. The last command issued for a specific index replaces all previous settings for that index. To reset a knob to its factory defaults, use: `KNOB x, ""`

1. Operation Modes: Flat vs. Image

The rendering engine operates in two primary modes: **Flat** (procedural vector) or **Image** (bitmap-based).

Global Context (`KNOB 0, ...`)

- **No specifier:** Defaults to **Flat Mode**. Borders and Indicators remain at factory defaults unless explicitly changed.
- **F (Flat):** Forces **Flat Mode**. Borders and Indicators are no longer "default" but are forced to procedural values.
- **I (Image):**
 - If the image is found: Activates **Image Mode**.
 - If the image is missing or the filename is empty: Activates **Flat Mode**.
 - In both cases, borders and indicators are forced to mode-specific behaviors.

Local Context (`KNOB x, ...`)

- **No specifier:** Inherits the Global setting (including the image).
- **F (Flat):** Forces **Flat Mode** for this specific knob.
- **I (Image):**

- If the image is found: Activates **Image Mode**. Border/Indicator are forced if a local override exists, otherwise they return to default.
- If the image is missing/empty: Inherits Global settings.

2. Logic of Inheritance (Borders & Indicators)

The relationship between Global and Local settings is governed by a state-dependent override system:

- **Pristine State:** If a Global value has never been modified, Local knobs use "Smart Defaults":
 - *Flat Mode:* Indicator = Auto Line; Border = Auto.
 - *Image Mode:* Indicator = OFF; Border = OFF.
- **Modified Global State:** If the Global value is touched (via **F**, **I**, or explicit **B/L** commands), Local knobs inherit the Global value unless a Local override is specified.
- **Forced Inheritance:** If you set a Global **I** or **F**, both Border and Indicator lose their "default" status. A Local knob subsequently set to **I** or **F** will inherit the Global values, not the factory defaults.

3. Parameter Definitions

Commands do not require a space after the character. Parameters are parsed starting from the first non-character value.

* - Do not reset flag

- Only allowed in the very first character, without following ;.
- Tells the parser to not reset the Knob data, allowing incremental definition.

Px,y - Position

- Sets the X and Y coordinates.
- **Automatic:** If x or y < 0. See SETFONT below for the explanation of the algorithm.
- **Manual:** If both x,y ≥ 0, automatic positioning is disabled.
- **Default:** Automatic positioning (triggered if x or y < 0).
- *Ignored for Global index (0).*

Dx - Diameter

- **Positive (x > 0):** Standard mode. Label is displayed below the knob. Diameter is x.
- **Negative (x < 0): Compact Mode.** The parameter label is hidden and moved to the tooltip. The diameter used is |x|.
- **Reset (x < -1000):** Resets to default diameter (30px).
- **Default:** +30px.

I<path> - Image Path

- Defines the absolute or relative path to a bitmap. Supports inheritance logic (see section 1).
- **Requirement:** Use **24-bit bitmaps**. **If the image has the indicator, it must be in 12 o'clock position and the borders of the knob must touch the image borders**

- **Transparency:** A circular mask is automatically applied. For custom shapes there is the **Chromakey** feature (replaces a single exact color). No alpha channel or color ranges supported.

Bx, r, g, b - Border

- **Global:**
 - $x < -0.1$: Automatic calculation (proportional to knob size).
 - $x \geq 0$: Sets thickness in pixels (values rounding to $< 1\text{px}$ are not drawn).
- **Local:**
 - $x < -1000$: Use Global setting.
 - $-1000 \leq x < -0.1$: Use automatic thickness.
 - $x \geq 0$: Set thickness in pixels.

Lx, r, g, b - Indicator (Line/Circle)

- **Global:**
 - $-1000 \leq x \leq -100$: Line (Auto thickness).
 - $-100 < x \leq -0.2$: Line (Thickness = $|x|$).
 - $0.2 \leq x \leq 100$: Circle (Diameter = x).
 - $x > 100$: Circle (Auto diameter).
 - $|x| < 0.2$ or $x < -1000$: Disable Indicator.
- **Local:**
 - $x < -1000$: Use Global setting.
 - Other values follow the Global logic.

Cr, g, b - Chromakey Color

- Sets the transparent color for **Chromakey Image Mode**.
- **Default:** Disabled.
- **Local Disable:** If enabled Globally, disable for specific knobs by setting r, g, or b < 0 .
- **Local Color:** even if disabled globally, you can set locally different transparent color.

Tx, r, g, b - Tick Style

Value	Style Name	Description
≤ 0	11 Ticks	Standard layout with 11 radial tick marks (default).
1	Fat Arc	A thick, continuous circular arc.
2	Fat Arc plus fat ticks	A thick, continuous circular arc with 2 emphasized "fat" ticks at the start and end.
3	Fat Arc, plus ticks and fat ticks	A thick, continuous circular arc with 2 emphasized "fat" ticks at the start and end with 9 smaller sub-ticks for finer resolution.

Value Style Name	Description
-1 Inherit	(Local only) Uses the Global T_x setting.

F_r, g, b - Flat Mode

- Forces procedural rendering (flat mode) and sets the inner knob color. Follows inheritance logic (see section 1).

Manual Entry: The Inheritance Hierarchy (Simplified)

To help users understand the "Default vs. Forced" logic:

- Level 1: Factory Defaults**
 - Flat knobs get Auto-Borders and Auto-Line Indicators.
 - Image knobs get NO Borders and NO Indicators.
- Level 2: Global Override ($KNOB\ 0$)**
 - If you set a Global F or I , all local knobs now look to the Global settings as their new baseline.
 - Example: If you set a Global Border $B2, 0, 0, 0$, every local knob will have a 2px black border unless you explicitly change it locally.
- Level 3: Local Override ($KNOB\ x$)**
 - Local settings always win.
 - If you want a local knob to ignore Global "forcing" and go back to a standard behavior, use the specific "auto" or "off" values (e.g., $x < -1000$ for borders to reset to global, or specific values to turn them off).

Pro-Tip:

- **Compact Mode ($D < 0$):** Use this when designing high-density interfaces or 'hidden' controls. By setting a negative diameter (e.g., $D-40$), the knob will occupy 40 pixels of space, but the text label will only appear when the user hovers the mouse over the control, keeping the GUI clean and focused.
- When using **Image Mode**, remember that the filename path (parameter I) does not require a leading space. The parser ignores all characters until the first valid path character. If an image is not found, the knob will automatically fallback to **Flat Mode** to ensure the UI remains functional.

Incremental KNOB Configuration

While standard $KNOB$ commands are destructive—meaning the most recent instruction for a specific index completely replaces all prior settings—the engine provides a specialized mechanism for **incremental definition**.

The "Do Not Reset" Flag (*)

To define a knob's properties across multiple instructions or files without wiping existing data, you must use the asterisk (*) prefix.

- **Syntax:** `KNOB <index>, "[*]<parcel1>; <parcel2>; ..."`
- **Operational Logic:** When the first character of the format string is an asterisk, the parser is instructed **not to reset** the existing knob data. This allows you to "layer" parcels incrementally.
- **Default Behavior (No Asterisk):** Without the * flag, a new `KNOB` command for an index (e.g., Knob 1) restores that specific control to factory defaults before applying the new parameters.

Strategic Applications for Incremental Definition

Incremental definition is particularly useful when managing complex instrument files or when utilizing the `INCLUDE` instruction to pull in global UI templates.

- **Modular Layouts:** You can define a global aesthetic in a `Settings.ini` or header file and then use incremental commands in specific instrument scripts to adjust only the physical coordinates.
- **Isolated Parameter Tuning:** You can change a single attribute—such as the position (`Px,y;`) or diameter (`Dx;`)—without needing to re-specify the image path, border thickness, or tick styles.
- **Dynamic Skinning:** You may set a global image skin for all knobs at once and then use incremental local commands to add unique indicators (line or circle) to specific controls.

Parcel Components for Incremental Building

When using the * flag, you can add or modify any of the following "parcels" individually:

Parcel	Function	Technical Detail
<code>Px,y;</code>	Position	Sets X/Y coordinates. Both must be <code>\$\ge\$ 0</code> to disable auto-positioning.
<code>Dx;</code>	Diameter	Sets size in pixels. Negative values enable Compact Mode (label in tooltip).
<code>I<path>;</code>	Image Path	Loads a 24-bit BMP skin. Supports circular masking and Chromakey.
<code>Bx,r,g,b;</code>	Border	Sets thickness and RGB color. <code>\$x < -1000\$</code> inherits global settings.
<code>Lx,r,g,b;</code>	Indicator	Sets line (<code>\$x < 0\$</code>) or circle (<code>\$x > 0\$</code>) pointer style and color.
<code>Tx,r,g,b;</code>	Tick Style	Chooses between 11 ticks (0), Fat Arcs (1-2), or Sub-ticks (3).
<code>Cx,r,g,b;</code>	Chromakey	Defines the exact RGB color to be treated as transparent for the skin.

Practical Implementation Example

To build a "Master Volume" knob incrementally, you might structure your script as follows:

1. **Define Global Aesthetic (No asterisk):** `KNOB 0, "F200,200,200; D40; T1; B2,0,0,0;"` (*Sets all knobs to be flat grey, 40px wide, with a Fat Arc and 2px black border*).
2. **Assign Local Position:** `KNOB 1, "P100,150;"` (*Moves Knob 1 to specific coordinates, resetting all the parameters to default, that is to inherit global settings*).

3. **Add Unique Indicator (With asterisk):** `KNOB 1, "*L10,255,0,0;"` (*Adds a red circle indicator to Knob 1 without resetting its data, and in particular without moving it or changing its size*).

Developer's Administrative Notes

- **Placement:** The * flag must be the **very first character** within the double quotes.
- **Resetting:** To completely clear a knob's data and return it to factory defaults (e.g., to "undo" an incremental chain), issue a command with an empty string: `KNOB index, ""`.
- **Efficiency:** The `KNOB` instruction is significantly more efficient than legacy `UIMOD` commands because it only updates the specific parcels provided rather than requiring a long, comma-separated string of every possible value.

4. Practical Examples (New Syntax)

Example A: Professional Flat Interface

Clean vector look with specific tick styles.

```
// Set Global Flat style, 40px diameter, Fat Arc ticks (Style 1)
KNOB 0, "F;D40;T1"

// Local: Knob 5 is a "Fine Tune" - make it tiny and use 11 ticks
KNOB 5, "F;D30;T0"
```

Example B: Global Image with Local Indicator Override

Uses a global aluminum skin, but Knob 5 uses a specific red line indicator.

```
KNOB 0, "Iknob_alum.bmp; L0;"
KNOB 5, "L-2,255,0,0;"
```

Example C: Manual Positioning and Custom Flat Style

Sets Knob 1 to a specific coordinate with a large diameter and a circle indicator.

```
KNOB 1, "F200,200,200; P150,100; D60; L10,255,255,0; T2;"
```

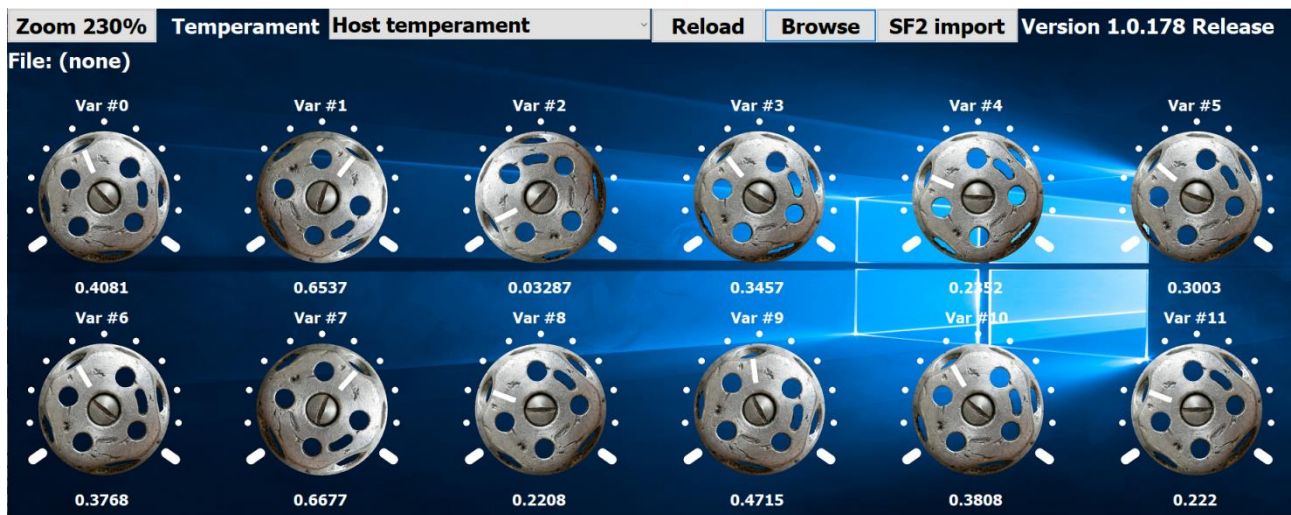
Example D: Compact Mode for specific knobs

Enables compact mode (label in tooltip) for the first 4 knobs, with increasing size.

```
KNOB 1, "D-30;"
KNOB 2, "D-50;"
KNOB 3, "D-70;"
KNOB 4, "D-100;"
```

Example E: Chroma key (transparency)

```
UIMOD 4, 1, "win10background.bmp"
KNOB 0, "D100;Ialuminum_holes.bmp;C0,0,0;T0,255,255,255"
```



Important Developer Notes

- If you are updating an old script:
 - Replace `UIMOD 100` with `KNOB 0`.
 - Replace `UIMOD 400+ID` with `KNOB ID+1`.
 - The `KNOB` instruction is significantly more efficient as it allows to specify only some parameters thanks to the “parcels” and moreover supports floating point values for some parameters.

Migration: From UIMOD to KNOB

The new `KNOB` syntax is more modular. Instead of remembering a long sequence of commas where every value must be present, you now only specify the "parcels" you want to change.

1. Mapping Table (Flat Knobs)

If you are converting a `UIMOD 400` (Local) or `UIMOD 0` (Global) command:

UIMOD Parameter	Old Position	New KNOB Parcel	Note
<code><x>, <y></code>	P1, P2	Px,y;	Use negative values for Auto-position.
<code><diam></code>	P3	Dx;	<code><diam> <0</code> for Compact Mode.
<code><thick></code>	P4	Bx;	Border thickness.
<code><outer_style></code>	P5	Tx;	Tick style (0, 1, 2, or 3).

UIMOD Parameter	Old Position	New KNOB Parcel	Note
<inner_style>	P6	Lx;	Indicator (<0 Line, >0 Circle). More flexible: old UIMOD had only auto line width and limited circle diameter.
<bkRGB>	P7, P8, P9	Fr,g,b;	Flat background color.
<lineRGB>	P10, P11, P12	L,r,g,b;	Indicator color (inside L parcel).
<extRGB>	P13, P14, P15	T,r,g,b;	Tick color (inside T parcel).

2. Conversion Examples

Example 1: Global Settings

Old UIMOD:

```
UIMOD 0, 40, 2, 1, 0, 50, 50, 50
```

(40px diam, 2px border, Fat Arc, Line Indicator, Dark Grey BG)

New KNOB:

```
KNOB 0, "D40; B2; T1; L-1; F50,50,50;"
```

Example 2: Local Specific Knob (ID 0)

Old UIMOD:

```
UIMOD 400, 100, 150, 60, 0, 3, 20, -1, -1, -1, 255, 255, 255
```

(Pos 100,150, 60px diam, No Border, Sub-ticks, Circle Ind 20, Default BG, White Ind)

New KNOB:

```
KNOB 1, "P100,150; D60; B0; T3; L20,255,255,255;"
```

3. Mapping Table (Image Knobs)

If you are converting UIMOD 100 (Global) or UIMOD 101+ (Local):

UIMOD Parameter Old Position New KNOB Parcel

<style> P1 **Lx;** (Style converted to L parcel)

"filename" P2 **I"filename";**

Old UIMOD:

```
UIMOD 101, 0, "knob_vint.bmp"
```

(Knob 0, Image+Line style, path)

New KNOB:

```
KNOB 1, "Iknob_vint.bmp; L-1;"
```

Key Advantages of Migration

- **Conciseness:** You no longer need to type `-1, -1, -1` to keep default colors. If you don't include the `F`, `B`, or `L` color parameters, the engine automatically keeps the inherited/default colors.
- **Flexibility:** You can change just the position with `P10,10;` without touching the diameter or skin.
- **Readability:** Parcels like `D60;` (Diameter 60) are much easier to debug than counting commas in a `UIMOD` string.

Final Note: The legacy `UIMOD` commands for Categories 0, 100, and 400+ will still work to ensure backward compatibility with your existing instrument library, but `KNOB` is required for all new features like Chromakey and advanced Unicode label alignment.

Migration of the Examples of the UIMOD section:

Example A: Shared Texture, Different Styles

Use a global aluminum skin, but set Knob 0 (Master) to a line while others use a circle.

```
// Use brushed aluminum background (Remains UIMOD)
UIMOD 4, 1, "bg_brushed_alum.bmp"
// Global: Aluminum skin, circle indicator (size 10)
KNOB 0, "Iknob_solid_alum.bmp; L10;"
// Override Knob 0: Inherit global texture, but change style to Line (-2)
KNOB 1, "L-2;"
```

Example B Forcing Flat on Specific Controls

Global aluminum skin, but Knob 5 (Fine Tune) remains a vector drawing.

```
UIMOD 4, 1, "bg_brushed_alum.bmp"
```

```
// Global: Aluminum skin, line indicator
KNOB 0, "Iknob_solid_alum.bmp; L-2;"
// Knob 5: Force Flat mode (using F parcel, default color)
KNOB 6, "F;"
```

Example C: Forcing a different image on Specific Controls

Global aluminum, but Master knob (0) uses a unique Bakelite image.

```
UIMOD 4, 1, "bg_brushed_alum.bmp"
// Global: Aluminum skin, line indicator
KNOB 0, "Iknob_solid_alum.bmp; L-2;"
// Knob 0: New image without indicator
KNOB 1, "Iknob_bakelite_alum_cap.bmp; L0;"
```

Example D: High-Definition Modern Interface

Environment settings remain UIMOD.

```
// --- COMMON SECTION ---
// Customize Filename title (UIMOD Category 2)
UIMOD 2, 260, 20, 48, 900, 255, 255, 255
// Enable 16x Anti-Aliasing (UIMOD Category 3)
UIMOD 3, 16

// Custom Log Colors (UIMOD Categories 711, 712)
UIMOD 711, 0, 0, 170
UIMOD 712, 0, 136, 255
```

Example E: Activate Knob Compact Mode

Sets knobs to Compact Mode with a visual diameter of 60 pixels.

```
// Global: Diameter -60
// (Negative = Compact Mode enabled, 60 = visual size)
KNOB 0, "D-60;"
```

Example F: Custom Master vs. Utility Knobs

Manual positioning and specific styles.

```
// Master Knob (0): Pos(50,90), 70px diam, Fat Arc (T2), Circle Ind (L20)
// Color provided for BG (F) and Indicator (L)
KNOB 1, "P50,90; D70; T2,255,255,255; L20,30,130,30; F255,255,255; B0;"

// Utility Knob (2): Pos(350,110), 30px diam, 11 Ticks (T0), Circle Ind (L50)
// Uses default BG colors (we omit the color in F)
KNOB 3, "P350,110; D30; T0,255,0,0; L50; B0;"
```

Example G: Custom Background

Remains UIMOD.

```
UIMOD 4, 1, "<path to image>.bmp"
```

Example H: Customized UI (Complex Setup)

Reflecting the specific diameters and styles.

```
// --- COMMON SECTION ---

// Global Default: 30px, No border, Arc Style 2, Circle Ind 5
KNOB 0, "D30; B0; T2; L5;"

// Environment (Remains UIMOD)
UIMOD 2, 260, 20, 48, 900, 255, 255, 255
UIMOD 3, 16

// Knob 0: Master Pos, 70px diameter, Sub-ticks (T3), Circle Ind 20, Custom
Colors
KNOB 1, "P50,90; D70; B0; T3,190,90,190; L20,30,130,30; F255,255,255;"

// Knob 1: Pos, 70px diameter, 3.5px border, Standard Ticks (T0), Line Ind,
Custom Colors
KNOB 2, "P200,90; D70; B3.5; T0,0,80,0; L-1,130,30,30; F150,150,0;"

// Knob 2: Pos, 30px diameter, Standard Ticks, Circle Ind 50, Red color
// Note: B0 is inherited from Global KNOB 0
KNOB 3, "P350,110; D30; T0; L50,255,0,0;"

// Knob 3: Pos, 30px diameter, Arc Style 2, Circle Ind 1
KNOB 4, "P490,110; D30; T2; L1;"

// Knob 4: Pos, 30px diameter, Fat Arc style, Line indicator
KNOB 5, "P630,110; D30; T1; L-1;"

// System Environment (Remains UIMOD)
UIMOD 709, 0, 136, 255
UIMOD 712, 0, 136, 255
UIMOD 711, 0, 0, 170
```

Key Takeaways for your Documentation:

- **Index Offset:** Remember that `UIMOD 400` refers to Knob ID 0, which is `KNOB 1` in the new syntax (Index 0 is always Global).
- **Color Inclusion:** In `KNOB`, colors are part of the specific parcel (e.g., `Lsize,r,g,b`). If you omit the colors, the engine uses the default or inherited ones.
- **Cleanliness:** Notice how Example H is much more readable now—you can clearly see which knob is being moved (`P`) or resized (`D`) without counting dozens of commas.

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

`SETFONT` defines the typeface and scale for the entire instrument. Because Crescendo uses **DLP (Device Logical Pixels)**, font heights are specified in "screen units," where **1 unit equals 0.5 pixels** (at 96 DPI).

1. Technical Syntax

SETFONT "<font_name>", <height>, <weight>, <knob_height>, <knob_weight>

- "<font_name>": The name of a TrueType font installed on the system (e.g., "Arial", "Verdana").
 - *Note: OEM-only fonts like "Modern" or "Script" are not supported.*
- <height>: Global text height for buttons and drop-down labels (in 0.5px units).
 - **Default:** 16 (8 pixels). **Maximum:** 48.
 - **Negative Values:** Specifies height *excluding* internal leading. This results in a larger visual glyph for the same numerical value.
- <weight>: Global thickness (0–900). 0 is "don't care," 100 is thin, 900 is extra bold.
- <knob_height>: Specific height for knob labels (names and values). Defaults to global <height>.
- <knob_weight>: Specific weight for knob labels. Defaults to global <weight>.

2. Operational Logic & Autopositioning

Crescendo features an intelligent grid system. The engine calculates the optimal distribution of knobs based on the font sizes provided in this instruction.

- **The "Flow" Pool:** By default, all knobs belong to an automated "pool." The engine spaces them perfectly based on their label size and diameter.
- **Manual vs. Automatic:** When you manually position a knob using `UIMOD`, you remove it from this pool.
 - **Sliding Logic:** When a knob is manually positioned, the next automatic knob "slides" into the next available spot in the grid. No "hole" is left behind.
- **Responsibility:** The engine calculates the minimum `INTERFACE` size based only on the knobs remaining in the automatic pool. If you position a knob manually, you must ensure it sits within the interface boundaries and does not overlap other elements.

3. Practical Examples

Example A: High-Visibility Layout

Using a larger font for knob values ensures they are readable even on smaller laptop screens.

```
// --- COMMON SECTION ---  
// Global font: Tahoma, 8px (16 units)  
// Knob labels: Tahoma, 12px (24 units), Bold (900)  
// The engine will widen the grid spacing to accommodate the larger labels.  
SETFONT "Tahoma", 16, 400, 24, 900
```

Example B: Compact Typography

Using thin fonts and default scaling for a dense, control-heavy interface.

```
// --- COMMON SECTION ---  
SETFONT "Segoe UI", 16, 400, 16, 400
```

4. Summary of Logic

| Feature | Behavior |
|---------------------------|---|
| Grid Calculation | Based on <code>height</code> and <code>knob_height</code> . |
| Manual Positioning | Removes the knob from the automatic grid calculation. |
| Interface Sizing | Calculated only based on the "Automatic Pool" of knobs. |
| Scaling | Fully DPI-aware (e.g., 192 DPI multiplies all units by 2). |

Developer's Note

Think of `SETFONT` as your layout designer. If your interface feels too cramped, simply increasing the `knob_height` by a few units will force the engine to "spread out" the automatic knobs, often fixing the layout without requiring dozens of manual `UIMOD` coordinates.

HIDEUI <option>

The `HIDEUI` instruction allows you to selectively hide system information (Bank, Program, BPM, etc.) to reclaim screen real estate. This is essential for creating custom skins where you want the focus to be on your `VSTVAR` knobs rather than standard MIDI telemetry.

1. Technical Syntax

`HIDEUI <option>`

- **<option>**: An integer (0–10) determining the visibility level.
- **Default**: 4 (All hidden).

2. Visibility Options

| Option | Visible Elements | Use Case |
|----------|---|-----------------------------------|
| 0 | Everything (Bank, Program, Time, BPM, Poly) | Full MIDI/Transport monitoring. |
| 1 | Time, BPM, Polyphony | Hides MIDI Bank/Program clutter. |
| 2 | BPM, Polyphony | Standard for tempo-synced synths. |

| Option Visible Elements | | Use Case |
|-------------------------|---|---|
| 3 | Polyphony only | Clean, performance-oriented layout. |
| 4 | Nothing (All Hidden) | Maximum space for custom knobs. |
| 5 | Program Name (Placed left of File Name) | Compact multi-timbral view. |
| 6 | Program Name + Polyphony | Informative performance view. |
| 7 | Automatic | Logic based on Debug and Sequencer status. |
| 8 | Polyphony (Placed left of File Name) | Minimalist resource monitoring. |
| 9 | Automatic | Switches between 4 or 10 if Programs are defined. |
| 10 | 16-Channel Program List + Pedals | Advanced orchestral/multi-channel setups. |

3. Performance & Layout Critical Notes

The "+P" Post-Processing Indicator

When polyphony is visible (Options 0–3, 6, 8, 10), the count may appear as 24 +P.

- **Always Active:** The +P appears if a `POST` step is defined in the instrument.
- **Constant CPU Load:** Unlike voices, which are allocated dynamically, the `POST` step is an architectural constant. If defined, it runs every single sample cycle regardless of whether notes are being played.
- **Optimization Tip:** To minimize the CPU impact of a constant `POST` step, developers often use an "if-exit" trick (checking if the Dry/Wet of some effect is high enough to be audible) to skip heavy calculations when an effect is disabled or negligible.

Fixed Position Overlap (UIMOD 2)

When using `UIMOD 2` to customize the title, the file name is anchored to a **fixed position** and does not slide to make room for other status elements.

- **Overlap Risk:** If you use a large title font or enable options that place text near the filename (like **5**, **6**, or **8**), the title may overlap the Program or Polyphony text.
- **Manual Layout:** If you customize the title, it is strongly advised to manually position your knobs (using `UIMOD 400+`) to ensure they don't collide with the fixed-position filename or the expanded status info.

4. Practical Examples

Example A: High-Performance Monitoring

```
// --- COMMON SECTION ---
UIMOD 2, 250, 20, 48, 900, 255, 255, 255 // Large title
HIDEUI 8 // Show Polyphony (and +P if POST is defined)
// Manually move knobs down to clear the large title area
UIMOD 400, 50, 150, 60, -1, 0, 0
```

Example B: The Multi-Channel Matrix

```
// --- COMMON SECTION ---
HIDEUI 10 // Displays all 16 MIDI channels and sustain pedal status
```

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

SAMPLEUI creates a small "browser" block in the instrument interface. It includes the current filename and a "Browse" button, and it supports Drag & Drop directly from the OS file explorer.

1. Technical Syntax

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

- **<Sample_slot>**: The ID of the sample slot (defined via the SAMPLE instruction) that this UI will control.
 - **Warning**: Assigning multiple SAMPLEUI elements to the same slot leads to **UNDEFINED** behavior.
- **<X_pos>, <Y_pos>**: The coordinates for the upper-left corner of the UI element.
- **"<Text>"**: A single line of descriptive text (e.g., "Load Kick Drum") displayed in the header of the UI block.

2. Operational Logic

- **Section Constraint**: Must be placed in the **COMMON** section.
- **Dynamic Loading**: When a user selects a new file via this UI, the engine immediately replaces the data in the assigned slot.
- **Plain Loading**: Any pre-processing (like prefiltering or panning) defined in the original script for that slot is **ignored** when a user swaps the file; the plain audio file is loaded directly.
- **Sizing**: The vertical height of the UI is fixed, while the horizontal width scales automatically based on the length of the filename and the header text.
- **Slot Requirement**: This should only be used with slots containing sampled data. If used on a "Synth" slot (e.g., a slot generating a wavetable), results are **UNDEFINED**.

3. Practical Example

Creating a "User Kick" Loader

In this example, we define a sample slot for a kick drum and then provide a UI for the user to change it.

```
// --- COMMON SECTION ---

// 1. Define the initial sample slot (Slot #5)
SAMPLE 5, "DefaultKick.wav", .... OTHER PARAMETERS

// 2. Create the UI to allow the user to swap Slot #5
// Positioned at (50, 200) with a custom header
SAMPLEUI 5, 50, 200, "User Kick Drum"

// 3. Optional: Customize the look of the rest of the UI
UIMOD 2, 260, 20, 32, 700, 255, 255, 255
```

4. Developer's Summary Table

| Feature | Detail |
|---------|--------|
|---------|--------|

| | |
|----------------------|---------------------------------|
| Input Methods | "Browse" button or Drag & Drop. |
|----------------------|---------------------------------|

| | |
|--------------------|---|
| Coordinates | (0,0) is top-left; scaled by DPI and Font settings. |
|--------------------|---|

| | |
|---------------|--------------------------|
| Header | Single row of text only. |
|---------------|--------------------------|

| | |
|--------------------|---|
| Limitations | No pre-processing; Sample data only (no synth slots). |
|--------------------|---|

Developer's Note

Because the horizontal size of `SAMPLEUI` is dynamic, a very long file path can "stretch" the UI and potentially overlap other knobs or elements. It is best to place `SAMPLEUI` elements in an area of your `INTERFACE` with plenty of horizontal clearance, or use it in conjunction with manual knob positioning (`UIMOD 400+`).

Branding and Professional Deployment

Crescendo features a "**Branding Mode**" that allows you to distribute plugins under your own brand.

1. Triggering Branding Mode

Branding Mode is automatically activated if the DLL filename does **not** start with "Crescendo". If the name is changed from the default, the engine hides its internal identity.

2. Plugin Type & Category (VSTi / VST)

The engine determines the plugin category based on the DLL filename suffixes:

- **Instrument (VSTi):** Default behavior. If no suffix is used, the plugin identifies as a Virtual Instrument.
- **Effect (VST):** You must include `_effect` or `-effect` in the filename (e.g., `MyFilter_x64_effect.dll`).
- **MIDI Effect:** You must include `_midi` or `-midi` in the filename (e.g., `MyArp_x64_midi.dll`).

3. Automatic Folder & Resource Mapping

The engine identifies its resource folder and main script by truncating the DLL name at the first underscore `_`, hyphen `-`, or dot `.`

- **Multi-Build Support:** `MySynth_AVX_effect.dll` and `MySynth_SSE_effect.dll` will both look for a folder named `MySynth/` and load the script `MySynth.txt`.
- **First Run:** If the resource folder is missing, the plugin creates it and generates a template `settings.ini` containing a pre-filled `BRANDING` instruction example.

4. Identity via `settings.ini`

To define your commercial identity, use the `BRANDING` instruction within `settings.ini`.

Important: For stability, `BRANDING` is only updated via `settings.ini`. If found in the `.txt` script, it is checked for syntax but otherwise ignored.

BRANDING "Product Name", "Vendor Name", "Support Contact", "Unique ID"

- **Product Name:** Your commercial name (e.g., "IronDistortion").
- **Vendor Name:** Your company name (e.g., "Awesome Inc.").
- **Support Contact:** An email or web URL displayed in the debug log during critical errors.
- **Unique ID:** A 4-character FOURCC code (e.g., "AWEP"). **Crucial:** Use a unique ID to ensure the DAW treats your product as a distinct plugin.

Note: The Product Name is automatically suffixed with the plugin type (-instrument, -effect, or -midi) and the engine version to maintain technical transparency.

5. UI Customization

When Branding Mode is active, the interface adapts for a professional "standalone" look, minimizing technical clutter:

- **Auto-Hide:** The upper technical bar is simplified and auto-hidden. It remains accessible only for essential controls like **Zoom** and the **Product Name** display.
- **Default Display:** The Product Name is automatically centered and rendered at double size in the main area by default.

- **Refinement:** Use the `UIMOD 2, x, y...` instruction to customize the position, color, and font style of this main label.
- **Type Detection:** If the DLL filename includes `_effect` or `_midi`, the **Temperament Combo Box** is automatically hidden. It is recommended to set `temperament "host"` in your script in this case.

6. Deployment Checklist

1. **Rename the DLL:** e.g., `AwesomeSynth_x64.dll`.
2. **Initialize:** Run the DLL in a DAW; this will create a subfolder named `AwesomeSynth/` and an autogenerated `settings.ini`.
3. **Configure Identity:** The auto-generated `settings.ini` is your primary identity template—just customize the `BRANDING` line.
4. **Assets:** Place your `.txt` script and any samples/images inside the `AwesomeSynth/` folder.
5. **FFMPEG Cache:** The engine uses `Documents/Crescendo/` as a base path. For final distribution, collect converted files from the cache and include them in your local resource folder.

7. Professional Error Handling

In the event of a script error, the engine replaces generic technical logs with a branded support block:

```
*****
** FILE "MySynth.txt" - 3 ERRORS DETECTED.
** PLEASE CONTACT support@yourbrand.com FOR ASSISTANCE.
*** (Right-click to copy this log for sending to the developer. CONTROL + C does
not work.) ***
*****
```

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

CURVE defines a series of coordinate points (x, y). The engine then connects these points using **linear interpolation**, allowing you to retrieve any value along the path during runtime.

1. Technical Syntax

`CURVE <index>, <x1>, <y1>, <x2>, <y2>, ... <xN>, <yN>`

- **<index>:** The slot number where the table is stored.
- **<x1>, <y1> ... <xN>, <yN>:** The coordinate pairs defining the shape.
 - **Strict Monotony:** Values for `x_i` must be **strictly increasing** (`x_1 < x_2 < x_3...`).
 - **Minimum Points:** You must define at least two points (`N >= 2`).
 - **Parameter Count:** The total number of parameters must be odd (`1 index + 2N` coordinates).

2. Operational Logic

- **Section Constraint:** Must be placed in the **COMMON** section.

- **Linear Interpolation:** If you request a value between x_1 and x_2 , the engine calculates the corresponding y value along a straight line connecting those two points.
- **Memory Management:** If a `CURVE` is assigned to an index already in use, the old data is overwritten.
- **Usage:** These slots are accessed at runtime by specific functions:
 - `CURVE(index, input)`: Standard LUT lookup.
 - `MOD2(val, index)`: For specialized modulation mapping.
 - `ENVCURVE`: For non-linear envelope segments.

3. Practical Examples

Example A: Simple Parabolic Arc

This defines a curve where $y = x^2$.

```
// --- COMMON SECTION ---
// Slot 42: x goes from 0 to 3, y follows the square
CURVE 42, 0, 0, 1, 1, 2, 4, 3, 9
```

Example B: Custom Velocity Curve (S-Curve)

Used to make a keyboard feel more "expressive" by compressing low velocities and expanding the mid-range.

```
// --- COMMON SECTION ---
// Slot 10: Mapping MIDI velocity (0-1) to a custom response
CURVE 10, 0, 0, 0.3, 0.1, 0.7, 0.9, 1.0, 1.0
```

Example C: Hard-Clipping / Step Function

You can create sharp transitions by placing x points very close together.

```
// --- COMMON SECTION ---
// Slot 5: Sudden jump from 0 to 1 at the midpoint
CURVE 5, 0, 0, 0.499, 0, 0.5, 1, 1, 1
```

4. Technical Constraints

| Rule | Requirement |
|------|-------------|
|------|-------------|

| | |
|---------------|--|
| X-Axis | Must always move forward (no vertical or overlapping lines). |
|---------------|--|

| | |
|----------------------|---|
| Interpolation | Always linear (for "curved" shapes, use more points). |
|----------------------|---|

| | |
|--------------------|---|
| Slot Access | Global; any layer can read any <code>CURVE</code> slot. |
|--------------------|---|

Developer's Note

While the interpolation is linear, you can emulate high-order curves (like exponential or logarithmic shapes) by simply increasing the density of points. For a smooth "warm" response, 5 to 10 points are usually sufficient for the human ear to perceive a continuous curve.

SILTH <number>

`SILTH` defines the amplitude level below which a layer is considered "silent." When this threshold is met—and specific envelope conditions are satisfied—the voice slot is terminated and returned to the pool.

1. Technical Syntax

`SILTH <number>`

- **<number>**: The absolute amplitude threshold.
 - **Default**: $1e-7$ (approx. -140 dB).
 - **Minimum**: $1e-12$ (approx. -240 dB).
 - **Maximum**: 0.1 (approx. -20 dB).

2. The "Deallocation" Logic

A layer slot is only freed if **all** of the following conditions are met at the end of a sample block:

1. **State Check**: The layer must be in **Release mode** (Note Off received) OR all associated envelopes must be at least in **Sustain mode** (or the layer has no envelopes).
2. **Amplitude Check**: The maximum absolute value of the signal sum ($|Left| + |Right|$) must remain **below** the `SILTH` value for the duration of the *entire* sample block across all outputs.

3. Strategic Considerations

The "Infinite Voice" Risk

If a layer contains an oscillator (like `OSCG`) or a looped `SAMPLE` and has **no envelopes** (or the envelopes are bypassed via `IF...GOTO`), the layer will **never terminate** unless the volume is manually automated to zero and the layer receives a Note Off. The engine will continue to process the signal forever, eventually leading to polyphony exhaustion.

Effects & Ring-out

Because the check happens on the final output of the layer:

- **Internal Delays/Reverbs**: These will keep the layer "alive" until their tails fade below the `SILTH` level.
- **POST Step**: Effects placed in the `POST` step do **not** affect layer deallocation, as they are processed after the layer has already mixed its output.

CPU vs. Precision

- **Higher Threshold (e.g., $1e-4$):** Frees voices faster. This is useful for dense orchestrations where you want to kill quiet reverb tails early to save CPU.
- **Lower Threshold (e.g., $1e-9$):** Ensures that long, delicate fades are never "cut off" prematurely, which is essential for high-fidelity ambient sounds.

4. Practical Examples

Example A: Aggressive Voice Management

For a percussion-heavy patch where you want to ensure voices are recycled quickly.

```
// --- COMMON SECTION ---  
// Set threshold to -80dB (approx 0.0001)  
SILTH 0.0001
```

Example B: High-Fidelity Fades

For a piano or pad where the natural decay of the sample or string resonance is vital.

```
// --- COMMON SECTION ---  
// Set threshold to -180dB for ultra-clean tail preservation  
SILTH 1e-9
```

5. Technical Constraints Table

Value Decibel Equivalent Result

| | | | |
|---------------------------|---------|-----------------|---|
| 0.1 | -20 dB | | Extremely aggressive; cuts off audible sound. |
| $1e-7$ | -140 dB | Default; | safe for almost all applications. |
| $1e-12$ | -240 dB | Floor; | essentially silence in a 24-bit/32-bit environment. |

Developer's Note

If you notice your polyphony count (visible via `HIDEUI`) isn't dropping back to zero after you stop playing, you likely have a "leak." This usually means your signal isn't dropping below the `SILTH` threshold—often caused by a feedback loop or a DC offset in a custom filter. No error is given for values out of range: values less than $1e-12$ (-240 dB) are treated as $1e-12$ and values greater than `.1` are treated as `.1`.

OVNAN <value>

Sets the threshold for the NaN (Not-a-Number) and Overload protection system.

- **Settings:**
 - **Value < 1.0:** Protection is completely **DISABLED**. The engine runs at maximum speed.
 - **Value >= 1.0:** Protection is **ENABLED**. If the signal absolute value exceeds this threshold, or if a mathematical error (NaN) occurs, the buffer is safely handled/muted and all the LAYERS and the POST step are silenced.
- **Max Limit:** Capped at **1e38**.
 - *Note: Using very high values (like the cap) effectively turns this into a "NaN-only" protection, as it's nearly impossible for a standard signal to reach 1e38, though it will still catch +/- Infinity.*
- **Default:** **1e5** (100,000.0).

CHOKE <time>

CHOKE forces a brief, automated release phase for any active voices sharing the **exact same MIDI key** when a new Note On event for that key is received.

The CHOKE instruction provides a specialized mechanism for handling note re-triggers. It prevents the "layering" of the same note on top of itself, which is essential for simulating physical instruments where a new strike naturally dampens the previous vibration of the same string or drum head.

1. Technical Syntax

CHOKE <time>

- **<time>:** The duration of the forced release in seconds.
 - **Default:** 0 (Choking is disabled; notes will layer indefinitely).
 - **Range:** Typically very small values (e.g., 0.001 to 0.1) are used for natural dampening.

2. Operational Logic

- **Section Constraint:** Must be placed in the **COMMON** section.
- **Trigger Mechanism:** When a key is pressed (e.g., C3), the engine checks if a voice is already playing C3. If it finds one, it immediately puts that older voice into a release phase of <time> seconds.
- **Dampening Simulation:**
 - **Acoustic Piano:** On a real piano, the hammer strikes the string while the damper is raised. If the string was already vibrating, the new strike briefly interferes with the old vibration.
 - **Drum Kits:** Prevents "machine-gun" build-up of cymbals or snares where each hit adds perfectly to the previous one, creating unrealistic volume peaks.

3. Strategic Use Cases

Example A: Piano String Behavior

To simulate the physical dampening of a piano string being struck again.

```
// --- COMMON SECTION ---
// Force a 20ms fade-out of the old note when re-triggered
CHOKE 0.02
```

Example B: Percussion Realism

Prevents unrealistic phase accumulation on high-hats or rides.

```
// --- COMMON SECTION ---
// Very fast choke to clean up the previous hit
CHOKE 0.005
```

4. Interaction with Envelopes

The `CHOKE` time acts as a temporary override.

- If your `ENVELOPE` has a long release time (e.g., 2.0s), the `CHOKE` value (e.g., 0.01s) will force the voice to end much faster *only* when the same key is hit.
- If the note is released normally (without a re-trigger), the standard `ENVELOPE` release time is used.

5. Developer's Summary

| Setting | Behavior | Effect |
|---------|----------|--------|
|---------|----------|--------|

| | | |
|--------------------|--------------|---|
| 0 (Default) | Polylayering | Notes of the same key stack on top of each other. |
|--------------------|--------------|---|

| | | |
|---------------|------------------|---|
| > 0 | Mono-Key Choking | Previous instance of the same key is faded out. |
|---------------|------------------|---|

Developer's Note

Be careful with `CHOKE` values that are too high. If the choke time is longer than the time between fast repetitions, you can still end up with overlapping voices. Conversely, a `CHOKE` time of 0.001 (1ms) might cause a subtle "click" if the fade is too abrupt. 0.01 to 0.05 is generally the "sweet spot" for natural instrument behavior.

PROGRAMNAME <Program number>;
<Bank number>; "<Program name>"

The `PROGRAMNAME` instruction allows you to provide descriptive labels for MIDI Programs. These names are displayed in the Crescendo interface (depending on your `HIDEUI` settings) and are communicated to the Host DAW, making it much easier for the user to navigate large sound sets or multi-timbral templates.

This instruction acts as a metadata tag for specific MIDI Program Change messages. It maps a human-readable string to a specific Bank and Program combination.

1. Technical Syntax

```
PROGRAMNAME <Program_number>, <Bank_number>, "<Program_name>"
```

- **<Program_number>**: The MIDI Program ID (0–127).
- **<Bank_number>**: The MIDI Bank ID (0–16383).
- **"<Program_name>"**: The text label for the program.
 - **Length Limit**: Maximum **19 bytes**. Exceeding this will trigger an error.
- **Storage Limit**: The engine supports up to **32,768** individual entries.

2. Operational Logic

- **Section Constraint**: Must be placed in the **COMMON** section.
- **UI Integration**:
 - If **HIDEUI** is set to **5** or **6**, the program name will appear in the top bar.
 - If **HIDEUI** is set to **10**, a full text box displays the program names for all 16 MIDI channels simultaneously.
- **DAW Visibility**: These names are typically passed to the VST host, allowing users to select patches by name from within the DAW's inspector or track headers.

3. Practical Examples

Example A: Standard GM-style Mapping

Defining names for different instruments within the first bank.

```
// --- COMMON SECTION ---
PROGRAMNAME 0, 0, "Grand Piano"
PROGRAMNAME 1, 0, "Bright Piano"
PROGRAMNAME 19, 0, "Church Organ"
```

Example B: High-Bank Multi-Timbral Setup

Defining a specific patch in a high-numbered bank.

```
// --- COMMON SECTION ---
PROGRAMNAME 5, 1024, "Cinematic Pad"
```

4. Developer's Summary Table

| Feature | Constraint |
|--------------------------|-------------|
| Max String Length | 19 Bytes |
| Max Total Entries | 32,768 |
| Bank Range | 0 to 16,383 |
| Program Range | 0 to 127 |

5. Developer's Note

While you can define thousands of names, remember the **19-byte limit**. This includes spaces. If you need to abbreviate, use standard musical shorthand (e.g., "Leg." for Legato, "Sust." for Sustain) to ensure the labels remain clear and fit within the interface's allocated text fields.

6. Integration: PROGRAMNAME & VSTVAR

When a `VSTVAR` is mapped to monitor a MIDI Program Change or a CC that controls patch selection, Crescendo automatically replaces the numerical value with the corresponding string defined in your `PROGRAMNAME` list. See `MIDICC` or `LINKCC` for details on linking.

a. Display Logic

- **The "Unnamed" Fallback:** If a user selects a Program/Bank combination for which no `PROGRAMNAME` has been defined, the UI will display: `((Unnamed_Program))`.
- **DAW Communication:** This name-swapping isn't just visual within the Crescendo GUI; the engine sends these strings to the **Host DAW**. When the user opens the VST parameters in their DAW's automation lane or generic editor, they will see the patch names instead of raw numbers.

b. Practical Implementation

To make this work effectively, you must have your `PROGRAMNAME` entries defined in the `COMMON` section, and your `VSTVAR` should be scaled to the appropriate 0–127 range.

```
// --- COMMON SECTION ---

// 1. Define the names
PROGRAMNAME 0, 0, "Deep Sine Bass"
PROGRAMNAME 1, 0, "Moog Lead"
PROGRAMNAME 2, 0, "Analog String"

// 2. Link the VSTVAR 0 with the Program MIDI CC of channel 0
// This knob will now show the names above instead of 0, 1, 2...
MIDICC 134, 0, 0, 0 // See MIDICC help for details
```

c. Strategic Advantages

- **User Clarity:** The user doesn't need to remember that "Program 12" is the "Flute"; the knob label simply says "Flute."
- **Searchability:** Many DAWs allow you to search for parameters or programs by name.
- **Consistency:** Using `HIDEUI 5` or `HIDEUI 10` alongside this `VSTVAR` mapping creates a cohesive interface where the "active" program is always visible in multiple places.

Developer's Note

Always ensure you define a name for **Program 0, Bank 0**. Since this is the default state upon loading the plugin, having it labeled "Default Init" or your primary sound is much better than the user seeing `((Unnamed_Program))` the moment they open your instrument.

MIDI Instructions, stage I:

Input/Output

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. See the SMOOTH instructions for details.
3. Filtering the data with MCCCTL or MIDICH.
4. Outputting an expression with MIDIOUT.
5. **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.
 - **LINKCC Instruction:** This instruction establishes a connection between two MIDI CCs.
 - **TEMPERAMENTCC Instruction:** This instruction links the Temperament drop-down list to a specific MIDI CC of a specific channel or a VST VAR, allowing dynamic changes to the tuning system using a MIDI controller or the DAW automation. This opens up creative possibilities for real-time shifts in temperament during a performance.

MIDICH <min>;<max>

The MIDICH instruction acts as the primary "gatekeeper" for MIDI data entering the engine. It defines which MIDI channels (0-15) the instrument is listening to. Any data outside this range is ignored by the global system, saving processing power and preventing unwanted cross-talk in multi-timbral setups.

This instruction defines the global listening window. It is the first layer of defense in MIDI Stage I.

1. Technical Syntax

MIDICH <min>, <max>

- **<min>:** The lowest channel number to monitor (0–15).
- **<max>:** The highest channel number to monitor (0–15).
- **Default:** 0, 15 (All 16 channels are active).

2. Operational Logic

- **Section Constraint:** Must be placed in the **COMMON** section.
- **Data Discarding:** MIDI messages (CC, Program Change, etc.) coming from channels outside this range are immediately discarded.
- **Internal State:** For disabled channels, the MIDI CC slots remain at their default value of 0.
- **MIDI Output:** When Crescendo acts as a MIDI processor, excluded channels are not transmitted to the `MIDIOUT`.
- **Note Handling:** While `MIDICH` filters global data, you use the `ONCHANNEL` trigger (which we will see in the execution sections) to perform specific filtering for **Note On** events.

3. Practical Examples

Example A: Dedicated Single-Channel Synth

If your instrument is designed to respond only to MIDI Channel 1 (Channel 0 in code).

```
// --- COMMON SECTION ---  
// Discard all MIDI data except for Channel 1  
MIDICH 0, 0
```

Example B: High-Channel Multi-Timbral Mode

If you are building a specialized sound module that only responds to the "upper half" of MIDI channels (9-16).

```
// --- COMMON SECTION ---  
// Monitor only channels 8 through 15  
MIDICH 8, 15
```

4. Developer's Summary Table

| Feature | Behavior |
|---------|----------|
|---------|----------|

| | |
|----------------------|--|
| Filtered Data | CC, Program Change, Pitch Bend, Poly Pressure. |
|----------------------|--|

| | |
|------------------------|------------------------------------|
| Default Setting | 0 to 15 (Omni-channel monitoring). |
|------------------------|------------------------------------|

| | |
|----------------------|---|
| Memory Impact | Disabled channels do not update their internal CC tables. |
|----------------------|---|

| | |
|--------------------|---|
| Note Filter | Use <code>ONCHANNEL</code> for more granular Note On/Off control. |
|--------------------|---|

Developer's Note

Setting a narrow `MIDICH` range is highly recommended for complex VST templates. If your instrument only needs one channel, limiting it to 0, 0 ensures that MIDI CC automation intended for other plugins in your DAW doesn't accidentally fall through the MIDI output.

MIDISWAP <ch1>;<ch2>

This instruction intercepts incoming MIDI data and swaps the channel assignments. It is particularly useful when using MIDI controllers with fixed output channels or when managing multi-track recording in the internal Sequencer.

1. Technical Syntax

`MIDISWAP <ch1>, <ch2>`

- **<ch1> / <ch2>**: The two channels to be swapped (0–15).
- **Dynamic Mapping**: You can use the `$` prefix followed by a MIDI CC address (e.g., `$602`) to make the swap dynamic. In this case, the channel number is determined by the current value of that MIDI CC/VST Variable.
- **Default**: No channels are swapped.

2. Operational Logic

- **Section Constraint**: Must be placed in the **COMMON** section.
- **Timing**: This happens at the very beginning of the MIDI processing chain.
- **Channel Agnostic VST Vars**: Because VST Variables are often channel-agnostic, using the dynamic `$` syntax allows the user to change the routing live from the GUI.
- **The Sequencer Workflow**: This is often used to record multiple tracks onto the internal "Tape" (Sequencer). A user with a keyboard fixed to Channel 0 can record onto Track 0, then use `MIDISWAP` to redirect their Channel 0 input to Channel 1 to record the next track without changing their hardware settings.

3. Practical Example: Dynamic Routing GUI

In this example, we create two knobs to define which incoming channel gets sent to which destination channel.

```
// Setup the interface
INTERFACE 1000, 200
VSTVARS 4
VSTVAR 0,.25,"VOLUME","",0,1,1
VSTVAR 1, 2,"x","",0,2,2 // Sequencer Knob. See Sequencer section
VSTVAR 2,0,"SRC CH","",0,15,2
VSTVAR 3,0,"DST CH","",0,15,2
TOOLTIP "Main Volume", 0
TOOLTIP "Sequencer Knob ( 音序器旋钮 )", 1
TOOLTIP "MIDISWAP source channel:\nThis channel will be swapped with the
destination channel.\nUse those knobs to record the wanted channel in the
sequencer, in case your keyboard do not allow channel choosing.\nSet this
channel number to the channel number configured in your keyboard.\n\nWait the 5
second countdown.\n\nNOTE: change only before start merging!", 2
```

```

TOOLTIP "MIDISWAP destination channel:\nThis channel will be swapped with the
source channel.\nUse those knobs to record the wanted channel in the sequencer,
in case your keyboard do not allow channel choosing.\nSet this channel number to
the one you want to record in the TAPE.\n\nWait the 5 second countdown.\n\nNOTE:
change only before start merging!", 3
// Swap the MIDI channel given by VST VAR 2 by that given by VST VAR 3
MIDISWAP $602,$603
// UI customization
UIMOD 0,30,0,2,5
UIMOD 400,50,90,70,0,3,20,30,130,30,255,255,255,190,90,190
UIMOD 401,200,90,70,3.5,0,0,130,30,30,150,150,0,0,80,0
UIMOD 402,350,110,30,0,0,50,-1,-1,-1,255,0,0
UIMOD 403,490,110,30,0,2,1
UIMOD 404,630,110,30,0,1,0
UIMOD 405,760,110,30,0,1,0
UIMOD 406,860,110,30,0,1,0
UIMOD 709,0,136,255
UIMOD 712,0,136,255
UIMOD 711,0,0,170
UIMOD 2,360,20,48,900,255,255,255
HIDEUI 4
// GAIN modulated by VST VAR 0 (Volume) and with a fixed envelope
GAINENV &600,0,.1,0,0,1,.1
// Setup of the sequencer with a countdown
// Start merge mode e.g. with channel 0 going in channel 0 to record the first
track (if your keyboard is configured to output on channel 0)
// Then rewind the tape, set rerouting of channel 0 to channel 1 and record the
track for the channel 1, and so on. You can record up to 16 tracks.
SEQUENCERI "ACTIVATE 601, 10000",0,15
SEQUENCERI "countdown 5"
// Simple sinusoidal oscillator with the simple envelope created above
// In true instruments you will put one instrument per channel, maybe swappable
with program and bank.
// E.g. you can copy this header, adapting the VST numbers, to a file produced
by the Soundfont import, after importing e.g. a GM SoundFont. With the _16 file
you can have 16 independent instruments.
LAYER
OUT=OSCG("sg")
DEBUG 0,250,50

```

4. Developer's Summary Table

| Feature | Detail |
|------------------------|---|
| Input Range | 0 to 15 (Standard MIDI 1–16). |
| Dynamic Support | Yes, via \$ <MCC> syntax. |
| Primary Use | Hardware compatibility and multi-track recording. |
| Placement | COMMON section only. |

Developer's Note

When using dynamic swaps with knobs (like §602), remember that the engine interprets the *value* of the knob as the channel number. Ensure your `VSTVAR` range is strictly 0 to 15 and uses a step of 1 to avoid undefined behavior. Changing these values during a live performance is possible, but it is best done while the "Tape" is stopped to avoid orphaned "Note Off" messages.

MCCCTL <0,1,2,3>, <mccmin>, <mccmax>, <chmin>, <chmax>

This instruction determines whether specific MIDI indices on specific channels are processed by the engine (Input) or transmitted out to the DAW/other plugins (Output).

The function is additive in the sense that consecutive instructions modify the current map and accumulates: e.g. if you first enable a range and then enable another range, as a result both ranges will be enabled.

While `MIDICH` acts as a broad "front gate" for the engine, **MCCCTL** (MIDI CC Control) provides the high-resolution "switchboard" for MIDI Stage I. It allows you to precisely enable or disable Input and Output for specific types of data, ranging from standard knobs to Note ON/OFF events.

1. Technical Syntax

`MCCCTL <Mode>, <mccmin>, <mccmax>, <chmin>, <chmax>`

- **<Mode>**: The I/O state (0, 1, 2, or 3):
 - **0**: Disable both Input and Output.
 - **1**: **Enable Input** / Disable Output.
 - **2**: Disable Input / **Enable Output**.
 - **3**: **Enable both** Input and Output.
- **<mccmin>, <mccmax>**: The range of MIDI Control addresses (see the Address Map below).
- **<chmin>, <chmax>**: The range of MIDI channels (0–15) to apply this filter to.

2. The MIDI Address Map

Crescendo uses a unified addressing system to treat almost everything as a "MIDI CC" for filtering purposes:

Address Range Data Type

| | |
|------------------|---|
| 0 – 127 | Standard MIDI CCs (Mod Wheel, Volume, etc.) |
| 128 – 327 | Extended MIDI Messages (Aftertouch, Pitch Bend, etc.) |

Address Range Data Type

600 – 727 **VST Variables** (Knobs 0–127)

1000 – 1007 Keyswitches

1100 – 1227 **Note ON** events

1300 – 1427 **Note OFF** events

Strictly Reserved (Always Rejected): Indices below 0, 128-132, 136-143, 145-154, 156-159, 328-599, 728-999, 1228-1299, and above 1428 are ignored by the filter.

3. Default Behaviors

By default, Crescendo is configured to be "plug-and-play" for standard music making:

- **Enabled (Mode 3):** All valid MIDI CCs below 327 (excluding Temperament 144) and Note ON/OFF (1100-1427).
- **Disabled (Mode 0):** VST Variables (600+) and Temperament (144). These are considered "local" to the plugin—they respond to GUI mouse movements or DAW automation, but won't listen to raw MIDI input unless you explicitly enable them via `MCCCTL`.

4. Practical Use Cases

A. Transmitting Knob Movements to the DAW

If you want the movements of your custom VST knob #0 (Address 600) to be sent out as MIDI data to the DAW:

```
// Enable Output (Mode 2) for VST Var #0 (600) on Channel 0
MCCCTL 2, 600, 600, 0, 0
```

B. Protecting a Channel from Performance Data

In a multi-timbral setup, you might want Channel 5 to be "silent" to notes but still receive global CC data:

```
// Disable Input/Output for Note ON (1100) and Note OFF (1300) on Ch 5
MCCCTL 0, 1100, 1427, 5, 5
```

C. Bandwidth Optimization

If your hardware sends constant Polyphonic Aftertouch that you aren't using, you can kill it to save CPU:

```
// Disable Input/Output for Aftertouch range (200-327) on all channels
MCCCTL 0, 200, 327, 0, 15
```

D. Creating a "Notes Only" Filter

To create a setup where the plugin receives musical notes but ignores all standard controller data (CC 0–127), you can disable the input for the controller range.

```
// Disable Input and Output for standard CCs 0-127 on all channels
MCCCTL 0, 0, 127, 0, 15
```

E. Disabling Channel Mode Messages

To prevent the plugin from responding to or passing through MIDI channel mode messages (such as "All Notes Off" or "Reset All Controllers"), you can disable their corresponding indices.

```
// Disable channel mode messages (typically index 120-127)
MCCCTL 0, 120, 127, 0, 15
```

Developer's Note

MCCCTL is your primary tool for "MIDI Sanitization." If your plugin is behaving strangely—for instance, if moving a knob on one instance of Crescendo is unexpectedly moving a knob on another instance—it is because the MIDI Output for VST Vars is enabled. Use MCCCTL to keep your MIDI stream clean and relevant.

POSTCH <channel>

The `POSTCH` instruction defines the "home" MIDI channel that the **POST** step will look at by default when querying MIDI Control Change (MCC) values.

1. Technical Syntax

`POSTCH <channel>`

- **<channel>**: The MIDI channel number (0–15).
- **Default**: 0 (MIDI Channel 1).

2. Operational Logic

- **Section Constraint**: Must be placed in the **COMMON** section.
- **Context**: The **POST** step is a global process (non-polyphonic). Because MIDI data is channel-specific, the engine needs to know which channel's data "stream" to use for global effects like a master volume or a shared filter cutoff.
- **Usage with MCC Functions**: When you use the following functions in your `POST` code, they will automatically pull data from the channel defined by `POSTCH`:
 - `MCCnnnn`: Direct access to a specific CC.
 - `MCC`: Standard CC access.
 - `MCC2`: Specialized CC access.

- **Overriding the Default:** If you need to access a different channel for a specific calculation within the same `POST` step, you must use the `MCC3` function, which allows you to explicitly define a different channel.

3. Practical Example: Master FX Control

Imagine you are building a multi-timbral instrument, but you want your Master Reverb (defined in the `POST` step) to always be controlled by the Mod Wheel (CC 1) of **MIDI Channel 16** (Channel 15 in code).

```
// --- COMMON SECTION ---
// Set the default channel for the POST step to 15
POSTCH 15

// ... later in the POST section ...
// This will automatically look at CC 1 on Channel 15
POST
    MOD = MCC(1)
    // Apply MOD to a global reverb effect...
```

4. Developer's Summary Table

Component Setting

| | |
|-----------------|--|
| Scope | Global (Affects only the <code>POST</code> step logic). |
| Range | 0 to 15. |
| Override | Use <code>MCC3(cc, channel)</code> to bypass this setting. |

Developer's Note

Think of `POSTCH` as your "Global Remote Control" setting. While each layer (voice) is naturally aware of its own triggering channel, the `POST` step is "blind" to individual notes. Setting `POSTCH` ensures your master effects respond to the specific channel you've designated for performance controls.

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

The `MIDICC` instruction is the fundamental bridge between MIDI data and the Crescendo engine's parameters. It goes beyond simple routing by allowing you to initialize starting values, link MIDI to GUI knobs, and leverage the complex scaling logic (like logarithmic curves) defined in your `VSTVAR` instructions.

This instruction defines the initial state of a MIDI controller and, optionally, "hard-wires" it to a VST Variable.

1. Technical Syntax

MIDICC <number>, <initial_value>, [<vstvar>], [<channel>]

- **<number>**: The MIDI address (0–199).
 - **0–127**: Standard MIDI CCs (Mod Wheel, Volume, etc.).
 - **133, 134, 135, 155**: Extended CCs for Aftertouch, Program, Pitch Bend, and Bank.
 - **160–199**: Developer "Wildcards." These 40 slots (per channel) can store constants or custom float values.
 - Other MIDI CC numbers cannot be initialized or linked, in particular the temperament has its own instruction.
- **<initial_value>**: The value the CC takes upon instrument load.
 - 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>**: The ID of the VSTVAR (0–127) to link.
 - Use **-1** to leave the CC unlinked.
 - To avoid confusion or cut and paste errors, also the 600-727 range is recognized.
- **<channel>**: The MIDI channel (0–15). Default is **0**.

2. The Mapping Logic

When a MIDI CC is linked to a VST Variable:

- **Value Mapping**: MIDI 0 is mapped to the `VST_min`; MIDI 127 is mapped to the `VST_max`.
- **Scale Preservation**: If the VSTVAR is defined as logarithmic, the linear MIDI input (0–127) will control the parameter logarithmically. This is a powerful way to handle frequency or decibel controls without complex math in the `LAYER` code.
- **Hidden Variables**: You can link a CC to a high-index VSTVAR (like #127) that isn't displayed on the GUI to perform background math transformations on incoming MIDI data.

3. Critical Rules of Engagement

| Rule | Requirement |
|----------------------------|--|
| Order of Operations | MIDICC must appear after the VSTVAR it references. If not, the VSTVAR initialization will overwrite the MIDICC value. |
| Unlinking | This instruction effectively "breaks" any previous links for that CC or VST variable, including temperaments. |
| Local | Every MIDICC update issues a <code>[LOCAL]</code> message, ensuring the local MIDI CC |

| Rule | Requirement |
|------------------------|--|
| Messaging | storage is synchronized. |
| DAW Interaction | Changes to a linked CC notify the Host DAW as if a knob was moved by a mouse, which may pause DAW automation to prevent conflicts. |

4. Practical Examples

Example A: Initializing Hardware Pedals

Ensuring Sustain (CC 64) and Sostenuto (CC 66) are linked to GUI switches.

```
// Define VST Knobs for GUI display
VSTVAR 0, 0, "HOLD", "", 0, 1, 4, "OFF", "ON"
VSTVAR 1, 0, "SOSTENUTO", "", 0, 1, 4, "OFF", "ON"

// Link MIDI CCs to the knobs defined above
MIDICC 64, 0, 0 // CC 64 (Sustain), Initial 0, link to VST Var #0
MIDICC 66, 0, 1 // CC 66 (Sostenuto), Initial 0, link to VST Var #1
```

In this setup, a value of 0 on the MIDI CC maps to the VST minimum (OFF), and 127 maps to the VST maximum (ON).

Example B: The "Wildcard" Constant

Using the 160+ range to store a custom global value (like a fixed calibration offset) that doesn't correspond to a standard MIDI message.

```
// --- COMMON SECTION ---
// Store a custom float (440.0) in slot 160 on Channel 0
MIDICC 160, 440.0, -1, 0
```

Example C: Initializing Extended Parameters (Bank and Aftertouch)

You can use MIDICC to ensure that polyphonic aftertouch or bank selections start at a specific state rather than defaulting to zero.

```
// Set initial Bank selection to 100 on Channel 5
MIDICC 155, 100, -1, 5

// Set initial Channel Aftertouch to 64 on Channel 0
MIDICC 133, 64
```

Example D: Logarithmic Transformation via Hidden Variables

A powerful sound-design application of MIDICC is to change the scaling of a linear MIDI controller. By linking a standard linear MIDI CC to a "hidden" VST variable (one not exposed to the GUI), you can exploit the VST variable's scaling logic.

```
// Define a hidden VST Var #127 with logarithmic scaling
// VSTVARs is not required to be set high enough to include this index
VSTVAR 127, 440, "Filter Log", "Hz", 20, 20000, 0

// Link MIDI CC 10 (Pan) to the logarithmic VST Var #127
MIDICC 10, 0, 127
```

The linear MIDI CC values (0–127) will now drive the internal frequency parameter logarithmically.

Developer's Note

The ability to use indices 160–199 as "floating point constants" is a secret weapon for advanced scripters. Since these slots are channel-aware (40 slots x 16 channels), you can use them to store per-channel calibration data or state variables that persist even if they aren't tied to a physical knob.

LINKCC <mcc1>;<ch1>;[<mcc2>;[<ch2>]]

This instruction establishes a persistent connection between a source MIDI CC and a destination MIDI CC or VST variable.

The `LINKCC` instruction is the sophisticated "patch cable" of the Crescendo engine. While `MIDICC` is great for simple one-to-one mapping, `LINKCC` allows you to create complex internal routings between different MIDI controllers, channels, and VST variables, handling the scaling math automatically based on which "Set" the controllers belong to.

1. Technical Syntax

```
LINKCC <mcc1>, <ch1>, [<mcc2>], [<ch2>]
```

- **<mcc1>, <ch1>**: The source MIDI CC and its channel (0–15).
- **<mcc2>, <ch2>**: The destination MIDI CC and its channel (0–15).
 - **Default <ch2>**: 0 (if omitted).
 - **Breaking a Link**: If <mcc2> is omitted or set to -1, any existing link for <mcc1> is broken.
- **Address Range**: Valid indices include 0–127, 133, 134, 135, 155, 160–327, 600–727 (VST VARs), and 1000–1007 (Keyswitches).
 - *Note: If an index is greater than 327, the channel is forced to 0.*

2. The Set System & Mapping Logic

Crescendo divides the MIDI address space into two sets to determine how values are translated during a link:

Set Members

Set 1 0–327 (excluding 135), and 1000–1007.

Set Members

Set 2 135 and VST VARs (600–727).

How Values Flow:

- **Intra-Set (1 to 1 or 2 to 2):** Direct 1:1 mapping. No transformation is performed other than basic rounding or clipping.
- **Inter-Set (1 to 2 or 2 to 1):** Scaled mapping. **0** maps to the **Minimum**; **127** maps to the **Maximum**.
 - *Example:* Linking CC 11 (Expression) to a VST Filter knob honors the knob's scaling (e.g., logarithmic).
- **Direct Exceptions:** CCs 0, 32, 134, 155, and the 160–199 range always use **direct numerical mapping** when linked to a VST Variable, bypassing the 0–127 scaling.

3. Operational Rules

- **Section Constraint:** COMMON section only.
- **The "Clean Slate" Rule:** Linking two objects immediately deletes any previous links they had, including links to temperaments.
- **The 144 Restriction:** You **cannot** use LINKCC with index 144. This will trigger an error; you must use TEMPERAMENTCC instead.
- **Initialization:** When the link is established, the current value of <mcc1> is immediately copied to <mcc2>.
- **Feedback Loops:** If <mcc1> == <mcc2> and <ch1> == <ch2>, the link is simply broken.

4. Practical Examples

Example A: Hardware Compatibility (Direct Set 1 Link)

Redirecting a Mod Wheel (CC 1) to act as a Breath Controller (CC 2).

```
// --- COMMON SECTION ---  
// Link CC 1 (Ch 0) to CC 2 (Ch 0)  
LINKCC 1, 0, 2, 0
```

Example B: Scaling a GUI Knob (Set 1 to Set 2)

Driving a VST Variable (Index 600) with the Expression Pedal (CC 11).

```
// --- COMMON SECTION ---  
// If VSTVAR 0 is logarithmic, CC 11 will now drive it logarithmically  
LINKCC 11, 0, 600, 0
```

The engine honors the scale of the VST Variable. If the knob is set to a logarithmic frequency scale (e.g., 20Hz to 20,000Hz), the linear MIDI CC values will drive that frequency logarithmically.

Example C: Breaking a Link

Silencing a previous routing for the Sustain pedal.

```
// --- COMMON SECTION ---  
// Break link for CC 64 on Channel 0  
LINKCC 64, 0, -1
```

Example D: Internal Controller Redirect (Set 1 to Set 1)

If you have a hardware controller that fixedly outputs on CC 1 (Modulation) but your instrument is designed to respond to CC 4 (Foot Controller), you can link them internally.

```
// Link Modulation (CC 1) on Channel 0 to Foot Controller (CC 4) on Channel 0  
LINKCC 1, 0, 4, 0
```

Any movement on the modulation wheel will now update the foot controller's internal storage.

Example E: Using User-Defined Constants (Direct Mapping)

The user-defined CC range (160–199) allows for storing 32-bit floating-point constants that can be mapped to VST knobs without 0–127 rescaling.

```
// Link User-Defined CC 160 on Channel 5 to VST Variable #5 (Index 605)  
LINKCC 160, 5, 605, 0
```

Developer's Note

LINKCC is particularly powerful because it issues a [LOCAL] MIDI message upon any update. This ensures that the entire engine stays perfectly in sync. If you link a MIDI CC to a VST VAR, the plugin will even notify the Host DAW to "pause" automation to prevent the physical controller and the DAW's automation from fighting over the same knob.

MIDIOUT <mcc>,<ch>,<T>,<expr>

The MIDIOUT instruction is the "transmitter" of the Crescendo engine. While previous instructions focused on how the engine *receives* and *routes* MIDI, MIDIOUT allows you to generate MIDI data based on internal calculations, GUI states, or layer logic and send it back out to the DAW or external hardware.

This instruction samples an expression and outputs the result as a MIDI message. It is the primary tool for MIDI feedback, controller translation, and high-precision automation.

1. Technical Syntax

```
MIDIOUT <mcc>, <ch>, <T>, <expr>
```

- **<mcc>**: The target MIDI address (0–127, 133–144, 160–327, 600–727, 1000–1007).
- **<ch>**: The MIDI channel (0–15). Use **-1** for the default channel.
- **<T>**: The sampling period in **BARs**.
 - The expression is evaluated at the audio rate, but only "sent" every <T> bars.

- A message is only transmitted if the value has changed since the last sampled value (including clipping and rounding).
- **<expr>**: The mathematical or variable-based expression to be sampled.

2. Operational Logic & Placement

The behavior of `MIDIOUT` changes significantly depending on where it is placed in your script:

Placement Behavior

COMMON Inherited by both the `POST` section and every `LAYER`. Forces an empty `POST` section if none exists.

POST **Recommended.** Runs continuously as a global process. Ideal for general MIDI translation or GUI-to-MIDI feedback.

LAYER Runs only while the specific note/layer is active. Useful for note-specific data (e.g., sending a CC based on a note's internal envelope).

Note on Performance: Standard MIDI CCs (0–127) are automatically rounded and clipped. For custom ranges (160–199), the engine preserves floating-point precision.

3. Strategic Use: The "Global Data" Bridge

A key limitation of variables in Crescendo is that they are **local**. An assignment like `FOO = 10` in Layer A cannot be read by Layer B.

To pass data between layers or between a layer and the `POST` step, you must use "Global Hooks":

1. **VST Variables:** Use hidden VST VARs (e.g., #127) to store shared states.
2. **Unused MIDI CCs:** Use the 160–199 range as a shared data pool.

Workflow:

1. **Assign:** `VAR127 = <calculation>` (Updates local storage).
2. **Output:** `MIDIOUT 600, 0, 0.01, VAR127` (Updates global state and MIDI output).

4. Practical Examples

Example A: High-Resolution 14-bit Volume

Splitting a high-res VST knob (#0) into standard MSB/LSB MIDI messages.

```
POST
// Send MSB to CC 7 and LSB to CC 39 every 0.01 bars
```

```
MIDIOUT 7, 0, 0.01, HIBYTE(VAR0)
MIDIOUT 39, 0, 0.01, LOWBYTE(VAR0)
```

Example B: Dynamic Gain Monitoring

Sending the current volume of a processed signal to an external meter.

```
// Calculate gain once
NEWGAIN = GAIN * SMOOTH(IN, 0.1)

// Output to CC 7 (scaled for MIDI 0-127)
MIDIOUT 7, -1, 0.05, NEWGAIN * 127

// Apply to audio
OUT = NEWGAIN * OSCG("sg")
```

Developer's Note

Be mindful of the period `<T>`. While you might be tempted to set it to a very low value for "perfect" resolution, MIDI is a serial protocol with limited bandwidth. Setting a value like `0.0001` could flood your DAW or external hardware with more messages than it can process, leading to lag or crashes. For most sliders and knobs, `0.01` to `0.05` is the "Goldilocks" zone.

RPNLINK <mode>, <number>, <channel>, <mcc>

This instruction creates a unidirectional link from a 14-bit RPN/NRPN message to an internal MIDI CC or VST Variable.

1. Technical Syntax

```
RPNLINK <mode>, <number>, <channel>, <mcc>
```

- **<mode>**: Defines the message type (RPN vs NRPN) and how the data is processed (MSB, LSB, or Scaled).
- **<number>**: The RPN/NRPN parameter number.
- **<channel>**: The MIDI channel (0–15).
- **<mcc>**: The target destination (Standard CC, VST VAR, or Extended CC).

The MIDI CC linkable are:

- 133, 134, 135, 155: Aftertouch, Program, Pitch bend and Bank number. The same channel of the RPN/NRPN message is linked.
- 144: Temperament. This MIDI CC does not have a channel.
- 160-199: Unassigned numbers can be used to store constants. The same channel of the RPN/NRPN message is linked. For future compatibility, you can assume that MIDI CCs up to 159 are assigned. 160-199 are left to the user.
- Other MIDI CC numbers cannot be linked.
- 600-727 are the VST Variables. These MIDI CC do not have a channel.
- 1000-1007 are the Keyswitches. These MIDI CC do not have a channel.

Use other MIDI CC numbers to unlink a MIDI CC from the RPN/NRPN message.

2. Mode Reference Table

| Mode Type | Data Handling | Best For... |
|-----------|---------------|-------------|
|-----------|---------------|-------------|

| | | |
|-------|------------------------------------|--------------------------------------|
| 0 / 5 | RPN / NRPN MSB Only (7-bit) | Standard switches or coarse control. |
|-------|------------------------------------|--------------------------------------|

| | | |
|-------|------------------------------------|----------------------|
| 1 / 6 | RPN / NRPN LSB Only (7-bit) | Fine-tuning offsets. |
|-------|------------------------------------|----------------------|

| | | |
|-------|-------------------------------------|---|
| 2 / 7 | RPN / NRPN Full 14-bit (Raw) | Internal math requiring raw values up to 16383. |
|-------|-------------------------------------|---|

| | | |
|-------|---------------------------------|---|
| 3 / 8 | RPN / NRPN Scaled [0, 1] | Continuous parameters (Filter, Volume). |
|-------|---------------------------------|---|

| | | |
|-------|----------------------------------|---------------------------------------|
| 4 / 9 | RPN / NRPN Scaled [-1, 1] | Bipolar parameters (Pan, Pitch Bend). |
|-------|----------------------------------|---------------------------------------|

3. Strategic Destination Mapping

Where you point the RPN data matters. Crescendo treats these targets with specific logic:

- **VST Variables (600–727):** Use the scaled modes (3, 4, 8, or 9). This ensures the 14-bit data maps smoothly across the knob's range, honoring any logarithmic or linear scaling you've defined.
- **Temperament (144):** Linking an RPN to the temperament index allows for high-precision, real-time tuning switches.
- **Extended CCs (160–199):** Excellent for storing high-resolution constants or user-defined states that don't need a visible GUI knob.

4. Operational Logic: The "Remote" Rule

- **Section Constraint:** Must be in the **COMMON** section.
- **Unidirectional:** Data flows from the external RPN/NRPN into the engine. Moving a linked VST knob on the GUI will **not** send an RPN message back out.
- **Output Silence:** RPN/NRPN messages are consumed by the engine and are **never** echoed to the MIDI Output. This keeps your MIDI stream lean and prevents downstream devices from being overwhelmed by 14-bit traffic.
- **[REMOTE] Tag:** Incoming RPN/NRPN data is tagged as `[REMOTE]`, ensuring it updates internal storage and linked variables as it was a normal MIDI CC message.

5. Practical Examples

Example A: High-Precision Tuning

Linking the standard MIDI RPN #1 (Fine Tuning) to a VST knob for a "Master Tune" control.

```
// --- COMMON SECTION ---  
// Link RPN #1 (Fine Tune) on Ch 0 to VST Var #0 (Index 600)  
// Mode 3 provides high-resolution 0.0 to 1.0 scaling  
RPNLINK 3, 1, 0, 600
```

Example B: Custom Hardware Control

If a specialized hardware controller uses NRPN #100 for "Brightness," redirect it to a standard internal CC for modulation.

```
// --- COMMON SECTION ---  
// Link NRPN #100 on Ch 5 to standard CC 10 (Pan)  
// Mode 5 uses the MSB for compatibility with 7-bit triggers  
RPNLINK 5, 100, 5, 10
```

Developer's Note

If you find your internal parameters "stepping" or clicking when moved via MIDI, standard 7-bit CCs are likely the culprit. `RPNLINK` with **Mode 3 or 8** is the professional way to solve this. Even if the hardware controller only sends 7-bit data, using the 14-bit scaled mode provides a much higher internal resolution for the engine's DSP calculations.

SMOOTH <Fc>

To ensure your virtual instrument sounds professional and "analog," Crescendo provides the `SMOOTH` instruction. This applies a global low-pass filter to all continuous control data, effectively removing the "zipper noise" (audible stepping) that occurs when a 7-bit MIDI controller moves between its 128 discrete values.

This instruction sets the cutoff frequency (F_c) for the internal smoothing filter. Every time a MIDI CC or VST VAR changes, the engine doesn't jump to the new value instantly; it slides there at a speed determined by this frequency.

1. Technical Syntax

`SMOOTH <Fc>`

- **<Fc>**: The cutoff frequency in Hz.
 - **Default:** 10 Hz.
 - **Range:** \$0.001\$ Hz to \$\infty\$.
 - **Dynamic Variable:** Use \$600–\$727 to link the smoothing speed to a VST Variable (knob).
- **Logic:**
 - **Lower F_c (e.g., 1 Hz):** Creates a very slow, "heavy" feel. Parameters will take a noticeable amount of time to reach their target.
 - **Higher F_c (e.g., 50 Hz):** Very responsive and snappy, but may start to reveal MIDI stepping if the source data is coarse.

2. Operational Logic

- **Section Constraint:** Must be placed in the **COMMON** section.
- **The Unsmoothed Rule:** When you link `SMOOTH` to a VST VAR (e.g., `SMOOTH $600`), the engine uses the **raw, unsmoothed** value of that knob. This is a critical safety feature: if the smoothing was applied to the knob *controlling* the smoothing, you could get stuck in a "lag loop" where it becomes impossible to quickly turn the smoothing back up.
- **Global Impact:** This setting affects all continuous MIDI CCs and VST VARs throughout the instrument.

3. Strategic Examples

Example A: High-Response Performance

If your instrument relies on fast rhythmic filter cuts or volume "stuttering" via a mod wheel.

```
// --- COMMON SECTION ---  
// Fast response for rhythmic precision  
SMOOTH 40
```

Example B: Atmospheric Damping

For cinematic pads where any sudden jump in a filter or reverb mix would ruin the mood.

```
// --- COMMON SECTION ---  
// Heavy damping to ensure silk-smooth transitions  
SMOOTH 2
```

Example C: User-Controlled "Lag"

Giving the performer a "Response" knob on the GUI to dial in their preferred feel.

```
// --- COMMON SECTION ---  
// Define a knob for Smoothing (0.5Hz to 100Hz)  
VSTVAR 10, 10, "Control Lag", "Hz", 0.1, 50, 0  
  
// Link the engine smoothing to VST Var #10 (Address 610)  
SMOOTH $610
```

4. Developer's Summary Table

| Setting (Fc) | Feel | Best For... |
|-----------------|------------------|---|
| < 5 Hz | Damped / Laggy | Pads, slow swells, ambient textures. |
| 10 Hz (Default) | Balanced | Standard General MIDI style performance. |
| > 30 Hz | Snappy / Instant | Percussion, lead synths, fast automation. |

Developer's Note

While 10 Hz is a great middle ground, remember that standard MIDI (non-RPN) is quite low resolution. If you notice "clicks" when moving a volume or filter knob quickly, your first instinct should be to **lower** the `SMOOTH` frequency. Conversely, if the instrument feels "mushy" or unresponsive to your hardware sliders, try **raising** it.

Keyswitches

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. You can change a Keyswitch value with the GUI, the note keys assigned or with the MIDI CC message from 1000 to 1007, generated by an assignment, the sequencer or inputted from the MIDI Input.

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.

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"

Keyswitches are the professional standard for managing real-time performance variations. Instead of cluttering your interface with a hundred buttons, you can map specific keys on a MIDI controller to switch between articulations (like Legato, Staccato, or Pizzicato) or toggle internal engine states.

In Crescendo, these are declared in the **COMMON** section and act as global switches that can be read by any layer.

A keyswitch "hijacks" specific MIDI notes. Once a note is assigned to a keyswitch, it no longer produces sound; instead, it updates an internal value.

1. Technical Syntax

```
KEYSWITCH "Name", <type>, <initial_value>, <key1>, <key2>, <min>, <max>,
[<channel>], "Name 1", ..., "Name N"
```

- **"Name"**: The label for the GUI drop-box (e.g., "Articulation").
- **<type>**: The behavior of the switch:
 - **0 (Multiple Keys)**: A range from `key1` to `key2`. Each key is a specific "slot."
 - **1 (One Key)**: `key1` cycles through values from `min` to `max`.
 - **2 (Two Keys)**: `key1` increments, `key2` decrements.
- **<initial_value>**: The value or key active upon loading.
- **<key1>** / **<key2>**: MIDI note numbers (0–127).
- **<min>** / **<max>**: The value range (ignored for Type 0).
- **[<channel>]**: The MIDI channel to monitor (0–15). Defaults to 0.
- **"Name 1" ...**: Labels for each option in the GUI drop-box.

Each Keyswitch can have maximum 32 options.

The maximum number of keyswitches is 8.

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).

2. Keyswitch Types Comparison

| Type | Mode | Logic | Best Use Case |
|------|---------------|--|--|
| 0 | Multiple Keys | Each key in range [<code>key1</code> , <code>key2</code>] is a unique index. | Orchestral articulations (C-2 = Legato, D-2 = Staccato). |
| 1 | One Key | Tapping <code>key1</code> cycles: 1 to 2 to 3 ... to N to 1. | Sequential effect presets or simple toggles. |
| 2 | Two Keys | <code>key1</code> is "Up," <code>key2</code> is "Down." | Adjusting discrete levels like "Vibrato" |

| Type Mode | Logic | Best Use Case |
|-----------|-------|---------------|
|-----------|-------|---------------|

Intensity."

3. MIDI CC Addressing (The Secret Sauce)

Crescendo provides two ways to access keyswitch data programmatically:

- **The Trigger CC 400–527:**

These addresses correspond to MIDI notes. If you have a keyswitch on Note 36 (C1), reading `MCC(436)` in a layer will return the keyswitch's state.

- For **Type 0**, it returns 1 if that specific key is the active choice, 0 otherwise.
- For **Types 1/2**, it returns the current value within the `min/max` range.

- **The Unified Interface (CC 1000–1007):**

These 8 slots represent the 8 possible keyswitches.

- **Read/Write:** You can change a keyswitch value by setting the corresponding keyword, e.g. `MCC1000 = <EXPR>`.
- **Sync:** This is the format used for MIDI I/O and Sequencer recording.

4. Practical Example: Sample Slot Switching

This script uses a single key (C-2) to cycle through three different sample styles.

```
// --- COMMON SECTION ---

// Define Keyswitch #1: "Style"
// Type 1 (Cycle), Start at 200, Key 0 (C-2), Range 200-202
KEYSWITCH "Style", 1, 200, 0, 0, 200, 202, "Legato", "Staccato", "Pizzicato"

// Define samples for the three slots
SAMPLE 200, "legato_pno.wav", .....
SAMPLE 201, "stacc_pno.wav", .....
SAMPLE 202, "pizz_pno.wav", .....

// Global processing: Select sample based on the Keyswitch state (MCC 400)
// Note: MCC 400 looks at the value of the key used in the keyswitch
OUT = OSCG("Sgf000S", MCC(400), .01, .1, .8, .2)

// --- LAYER SECTION ---
LAYER
// Standard notes trigger on 1-127 (Note 0 is reserved for the keyswitch)
ONNOTEON 1, 127, 0, 127
```

Developer's Tips

- **Aftertouch is still alive:** Even though a keyswitch key won't trigger a note, the **Polyphonic Aftertouch** on that key is still sent to the engine. You can use this to "swell" an effect (like vibrato depth) *while* holding down the articulation key.
- **The "Deaf" Note:** Remember that keys assigned to a `KEYSWITCH` will not produce sound. If your instrument suddenly stops responding to "Middle C," check if you accidentally assigned it as a keyswitch!
- **GUI Sync:** Changes made via the physical MIDI keyboard will update the drop-box in the Crescendo GUI automatically.
- No check is performed to see if some interval of keys is overlapped among multiple `KEYSWITCH`s. In this case the behavior is undefined.
- A `[LOCAL]` MIDI message is issued for the `MIDI CC 999 + Keyswitch number`, so all the processing, like local storage updating, message merging in the `TAPE`, linked `MIDI CC`, `MIDI Output` is performed.

Polyphony, Glide, and Portamento

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 and can be automated. 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.

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

The `POLY` instruction is the "air traffic controller" of the Crescendo engine. It manages how many voices can speak at once, which ones get silenced when the room gets too crowded, and how the pitch slides between notes.

Whether you're building a 256-voice orchestral behemoth or a gritty, monophonic lead synth, `POLY` is where the behavior is defined.

This instruction dictates polyphony limits, note-stealing logic, and the mechanics of legato and portamento.

1. Technical Syntax

POLY <polyphony>, <discard>, <mode>, <glidetime>

- **<polyphony>**: The maximum number of simultaneous layers (1–256).
- **<discard>**: The logic used to "kill" a note when the polyphony limit is hit.
- **<mode>**: The behavior of the voice when a new note is triggered (Normal, Retrig, Portamento, or Glide).
- **<glidetime>**: The duration of the pitch transition in seconds (0–100).

2. Polyphony: Global vs. Per-Channel

Crescendo allows for two distinct ways to manage your voice pool:

- **Global Polyphony (Positive Value)**: POLY 64, ...

All 16 MIDI channels share a single pool of 64 voices. This is standard for most multi-timbral workstations.

- **Per-Channel Polyphony (Negative Value)**: POLY -1, ...

Each individual MIDI channel gets its own private pool. Setting this to -1 creates a "Monophonic" instrument on every channel—perfect for simulating a 6-string guitar (using 6 channels) or MPE (MIDI Polyphonic Expression) controllers.

3. Discard Policies & Operation Modes

When the engine runs out of voices, it needs a "hit list." You define that priority here.

Discard Policies (<discard>)

| Value | Policy | Best For... |
|-------|-------------------------|--|
| 0 | Oldest (Default) | Standard piano/pad playing. |
| 1 | Lowest Pitch | Ensuring high-frequency melodies aren't cut. |
| 2 | Highest Pitch | Preserving bass lines. |
| 3 | Least Intensity | Cleaning up quiet, decaying notes first. |

Operation Modes (<mode>)

| Mode Behavior | Description |
|-------------------------|--|
| 0 NORMAL | Every note is a "fresh" start. Phases and envelopes reset. |
| 1 RETRIG | Reuses an active voice if the formulas match. Envelopes continue from current level. |
| 2 PORTAMENTO | Glides from the nearest active pitch to the new pitch. |
| 3 GLIDE | Like Retrig, but slides the pitch without resetting the envelope. |

4. Implementation Examples

1. Standard Polyphonic Synthesizer

This configuration provides a generous voice pool and uses standard triggering logic.

```
// 128 global voices, discard oldest note, normal triggering, no glide  
POLY 128, 0, 0, 0
```

- **Polyphony (128):** The instrument shares a pool of 128 voices across all MIDI channels.
- **Discard (0):** When the 129th note is played, the oldest active note is terminated to free a slot.
- **Mode (0):** New notes are instantiated with fresh phases and envelope positions.

2. Monophonic Legato Lead (RETRIG Mode)

To emulate a monophonic lead synth with legato transitions, set the polyphony to 1 and utilize the RETRIG mode.

```
// 1 voice per channel, discard oldest, retrigger mode, 100ms glide  
POLY -1, 0, 1, 0.1
```

- **Polyphony (-1):** The negative sign specifies "per-channel" polyphony; each of the 16 MIDI channels can play exactly one note independently.
- **Mode (1):** If a note is already playing on a channel, the new note reuses the existing voice slot, maintaining its current envelope position and phase for a smooth legato effect.
- **Glidetime (0.1):** The pitch transitions from the old note to the new note over a duration of 100 milliseconds.

3. MPE-Optimized or Guitar Simulation

For instruments requiring per-string or per-note control, such as a guitar or an MPE (MIDI Polyphonic Expression) controller, use per-channel limitations.

```
// 6 voices per channel (simulating a 6-string guitar), discard lowest pitch
POLY -6, 1, 0, 0
```

- **Discard (1):** If the per-channel limit is reached, the note with the lowest pitch is sacrificed.

4. Fully Automated Performance Control

Crescendo allows you to link every parameter of the `POLY` instruction to VST variables (knobs) or MIDI CCs for real-time control during a performance.

```
// Automated via VST Variables 0, 1, and 2
POLY $600, 3, $601, $602
```

- **\$600 (Polyphony):** The maximum number of voices is determined by the value of VST Var #0.
- **Discard (3):** The engine sacrifices the note with the least current intensity.
- **\$601 (Mode):** Switches between Normal, Retrigger, Portamento, and Glide modes based on the position of VST Var #1.
- **\$602 (Glidettime):** Adjusts the glide duration (up to 100 seconds) using VST Var #2.

5. JP-8000 Style Portamento

This example uses the specialized `PORTAMENTO` mode to emulate the behavior of classic hardware samplers.

```
// 16 global voices, discard oldest, portamento mode, 0.5s glide
POLY 16, 0, 2, 0.5
```

- **Mode (2):** When a new note is triggered, the engine picks the newest active note with the nearest pitch and performs a pitch slide to the new note's frequency.

5. Developer's Note

Detailed parameters explanation:

<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 retrigger/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 or you could use automation (see below).
- 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 or you could use automation (see below). The total polyphony is still capped to 256.

- `<mcc>` means polyphony automation at trigger time with a MIDI CC.
 - `<mcc>`: 0-199. The automation is per channel, so potentially each channel can have different polyphony. To exploit this feature with VST Vars, just link the VST Vars with different channels of an user defined MIDI CC (e.g. 160) and use POLY \$160, ...
Only polyphony between 1 and 127 is possible with these MIDI CCs.
 - `<mcc>`: 600-727. The automation is shared with all channels.
 - True Polyphony is capped between 1 and 256.
 - Every unsupported MIDI CC number sets Polyphony to 256.
 - If the automation sets a Polyphony lower than the current notes number playing, the notes in excess are not put in release, but new notes will reuse the slots as per the `<discard>` and `<mode>` options below.

`<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>` policy 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 behavior as NORMAL.
If not, an occupied one that is compatible is selected with the `<discard>` policy.
If there is not a compatible voice, then it's the same as NORMAL: an occupied one with the `<discard>` policy 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 glidettime=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 glidettime=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 has 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 glidettime=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).

<glidettime> 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> or &<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.

NOTES:

- For automated operation <mode>, the MIDI CC is sampled at trigger time and from the same MIDI channel of the target note.
- For automated <glidetime>, the value is sampled at trigger time and from the same MIDI channel of the target note for the \$<MCC> syntax. For the &<MCC> syntax the automation is continuous.

Tip: the glide time is global for all notes and it seems that the automation is possible only with a simple MIDI CC or VST VAR.

But you can exploit the assignments: in the POST step you can place an assignment to e.g. VAR127 (use an unused and hidden VST Var) to set to it to a custom expression and use the &727 automation for <glidetime>: you will have continuous automation of the glide time with a custom expression.

Automated or continuous Glide samples the MIDI CC with a one sample delay, due to implementation constraints. The very first sample of a LAYER the glide time is set to an high value to not move the frequency. This should not be a problem since the first sample of any envelope is always zero, so no artifact would be heard.

Due to the same implementation constraint, if there is an instruction that modifies the MIDI CC or VST VAR in the layer, e.g. to have custom continuous automation, it does not matter where it is put in the LAYER: in any case all OSCGs that are to be glided will use the calculated glide time.

The current implementation will perform each layer processing per block, to exploit the cache, so the automation is stored at the end of each sample in a temporary buffer per LAYER, initialized with an high value at layer creation. This means that every OSCG that glides, will use the glide time calculated at the previous sample. This means also that if you play a four-note chord, the code within the LAYER block is executed four times per sample. Each time it runs for a specific voice, it updates the MIDI CC or VST VAR with that voice's unique value and the value is stored in the temporary buffer, that is private to that layer, ready to be used the next sample to calculate that voice's pitch. This also means that each note will maintain its own independent, "elastic" glide speed without interference from other simultaneous notes.

See the tutorial section for an example.

NOTES for 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.
- The gliding works only if your oscillators use in some way KEYF, FREQ or FREQ0, e.g. the default mapping (KEYI is not suitable because it's rounded, unless you want fretted glide, but you can use ROUNDf or ROUNDf2 functions). If you want a custom mapping that glides, then you must derive your custom frequency from one of these Keywords. Note

that KEYI and KEYF normally contain the virtual key recalculated from the current frequency (FREQ0) assuming BASEF = 440 Hz, KEYCENTER = 69.0 and KEYTRACK =1. Take this into account when calculating your custom mapping or derive it directly from FREQ0 or FREQ.

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 among two release times effectively implementing a dampen pedal.

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

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

In the world of Crescendo, the `PEDAL` and `SOSTENUTO` instructions are more than just MIDI filters—they are high-level DSP switches. While they emulate the physical pedals of a piano, their internal logic is a clever "cheat": **when a pedal is active, the engine swaps the Release time for the Decay time.**

The Mechanic: Release Time Swapping

Unlike a simple MIDI "Note Off" blocker, these instructions allow you to define two different ways a note can end.

- **Normally:** When you release a key, the sound follows the `Release` stage of the envelope.
- **Pedal On:** When you release a key, the sound follows the `Decay` stage instead.

This is why, in a well-designed instrument, you typically set a **long Decay** (the ringing piano string) and a **short Release** (the damper hitting the string).

1. Technical Syntax

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

- **<mcc1>:** The MIDI CC that controls the **Global** pedal (entire keyboard).
- **<mcc2>:** The MIDI CC that controls a **Zone-Specific** pedal.
- **<minkey>; <maxkey>:** The range of notes (0–127) affected by `<mcc2>`.
- **-1:** Use this to disable a slot or to ensure a clean slate at the start of your script.

2. Trigger Thresholds

Crescendo looks for specific values to determine if the "foot" is down:

| Input Type | Index Range | "ON" Condition |
|------------------|-------------|------------------|
| Standard MIDI CC | 0 – 199 | Value > 63 |
| Keyswitches | 400 – 527 | Value > 0 |
| VST Variables | 600 – 727 | Value \geq 0.5 |
| Disabled | Any other | Always OFF |

3. Sustain vs. Sostenuto: The "Moment of Truth"

The primary difference lies in *when* the engine checks which notes to hold:

- **PEDAL (Standard Hold):** This is a state. As long as it's ON, any note currently playing—or played in the future—will be sustained.
- **SOSTENUTO:** This is a snapshot. When the pedal moves from OFF to ON, the engine "grabs" only the notes currently being held down. Notes played *after* the pedal is down will release normally.

4. Advanced: The "Dampen" Pedal Strategy

You can use these instructions to create a "Dampen" pedal for instruments like vibraphones or dry percussion.

1. Set your Envelope **Sustain to 1.0**.
2. Set the **Decay** to a long time (the "Open" sound).
3. Set the **Release** to a short time (the "Dampened" sound).
4. Activate the PEDAL.
 - When the pedal is **ON**, the note uses the Decay time (it rings out).
 - When the pedal is **OFF**, the note uses the Release time (it is dampened immediately).

5. Practical Examples

A. The Traditional Piano Setup

Standard CC 64 for Sustain and CC 66 for Sostenuto.

```
// --- COMMON SECTION ---
// CC 64: Global Sustain
PEDAL 64; -1; 0; 127
// CC 66: Global Sostenuto
SOSTENUTO 66; -1; 0; 127
```

B. The Bass-Only Sustain (Split)

Useful for synth players who want to hold a bass drone with a pedal while playing staccato leads on top.

```
// CC 64 affects only notes 0 to 48 (C2 and below)
PEDAL -1; 64; 0; 48
```

C. Safety Reset

```
PEDAL -1; -1
SOSTENUTO -1; -1
```

D. GUI Control via VST Variables

```
// Define VST Knobs for Sustain and Sostenuto
VSTVAR 0, 0, "HOLD", "", 0, 1, 4, "OFF", "ON"
VSTVAR 1, 0, "SOSTENUTO", "", 0, 1, 4, "OFF", "ON"
// Link VST knobs (indices 600 and 601) to the pedal instructions
PEDAL 600
SOSTENUTO 601
```


In this context, the pedal is considered "ON" if the VST Variable value is 0.5 or greater.

E. Dampen Pedal Simulation

Because these instructions essentially select between two different release rates, they can be used to implement a "dampen pedal" effect if the envelope sustain level is set to 1.0. This allows a performer to toggle between two distinct release behaviors (e.g., a short, dampened release versus a long, ringing release).

Example Strategy:

1. Configure an envelope with `Sustain = 1.0`.
2. Set a specific `Decay` time for the "un-dampened" sound and a `Release` time for the "dampened" sound.
3. Activate the pedal to use the decay time as the active release time.

Developer's Note

If you find your notes are cutting off too abruptly even with the pedal down, check your Envelope settings. Remember: **the pedal doesn't just "hold" the note; it tells the note to use the Decay parameters.** If your Decay time is set to 0.1s, your pedal will feel like it isn't working!

Multichannel considerations:

If using per channel MIDI CC (0-199), 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. This is usually not a problem because most probably you will use standard MIDI CCs (64 and 66).

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 seven consecutive steps.

The processing order is fixed, but the instructions below are declarative, so they can be specified in any order in the file (except that the ordering of the MIDIPOST instructions is important).

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 most instructions 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.

NOTES:

- 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).
 - 0-199 standard and some extended MIDI CC. The effect is enabled if the MIDI CC value is greater than 64. Some MIDI CC don't have a range suitable, like Pitch Bend and temperament if there are few temperaments and some MIDI CC do not work at all, like the per-layer.
 - 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.
- 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 0-199. The other MIDI CCs are global).
- **Some other parameters are automatable**, at trigger time. Check the instruction description to see which of it (if any). The actual value used is not scaled for VST VARs and is scaled with a

value given in the parameter description for MIDI CC below 400. Those parameters can be automated with a custom formula with the same trick of glide time: an hidden VST VAR, modified in the POST step with a custom formula.

- The current temperament detunes are applied at the end of the 6-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.

- The KEY, KEYF, FREQ, FREQ0, GAIN, GAIN0, 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.
- If the sequencer is enabled (see below), then both ARPEGGIO and CHORD are disabled.

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.
- **MIDIPOST:** A full-fledged programmable step with a set of assignments, to modify note, velocities, trigger time, trigger duration and aftertouch in an unlimited complex manner.

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

The ARPEGGIO instruction takes a single note and explodes it into a rhythmic sequence.

1. Syntax

```
ARPEGGIO <cc>; <min>; <max>; <size>; <s1>; <d1>; <dur1>; <v1> ... <sN>; <dN>;  
<durN>; <vN>
```

- **<size>**: The total loop length in **BARs**.
 - *Positive (e.g., 1.0)*: Stops immediately when you release the key.
 - *Negative (e.g., -1.0)*: The loop finishes its current cycle even if you release the key.
 - 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.
- **The Note Block (<semi>, <delay>, <duration>, <velmul>)**:
 - **semi**: Semitone offset from the held note (can be fractional/floating point).
 - **delay**: Delay in bars from the moment the key was pressed.
 - **duration**: How long the note held in bars.
 - **velmul**: Velocity multiplier (\$0.0\$ to \$1.0+\$) based on the original hit.
 - The last note, if incomplete, takes the default values of 0,.25,1 for delay, duration and velmul.
 - 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.

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.

A key hit triggers the ARPEGGIO.

Each key can trigger a different simultaneous ARPEGGIO, with different base note.

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.

2. Critical Rules

1. **Mutual Exclusion**: You cannot use ARPEGGIO and CHORD at the same time. Setting one disables the other.
2. **The Trigger Note**: You must specify the first note manually in the sequence (usually `semi=0, delay=0`) if you want the original note to sound.
3. **Loop Regeneration**: As soon as the last note in your list is triggered, the engine checks if the key is still held. If it is, it calculates the next "loop" of notes instantly.
4. **Polyphony Warning**: Each note in an arpeggio consumes a voice slot. High-speed arpeggios with long durations can hit the POLY limit very quickly.

3. Examples:

The Cinematic Minor "Pulse"

This arpeggio plays a 1/8th note minor pattern that completes its loop even if the key is tapped quickly.

```
// --- COMMON SECTION ---
// CC: -1 (Always on)
// Keys: 0-127 (Whole keyboard)
```

```
// Size: -0.5 (Half-bar loop, completes on release)
// Notes: Root (0), Minor 3rd (+3), 5th (+7), Octave (+12)
ARPEGGIO -1, 0, 127, -0.5, 0,0,.1,1, 3,.125,.1,0.9, 7,.25,.1,0.8, 12,.375,.1,1.1
```

Simple Major Arpeggio

```
// 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
```

Developer's Note

The ARPEGGIO timing is sensitive to the **Sequencer** and **TIMING** steps. If you have a global delay in the TIMING step, it will push the start of your arpeggio. Additionally, because the engine creates a whole loop at a time, very sudden tempo changes from your DAW might cause the "end" of a long arpeggio loop to feel slightly out of sync until the next loop re-triggers.

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.

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.

Pedals and sostenuto modify only the release time of the notes: only the note off are used to stop an arpeggio.

The initial OFF velocities for all notes is the corresponding ON velocity. These values can be changed by the next steps, though.

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

The CHORD instruction allows you to play complex harmonies with a single finger. Every note generated by this instruction is treated as an independent voice with its own pitch and velocity, but they all share the same trigger time and life cycle as the original key press.

1. Technical Syntax

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

- **<cc>; <min>; <max>**: The standard activation gate. (e.g., -1; 0; 127 for always on).
- **<semiI>**: The semitone offset from the pressed key.
 - 0 is the root note.
 - 4 is a major third.
 - 7 is a perfect fifth.
 - *Fractional values* (e.g., 4.5) are supported for microtonal chords.

- **<velmul>**: Velocity multiplier for that specific note in the chord.
 - Useful for creating "weighted" chords where the top note is louder than the inner voices.

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.

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

2. Critical Rules

1. **Mutual Exclusion:** If you declare a CHORD, it automatically disables any previously declared ARPEGGIO.
2. **Explicit Root:** You **must** include 0, 1 in your list if you want the note you actually pressed to sound. If you don't, the "root" will be silent, and only the shifted notes will play.
3. **Note Lifecycle:** When you release the physical key, all generated notes enter the release phase simultaneously.

3. Practical Examples

A. Classic Major Triad (Lower Octaves)

This creates a "one-finger" piano accompaniment for the left hand (notes below Middle C).

```
// --- COMMON SECTION ---
// CC: -1 (Always on)
// Keys: 0 to 59 (Up to B2)
// Notes: Root(0), Major 3rd(+4), 5th(+7)
CHORD -1, 0, 59, 0,1, 4,1, 7,1
```

B. Weighted Jazz "Shell" Chord

This generates a root, a 7th, and a 10th (major 3rd an octave up), with the higher notes slightly quieter to prevent masking the melody.

```
// Root (0), Major 7th (11), Major 10th (16)
// The 10th is at 80% velocity
CHORD -1, 0, 127, 0,1, 11,0.9, 16,0.8
```

C. Safety Reset

To ensure no chords or arpeggios are active (important when merging multiple script files):

CHORD -1, -1, -1

Developer's Note

Keep an eye on your `POLY` settings! If you have a `POLY` limit of 8 and you play a 4-note chord with both hands, you have used up your entire voice pool. If you add an `OSCG` with multiple layers on top of that, you will experience heavy note-stealing.

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.

Pedals and sostenuto modify only the release time of the notes: only the note off are used to stop the chord notes.

The initial OFF velocities for all notes is the corresponding ON velocity. These values can be changed by the next steps, though.

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

The `MAP` instruction looks at the semitone of an incoming note (C, C#, D, etc.) and replaces it with a semitone you define. It preserves the octave but changes the "note name" to fit your desired scale.

1. Technical Syntax

MAP <cc>; <min key>; <max key>; <C>; <C#>; <D>; <D#>; <E>; <F>; <F#>; <G>; <G#>; <A>; <A#>;

- `<cc>; <min>; <max>`: Standard activation gate.
- `<C>...`: Twelve values (0–11) representing what each semitone should become.
 - 0=C, 1=C#, 2=D, 3=D#, 4=E, 5=F, 6=F#, 7=G, 8=G#, 9=A, 10=A#, 11=B

2. Why use MAP?

If you use the `ARPEGGIO` example from Step 1, playing a **C** results in a **C Major** arpeggio. However, if you move up to **A**, the engine would normally play an **A Major** arpeggio. If your song is in C Major, that C# in the A Major chord will sound "wrong."

By using `MAP`, you force that C# to become a C (the minor third), automatically turning your A Major arpeggio into an **A Minor** arpeggio to fit the song's key.

3. Practical Examples

A. The "White Keys Only" (C Major / A Minor Scale)

This maps every accidental (black key) down to the nearest natural note (white key).

```
// --- COMMON SECTION ---
// Semitones: C  C# D  D# E  F  F# G  G# A  A# B
// Mapping:    0  0  2  2  4  5  5  7  7  9  9  11
MAP -1, 0, 127, 0, 0, 2, 2, 4, 5, 5, 7, 7, 9, 9, 11
```

B. The C minor Blues Scale

Force the keyboard to only play the Blues scale notes: C, Eb, F, F#, G, Bb.

```
// C maps to C(0), C# to C(0), D to Eb(3), D# to Eb(3), E to F(5)...
MAP -1, 0, 127, 0, 0, 3, 3, 5, 5, 6, 7, 7, 10, 10, 10
```

C. Octave "Folding" (Experimental)

You can map every note to just C and G to create a "Power Chord" drone instrument regardless of what keys are hit.

```
// Map everything to either C (0) or G (7)
MAP -1, 0, 127, 0, 0, 0, 7, 7, 7, 0, 0, 7
```

D. The C Dorian Map, e.g. for playing Scarborough Fair

Here is the MAP instruction. It redirects every "black key" accidental to the correct note in the C Dorian scale:

```
// --- COMMON SECTION ---
// Semitones: C  C# D  D# E  F  F# G  G# A  A# B
// Mapping:    0  0  2  3  3  5  5  7  7  9  10 10
MAP -1, 0, 127, 0, 0, 2, 3, 3, 5, 5, 7, 7, 9, 10, 10
```

How this map behaves:

- **C# / Db** is pulled down to **C**.
- **D#** is mapped to **3 (Eb)**.
- **E natural** is also pulled down to **3 (Eb)** to ensure you don't accidentally play a Major 3rd.
- **F#** is pulled down to **F**.
- **G#** is pulled down to **G**.
- **A#** is mapped to **10 (Bb)**.
- **B natural** is also pulled down to **10 (Bb)**.

4. Pipeline Logic: Order Matters

Because MAP happens **after** ARPEGGIO and CHORD, it is incredibly powerful:

1. **Input:** You press "A".
2. **Step 1 (ARPEGGIO):** Generates A, C#, E.
3. **Step 2 (MAP):** Sees the C#, checks the C Major map, and changes it to C.
4. **Output:** A, C, E (A Minor).

Developer's Note

If you map two different input semitones to the same output semitone (e.g., mapping both C and C# to C), and you play both keys at once, the engine will trigger **two separate voices** on the note C. This can cause phase cancellation or unintended volume boosts. Be sure CHOKe is activated.

For more complex microtonal tuning (where you need more than 12 semitones per octave), Crescendo supports .scl (Scala) files, which we can discuss later.

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>

The VELOCITY instruction applies a mathematical transformation to the velocity value (v_{in}) of Note ON and/or Note OFF messages.

1. Technical Syntax

VELOCITY <cc>; <min key>; <max key>; <min_in>; <max_in>; <min_out>; <max_out>; <drive>; <spread>; <flag>; <mul0>; <mul63>; <mul127>

- **<min_in>; <max_in>**: The "Window." Only notes hit with a velocity in this range are processed.
- **<min_out>; <max_out>**: The "Re-mapping." Rescales the window to these new output limits.
- **<drive>**: The "Curve." 1.0 is linear; > 1.0 is exponential (harder to hit high velocities); < 1.0 is logarithmic (very sensitive).
- **<spread>**: The "Humanizer." Adds a random value between +-spread.
- **<flag>**: 0 = Note ON, 1 = Note OFF, 2 = Both.
- **<mul0/63/127>**: "Key Scaling." Multiplies velocity based on keyboard position (e.g., making the high notes of a piano naturally quieter).

2. The Mathematical Formula

The engine processes the note in three sub-steps:

1. **Key Scaling:** v_{in} is multiplied by a value interpolated between `mul0`, `mul63`, and `mul127` based on the key number.
2. **Transformation:**

$$v_{out} = \left(\left(\frac{v_{in} - min_in}{max_in - min_in} \right)^{drive} \times (max_out - min_out) \right) + min_out + spread$$

3. **Clamping:** The final v_{out} is clipped to the valid MIDI range of [0, 127].

3. Strategic Examples

A. The "Studio Compressor" (Limiting Range)

Perfect for a pop synth where you want the sound to stay loud and consistent even if the player is inconsistent.

```
// Input 0-127 is compressed into 100-127. Everything sounds "Loud."  
VELOCITY -1, 0, 127, 0, 127, 100, 127, 1, 0, 0, 1, 1, 1
```

B. The "Weighted Key" Feel (Drive)

If your MIDI controller feels "cheap" or too "light," setting a higher drive makes the instrument feel like it has heavy, weighted wooden keys.

```
// Drive of 2.5 makes it much harder to reach top volume  
VELOCITY -1, 0, 127, 0, 127, 0, 127, 2.5, 0, 0, 1, 1, 1
```

C. Keyboard Balancing (Grand Piano Style)

High notes on a real piano have less mass and naturally sound "thinner" or quieter than the booming bass strings.

```
// Low keys (mul0) are full strength, High keys (mul127) are scaled to 70%  
VELOCITY -1, 0, 127, 0, 127, 0, 127, 1, 0, 0, 1, 1, 0.7
```

D. Real-Time Performance Control

By linking parameters to VST Variables (\$600+), you can allow the player to adjust the "Response" and "Randomness" of the instrument from the plugin's UI.

```
// --- COMMON SECTION ---  
// Configure the UI with two knobs for velocity control  
VSTVARS 2  
// VST Var #0: Adjusts the non-linear drive (0.5 to 4.0)  
VSTVAR 0, 1.0, "Curve/Drive", "", 0.5, 4.0, 1  
// VST Var #1: Adjusts the random humanization spread (0 to 15)  
VSTVAR 1, 0, "Randomness", "", 0, 15, 2  
  
// Configure the VELOCITY modifying step  
// <cc>: -1 (Effect is always active for all keys)  
// <min key>;<max key>: 0, 127 (Applies to the full keyboard)  
// <min vel in>;<max vel in>: 0, 127 (Processes all incoming velocities)  
// <min vel out>;<max vel out>: 0, 127 (Output uses the full velocity range)  
// <drive>: $600 (Linked to VST Variable #0)  
// <spread>: $601 (Linked to VST Variable #1)  
// <velocity flag>: 0 (Apply to Note ON only)  
// <mul0>;<mul63>;<mul127>: 1, 1, 1 (No key-based scaling)  
  
VELOCITY -1, 0, 127, 0, 127, 0, 127, $600, $601, 0, 1, 1, 1
```

Functional Analysis

1. **Selection:** The control CC is set to -1, ensuring the velocity processor remains active regardless of external controller states.

2. **Dynamic Drive:** As the user rotates the "Curve/Drive" knob, the `<drive>` parameter is updated. If a note is struck while the knob is at 2.0, the engine applies an exponential curve, making high velocities harder to reach.
3. **Humanization:** The "Randomness" knob controls the `<spread>`. At a value of 10, the engine adds a random value between -10 and +10 to every triggered note's velocity, simulating human performance variations.
4. **Trigger-Time Sampling:** Because the `$` syntax is used, the engine captures the positions of VST Var #0 and #1 only at the moment of the `NOTE ON` event. Moving the knobs while a note is currently sounding will not affect the velocity of that specific active note but will influence all subsequent notes.
5. **Formula Integration:** The final velocity (`vout`) is calculated using the formula:
$$vout = ((vin - min_in) / (max_in - min_in)) ^ drive * (max_out - min_out) + min_out + spread.$$

E. GUI-Controlled Toggle via VST Variables

You can link the enablement of the velocity effect to a knob on the Crescendo interface.

```
// Setup a VST Toggle Knob
VSTVAR 0, 0, "Humanize", "", 0, 1, 4, "OFF", "ON"

// Use VST Var #0 (MCC 600) to control the effect
VELOCITY 600, 0, 127, 0, 127, 0, 127, 1, 15, 0, 1, 1, 1
```

The effect is only active if VST Variable #0 is set to "ON" (value ≥ 0.5).

Developer's Note

The `<spread>` parameter is a powerful way to breathe life into mechanical MIDI sequences. Even a small value of 5 ensures that no two notes in a repeating pattern (like an 8th-note bass line) are ever exactly the same, which effectively masks the "machine-gun effect."

`<drive>` is automatable at trigger time with the syntax `$<MCC>`. The scale factor for MIDI CCs below 400 is 0.1.

`<spread>` is automatable at trigger time with the syntax `$<MCC>`. The scale factor for MIDI CCs below 400 is 0.01.

`<mul0>`, `<mul63>`, `<mul127>` are automatable at trigger time with the syntax `$<MCC>`. The scale factor for MIDI CCs below 400 is 0.01.

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

The `PITCH` instruction modifies the floating-point pitch value of a note. Because Crescendo uses fractional semitones, you can transpose by exactly one octave or by a tiny, "out-of-tune" fraction.

1. Technical Syntax

```
PITCH <cc>; <min key>; <max key>; <shift>; <spread>; <chance>; <rounding>;  
[<qstrength>]
```

- **<shift>**: The fixed transposition in semitones (e.g., -12 for an octave down).
- **<spread>**: The maximum random offset (e.g., 0.02 for a tiny drift).
- **<chance>**: How often the random spread is applied (0.0 to 1.0).
- **<rounding>**: The "grid" for quantization. 1.0 rounds to the nearest semitone; 12.0 rounds to the nearest octave.
- **<qstrength>**: How "hard" the rounding pulls (0–100%). At 50%, a note halfway between C and C# is pulled only 25% of the way toward the target.

2. Strategic Applications

A. The Analog "Drift" (Vintage Character)

Real analog synths are rarely perfectly in tune. You can simulate this by adding a tiny bit of random pitch spread to every note.

```
// Apply a tiny pitch drift (+/- 0.03 semitones) 100% of the time  
PITCH -1, 0, 127, 0, 0.03, 1.0, 0
```

B. Stepped "Bit-Crushed" Pitch

By using a high **<rounding>** value, you can create a "stepped" effect where the pitch only jumps in specific intervals, similar to a modular synth sample-and-hold effect.

```
// Force all notes to jump in increments of 3 semitones (Minor Thirds)  
PITCH -1, 0, 127, 0, 0, 0, 3.0, 100
```

C. The "Auto-Tune" Effect (Partial Quantization)

If you have an input source that is slightly "off" (like a fretless MIDI controller or a poorly calibrated sequencer), you can gently pull it toward the nearest semitone.

```
// Round to the nearest semitone, but only with 40% strength  
// (keeps some of the original "human" detuning)  
PITCH -1, 0, 127, 0, 0, 0, 1.0, 40
```

D. Transposing our Dorian Tutorial

Remember our **Scarborough Fair** setup? Most people find **D Dorian** easier to sing than C Dorian. Instead of rewriting our MAP, we can simply use PITCH to shift the whole performance.

```
// --- COMMON SECTION ---  
// Shift the entire Dorian setup up 2 semitones (from C to D)  
PITCH -1, 0, 127, 2, 0, 0, 0
```

```
// Map stays the same (it processes the original notes)  
MAP -1, 0, 127, 0, 0, 2, 3, 3, 5, 5, 7, 7, 9, 10, 10
```

E. Fixed Global Transposition (Octave Shift)

To shift the entire MIDI input up by exactly one octave (12 semitones) without any randomization, use the following:

```
// Transpose all keys (0-127) up by 12 semitones
PITCH -1, 0, 127, 12, 0, 0, 0
```

In this scenario, the shift is always applied because the control CC is set to -1.

F. Pitch "Humanization" for Bass Notes

You can introduce subtle, random pitch drift to a specific region of the keyboard—such as the low bass notes—to simulate the organic instability of an analog instrument.

```
// Apply a random spread of +/- 0.05 semitones to keys below C2 (0-36)
// This spread has a 75% chance (0.75) of occurring for each Note ON
PITCH -1, 0, 36, 0, 0.05, 0.75, 0
```

This configuration adds character to a performance by ensuring that every low note is not perfectly in tune.

G. GUI-Controlled Transposition Toggle

By linking the instruction to a VST variable, you allow the end-user to toggle a transposition effect directly from the plugin interface.

```
// COMMON SECTION
// Define a VST Toggle Knob (Knob #0)
VSTVAR 0, 0, "Shift +7", "", 0, 1, 4, "OFF", "ON"

// Use VST Var #0 (Index 600) to enable a 7-semitone shift
PITCH 600, 0, 127, 7, 0, 0, 0
```

The pitch shift will only trigger if the "Shift +7" knob is set to "ON" (value ≥ 0.5).

H. Performance-Controlled Pitch Correction

In this scenario, we use two VST knobs to allow a performer to adjust global transposition and the "tightness" of pitch quantization (pulling out-of-tune notes toward the nearest semitone) in real-time.

```
// --- COMMON SECTION ---
// 1. Declare two VST Variables for the GUI
VSTVARS 2
// Knob #0: Transposition (-12 to +12 semitones, integer scale)
VSTVAR 0, 0, "Transpose", "semi", -12, 12, 2
// Knob #1: Correction Strength (0 to 100%)
VSTVAR 1, 100, "Correction", "%", 0, 100, 1

// 2. Configure the PITCH instruction with automated parameters
// <cc>: -1 (Effect is always active)
// <min/max key>: 0, 127 (Applies to the full keyboard)
// <shift>: $600 (Automated by VST Var #0)
// <spread/chance>: 0, 0 (No randomization used here)
// <rounding>: 1.0 (Quantize to the nearest whole semitone)
// <qstrength>: $601 (Quantization intensity automated by VST Var #1)

PITCH -1, 0, 127, $600, 0, 0, 1.0, $601
```

Functional Analysis

1. **Trigger-Time Selection:** Because the \$ prefix is used, the engine "freezes" the values of the Transpose and Correction knobs the instant a NOTE ON message is received. Moving the knobs while a note is sustaining will not change the pitch of that active note, but will affect all subsequent notes.
2. **Transposition Logic:** If VST Var #0 is set to -5, the note is shifted down five semitones before any further processing occurs.
3. **Partial Quantization:** The <rounding> parameter of 1.0 creates a target grid of whole semitones. If the incoming MIDI note is slightly sharp (e.g., 60.4) and the "Correction" knob (VST Var #1) is set to 50%, the engine calculates a weighted mean, resulting in a final pitch of 60.2.
4. **Pipeline Position:** This automated pitch processing occurs as the fifth step in the MIDI pipeline, meaning it affects the values used by the `FREQ` and `KEYF` keywords in the **LAYER** sections.

Developer's Note

Defaults are -1;-1;-1;0;0;0

<shift> is automatable at trigger time with the syntax \$<MCC>. The scale factor for MIDI CCs below 400 is 1.

<spread> is automatable at trigger time with the syntax \$<MCC>. The scale factor for MIDI CCs below 400 is 0.01.

<chance> is automatable at trigger time with the syntax \$<MCC>. The scale factor for MIDI CCs below 400 is 0.01.

<rounding> is automatable at trigger time with the syntax \$<MCC>. The scale factor for MIDI CCs below 400 is 1.

<qstrength> is is automatable at trigger time with the syntax \$<MCC>. The scale factor for MIDI CCs below 400 is 1.

The quantization is performed before giving the NOTE to an eventual MIDIPOST step.

**TIMING <cc>; <min key>; <max key>;
<delay>; <on spread>; <duration spread>;
<quantization>; [<qstrength>]**

The **TIMING** instruction controls two major factors: **Humanization** (adding randomness) and **Quantization** (snapping to a grid).

1. Technical Syntax

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

- **<delay>**: A fixed offset in **bars** (e.g., 0.25 = a quarter note delay).
- **<on spread>**: Random positive delay added to the Note ON.

- **<duration spread>**: Random positive length added to the note's life.
- **<quantization>**: The grid size in bars (e.g., 0.0625 for 1/16th notes).
- **<qstrength>**: How "magnetically" the notes snap to the grid (0–100%).

2. The "Architecture" Warning

Crescendo handles timing in a specific way:

1. **Humanization/Delay** happens first.
2. **MIDIPOST** (the final scripting step) can then modify these times further.
3. **Quantization** is performed as the **very last step** in the chain.

This means even if you use a script to mess with timing, the `TIMING` instruction's quantization grid will have the final say.

3. Strategic Examples

A. 1/16th Note "Pocket" (Partial Quantization)

This is great for drum machines. You want it to be tight, but not "computery." By using 70% strength, you keep some of the original "swing" or "push/pull" of the player.

```
// Grid: 0.0625 (1/16th note), Strength: 70%
TIMING -1, 0, 127, 0, 0, 0, 0.0625, 70
```

B. The "Loose" Analog Sequencer

Simulate a vintage sequencer where the clock isn't quite perfect.

```
// Add a small random spread to the onset and the duration
TIMING -1, 0, 127, 0, 0.01, 0.01, 0
```

C. Fixed Latency Compensation

If you have a Layer that uses a sample with a "pre-roll" (like a violin bow stroke that starts before the actual note), you can delay the rest of the keyboard to match.

```
// Delay everything by 0.02 bars to allow slow-attack samples to sync
TIMING -1, 0, 127, 0.02, 0, 0, 0
```

D. Interactive Rhythmic Control

You can give the performer a "Tightness" knob on the UI. This is incredibly effective for MPE or live keyboard performances.

```
// Knob 0: How much random "slop" (0 to 0.04 bars)
// Knob 1: How much "Snap" to the 1/8th note grid (0 to 100%)
VSTVAR 0, 0, "Humanize", "", 0, 0.04, 1
VSTVAR 1, 100, "Snap", "%", 0, 100, 1

// Automation: $600 for spread, $601 for strength
// Grid: 0.125 (1/8th note)
```

```
TIMING -1, 0, 127, 0, $600, 0, 0.125, $601
```

E. GUI-Controlled Humanization Toggle

By linking the `TIMING` instruction to a VST toggle, you allow the end-user to enable or disable rhythmic humanization in real-time.

```
// COMMON SECTION
// Define a VST Toggle Knob (Knob #0)
VSTVAR 0, 0, "Humanize", "", 0, 1, 4, "OFF", "ON"

// Use VST Var #0 (Index 600) to enable a 1/64 bar humanization spread
TIMING 600, 0, 127, 0, 0.015625, 0.015625, 0
```

The timing modifications will only occur if the "Humanize" knob is set to "ON" (value ≥ 0.5).

F. Configurable partial quantization

In this example, the `TIMING` instruction is configured to apply a 60% quantization strength to a 1/16th note grid, which tightens the performance while retaining some of the original rhythmic character.

```
// --- COMMON SECTION ---
// Configure TIMING modification for all keys (0-127)
// cc: -1 (Always enabled)
// quantize: 0.25 (1/16th note grid based on 1.0 = Quarter note)
// offset: 0 (No constant time shift)
// qstrength: 60 (Snap notes 60% of the way to the grid)

TIMING -1, 0, 127, 0.25, 0, 60
```

G. Performance-Controlled Rhythmic Feel

In this scenario, two VST knobs are defined to allow the performer to adjust rhythmic "Humanization" (random start-time drift) and "Quantization Strength" (the tightness of the snap to a 1/16th note grid) in real-time.

```
// --- COMMON SECTION ---
// 1. Declare two VST Variables for the interface
VSTVARS 2
// Knob #0: Rhythmic Humanization (0 to 0.05 bars)
VSTVAR 0, 0, "Humanize", "bars", 0, 0.05, 1
// Knob #1: Quantization Tightness (0 to 100%)
VSTVAR 1, 100, "Snap %", "%", 0, 100, 1

// 2. Configure the TIMING modification step
// <cc>: -1 (Effect is always active)
// <min/max key>: 0, 127 (Applies to all keys)
// <delay>: 0 (No constant offset)
// <on spread>: $600 (Random start drift automated by VST Var #0)
// <duration spread>: 0 (No random duration drift)
// <quantization>: 0.0625 (Target a 1/16th bar grid: 1.0 / 16 = 0.0625)
// <qstrength>: $601 (Quantization intensity automated by VST Var #1)

TIMING -1, 0, 127, 0, $600, 0, 0.0625, $601
```

Functional Analysis

1. **Trigger-Time Capture:** Because the \$ prefix is used for the spread and strength parameters, the engine "freezes" the current positions of the "Humanize" and "Snap %" knobs the instant a NOTE ON message is received.
2. **Rhythmic Humanization:** If VST Var #0 is set to 0.02, the engine adds a random positive delay between 0 and 0.02 bars to the note's start time.
3. **Partial Quantization:** The <quantization> value of 0.0625 targets the nearest 1/16th note boundary. If VST Var #1 is set to 50, the engine calculates a weighted mean between the humanized start time and the perfect grid position, effectively pulling the note 50% of the way toward the grid.
4. **Pipeline Integration:** This final calculated time is assigned to the DTRIG (delta trigger) internal variable. This adjusted timing is then passed to the sound-generating layers, ensuring that all time-dependent elements—such as envelopes and sample playback—commence at the newly calculated moment.

Developer's Note

Note that <on spread> and <duration spread> are **always positive**. If you need a note to trigger *earlier* than the MIDI message, you must implement a global <delay> first, then subtract from it or use the random spread to move notes "forward" within that delay window.

Defaults are -1;-1;-1;0;0;0

<delay> is automatable at trigger time with the syntax \$<MCC>. The scale factor for MIDI CCs below 400 is 0.1.

<on spread> is automatable at trigger time with the syntax \$<MCC>. The scale factor for MIDI CCs below 400 is 0.1.

<duration spread> is automatable at trigger time with the syntax \$<MCC>. The scale factor for MIDI CCs below 400 is 0.1.

<quantization> is automatable at trigger time with the syntax \$<MCC>. The scale factor for MIDI CCs below 400 is 0.01.

<qstrength> is automatable at trigger time with the syntax \$<MCC>. The scale factor for MIDI CCs below 400 is 1.

The quantization is performed after the MIDIPOST step. This is an architectural limitation. The MIDIPOST processing of the timings is performed after that of this step, but the quantization is performed as last step, so it is not available in the MIDIPOST instruction: you must use the TIMING step to quantize. If the timing processing of the TIMING step is enough for your needs, then you don't need the MIDIPOST step. Otherwise, set the quantization and quantization strength on the TIMING step and use the MIDIPOST step to perform custom timing processing

MIDIPOST 0 OR MIDIPOST “assignment”

MIDIPOST is executed at the very end of the Note ON/OFF sequence. It allows you to manipulate six core variables that determine exactly how a note will behave across different layers.

1. The Core Variables

These variables are floating-point and represent the state of the note *after* steps 1-5 have already acted on it:

| Variable | Description | Impact |
|----------|--|------------------------------------|
| NOTE | The pitch in semitones (69.0 = A3). | Affects frequency and tuning. |
| ONVEL | Note ON velocity (scaled 0.0 to 1.0). | Affects dynamics and mapping. |
| OFFVEL | Note OFF velocity (scaled 0.0 to 1.0). | Affects release behavior. |
| AFTER | Polyphonic aftertouch (scaled 0.0 to 1.0). | Affects pressure-based modulation. |
| DTRIG | Delta Trigger in Bars . | Introduces custom delays. |
| DDUR | Delta Duration in Bars . | Shortens or prolongs the note. |

2. Syntax & Execution

- `MIDIPOST 0`: Clears any existing program (best practice to start your script with this).
- `MIDIPOST "assignment"`: Adds a line of code to the program.

The program runs **once for every layer** at Note ON (and once at Note OFF). This is critical: if you use a random function (`RND` or `NOISE`), each layer can receive a slightly different pitch or timing, allowing for massive "unison" or "multi-tracked" sounds from a single MIDI note.

3. Advanced Logic with EXECIF

You can apply logic selectively based on the **original** key and velocity hit by the player (before steps 1-5 modified them).

Syntax: `MIDIPOST "EXECIF <minkey>, <maxkey>, <minvel>, <maxvel> <assignment>"`

See below for details.

4. Practical Creative Examples

A. The "Velocity-Sensitive Detune"

In physical instruments, striking a string harder often causes a momentary sharp pitch. You can simulate this by linking `ONVEL` to `NOTE`.

```
MIDIPOST 0
// Harder strikes (ONVEL) add up to 0.1 semitones of random sharpness
MIDIPOST "NOTE = NOTE + (ONVEL * RND(0.1))"
```

B. Layer-Specific "Strumming"

You can create a "pseudo-strum" effect by giving each layer a slightly different trigger delay.

```
// Each layer is delayed by a different random amount up to 0.05 bars
MIDIPOST "DTRIG = DTRIG + RND(0.05)"
```

C. Fixed-Key Functionality (The Percussion Trick)

If you want a specific key (like C2 / Key 36) to always play at a fixed velocity regardless of how hard it is hit (common for kick drums):

```
// Force Key 36 to always have a velocity of 0.8
MIDIPOST "EXECIF 36, 36, 0, 127 ONVEL = 0.8"
```

D. Configurable MIDI Humanization

The following "Hello World" sample demonstrates a humanization effect where the amount of randomness is controlled by VST knobs.

```
// --- COMMON SECTION ---
VSTVARS 3
VSTVAR 0, .1, "VEL.SPREAD", "", 0, 1, 1 // Knob for velocity variation
VSTVAR 1, .1, "TRIG.SPREAD", "", 0, 1, 1 // Knob for timing variation
VSTVAR 2, .1, "DUR.SPREAD", "", 0, 1, 1 // Knob for duration variation

// Reset the MIDI POST program
MIDIPOST 0

// Modify velocity using a random range based on VST Var #0
MIDIPOST "ONVEL = ONVEL + RND(VAR0)"

// Introduce a custom trigger delay (DTRIG) based on VST Var #1
MIDIPOST "DTRIG = ABS(RND(VAR1))"

// Adjust note duration (DDUR) based on VST Var #2
MIDIPOST "DDUR = DDUR + RND(VAR2)"
```

E. Note-Specific Logic using EXECIF

You can use the `EXECIF` prefix within the `MIDIPOST` program to apply transformations only to specific keys or velocity ranges. The values used for these checks are the raw, original MIDI inputs.

```
// Only add a trigger delay to notes below Middle C (Key 60)
MIDIPOST "EXECIF 0, 59, 0, 127 DTRIG = 0.25"

// Only boost the velocity of notes played very softly (Velocity 0-20)
MIDIPOST "EXECIF 0, 127, 0, 20 ONVEL = ONVEL * 2.0"
```

5. What You Can't Do

MIDIPOST is designed for "Snapshot" logic at the moment of a hit. Because it runs once per note, it **does not** support:

- Oscillators, Filters, or Envelopes.
- IF...GOTO or LABEL (use EXECIF instead).
- Functions with "memory" (it is invoked only 2 times).

Developer's Note

If you are using TAPE (pre-recorded MIDI) or ARPEGGIO, the Note OFF logic is calculated simultaneously with the Note ON. This means your OFFVEL and DDUR adjustments happen instantly when the note is spawned.

This instruction is used to define a MIDI POST processing program, executed as sixth last step (before the temperament on Note ON) after a Note ON or Note OFF message is received. This instruction can be only in the COMMON section.

The assignments are similar to the normal assignments, but support less functionalities (a list of supported and not supported features is given below): they are executed for one sample at Note ON and one sample at Note OFF and they act on monophonic items and so the expression is always monophonic.

The instructions are executed at each Note ON and Note OFF in the order as they are declared. There can be other instructions between the MIDIPOST instructions, but only that instructions matter for the MIDI POST program.

At Note ON and Note OFF your MIDI POST program has at its disposal all the normal MIDI CCs (standard, extended, VST VARs, Keyswitches, Temperament, some keywords), plus those predefined variables mentioned above, all floating point:

The MIDI POST program instructions are executed in order, one time for each LAYER (so if an instruction have the RND or RNDN function or use the NOISE MIDI CC, you can have different values for each LAYER), each time with the variables above reset to the initial value. An instruction can be skipped if it is prefixed with an EXECIF clause and the conditions are not satisfied. RANDOM can be used as MIDI CC to skip all instructions together or none of them. NOISE MIDI CC can be used for deciding each time (see the description below).

The purpose of this program is to rewrite, with simple conditional assignments, those six variables. The values of those variables at the end of the program are transplanted in the corresponding LAYER variables, except polyphonic aftertouch, that is shared among all the LAYERS belonging to the same note and can be modified by external Polyphonic Aftertouch messages that can arrive after the Note On or Note OFF message is processed (you can disable at least the Polyphonic messages at the MIDI Input with the MCCCTL instruction).

These new values are then used in the LAYER formulas. In particular the first five are sensed at Note ON, and can modify, e.g. the note pitch played, trigger, duration, etc.

Only the OFFVEL is sensed at the Note OFF, but in any case all other variables are filled also at the NOTE OFF, in the case they are needed for OFFVEL calculation, but are ignored after the program execution.

You can always issue normal instruction of the type `MCCnnnn = <expr>` in the program to force the modify of another MIDI CC on each Note ON or OFF. The only limitation is that normally ONVEL and OFFVEL are not supported on the left on assignments. Only here they are supported and only with the keywords given.

The unsupported features in the MIDI Program language are:

IF...GOTO/EXIT, LABEL, HOLD, PREV, oscillators, envelopes, filters, delays, reverbs, strings and functions with string parameters and in general functions that have some memory or are designed with audio data in mind.

The supported functionalities are:

EXECIF prefix, Standard operators, MCCnnnn, MCC(nnn), VARnnnn, VAR(nnn), MCC2, MCC3, CURVE, MOD2, RPN, RPN2, RND, RNDN, ABS, SAT, SAT2, SIGM, SIGMDW, LOG, EXP, LIN2DB, DB2LIN, SEMI, CENTS, ROUND, ROUNDf, ROUNDf2, CEIL, FLOOR, FRAC, QUANTIZE, QUANTIZE2, SIGN, COPYSIGN, POWABS, trigonometric functions.

Notes:

- The values used in the checks for key and velocities for the EXECIF clause are the original values received in the NOTE ON message, before the MIDI POST process steps.
- The MIDI CC and VST VARS are not smoothed and the MIDI CC with number below 400 (excluded 0, 32, 134, 135, 144, 155 and 160-199 that are direct mapped) are divided by 128 like MCC2 in any case, even with MCC syntax.
- The MIDI Output receives those fully modified parameters. So the Crescendo VST can be used as a full MIDI processor, producing or modifying all types of MIDI messages.
 - Crescendo supports different scaling per layer if using the internal temperaments (see below). The scaling employed for the MIDI Output notes is the COMMON one to avoid to choose one. When using external files for temperaments, all layers have the same keyboard mapping, so this problem does not arise (see below).
- To save the modified notes you could attach a VST (including Crescendo) to the MIDI Output able to save the MIDI CC messages, since those messages are modified by the MIDI POST processing steps.
- When playing from a TAPE (or producing an arpeggio), the only difference from a live note is that the Note OFF is already known at Note ON and the note is created already with the release set and never get released by a physical note OFF: so both processing are performed back to back when the note is issued from the TAPE (or the arpeggio). The other MIDI CCs available in the program will reflect this fact (e.g. if you use a varying MIDI CC, the value used at Note OFF for non-live notes is the same seen at Note ON).

Temperament Step

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

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 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.

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

Understanding Microtuning and Temperaments

In digital synthesis, a **Temperament** defines the precise mathematical tuning of each interval within a scale.

While the modern world primarily uses **Equal Temperament (12-TET)**—where the octave is divided into 12 identical semitones—Crescendo allows you to explore the rich harmonic landscapes of historical and experimental tunings.

Why Use Alternative Temperaments?

- **Harmonic Purity:** Standard tuning (Equal) is a compromise; most intervals are slightly "out of tune" to allow playing in every key. "Fancy" temperaments like **Just Intonation** or **Kirnberger III** provide perfectly tuned thirds and fifths. This significantly reduces "harmonic beating" (interference), resulting in a cleaner, more resonant sound.
- **Tonal Character (Key Color):** In historical temperaments, every key has a unique "mood." C Major might sound pure and stable, while F# Major might sound tense or brilliant. This adds a layer of emotional depth that is lost in standard tuning.
- **Synergy with Effects:** When using the **SUPERSAW** or a **CHORUS** in the `POST` layer, a pure temperament prevents the "blurring" of frequencies, allowing the modulation to sound more organic and choir-like.

Technical Considerations in Crescendo

- **Sample-Accurate Resampling:** When using **SAMPLES**, Crescendo's high-quality interpolation (`QUALITY` instruction) ensures that even the smallest microtonal deviations (fractions of a cent) are reproduced without aliasing or artifacts.
- **Note-Level Consistency:** The temperament is assigned at the moment of the `Note ON`. If you switch temperaments while holding a chord, the active notes will maintain their original tuning, preventing jarring pitch jumps and ensuring a professional performance.
- **Phase Alignment:** Pure intervals in non-equal temperaments allow the harmonics of different notes to align perfectly in phase. This creates a "physical" presence in the sound that is especially noticeable when using high-order filters or resonance.

The MIDI "HOST" Mode

Crescendo includes a unique **HOST** temperament. This mode allows the plugin to listen to micro-detuning data sent by advanced DAWs (per-note detune).

If your DAW supports it, Crescendo becomes a transparent extension of your host's tuning engine.

A Guide to Historical & "Fancy" Temperaments

If you are new to microtuning, here is a brief overview of the default temperaments provided in the configuration and what they bring to your sound:

- **Just Intonation:** The "Holy Grail" of purity. It uses simple integer ratios (like 3:2 or 5:4). Intervals sound incredibly still and clear. It is perfect for slow-moving pads, drones, and meditative soundscapes where you stay within a specific key.
- **Meantone (1/4 Comma):** The sound of the Renaissance and early Baroque. It offers beautiful, pure Major Thirds at the expense of a "Wolf Fifth" (a very dissonant interval) on the far side of the circle of fifths. It makes your synth or organ sound like a time machine to the 16th century.
- **Kirnberger III:** A personal favorite for pipe organs. It provides a "Well-Tempered" bridge between the purity of the past and the flexibility of the future. The "natural" keys (C, G, F) are extremely resonant and pure, while distant keys gain a subtle "shimmer" and tension.
- **Werckmeister (III, IV, V):** These are the tunings that made J.S. Bach's music possible. They are designed to allow playing in all 24 keys while maintaining a distinct "personality" for each one. Werckmeister III is particularly famous for its balanced yet colorful harmonic movement.
- **Vallotti & Young:** These are the most versatile "circulating" temperaments. They sound very close to modern Equal Temperament but with more "breath" and natural resonance in the most common keys. They are the standard choice for modern Baroque ensembles and orchestral sample libraries.

Dynamic Temperament Blending

While historical temperaments offer harmonic purity, they can sometimes feel too rigid for modern production. Crescendo introduces a unique **TEMPERAMENTBLEND** engine that allows you to crossfade between the active Temperament and standard Equal Temperament (12-TET) in real-time.

- **Tonal Morphing:** You can start a note in standard tuning and slowly "settle" it into a pure historical temperament as it sustains. This creates a physical sense of "locking in" the harmony, which is especially effective for pads and strings.
- **Performance Expression:** Using MIDI CCs or VST Variables, you can map the blend factor to your playing style. For example, use **Flag 1 (Triggered)** to link the blend to Velocity, so that harder-struck notes are tuned more purely than softer ones.
- **Correcting "Wolf" Intervals:** If a specific key in a historical temperament sounds too dissonant for a particular chord, you can use the blend factor to momentarily drift back toward Equal Temperament, "taming" the dissonance without switching scales entirely.
- **MPE Integration:** In MPE mode, each note can have its own independent blend factor. This allows for unprecedented polyphonic control, where different voices in the same chord can follow different tuning trajectories.

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/folder path";
"KBM file path"/<middle key>**

In Western music, we typically use **Equal Temperament (12-TET)**, where every semitone is exactly the same distance apart. However, this is a compromise. Crescendo allows you to break this grid to achieve harmonic purity or historical accuracy.

1. Default Options

- **HOST (0):** Listens to per-note detune data from your DAW. If your DAW doesn't send this, it acts as Equal Temperament.
- **EQUAL (1):** Standard modern tuning. Every interval is slightly "out of tune" so you can play in every key equally.

2. Technical Syntax: Defining Custom Tunings

```
TEMPERAMENT "Name"; <C>; <C#>; <D>; ... ; <B>; [<middle_key>]
```

- **Cents Deviation:** If you provide small numbers (e.g., 0; 2; -1...), the engine assumes these are deviations from Equal Temperament.
- **Absolute Cents:** If you provide an increasing pattern (e.g., 0; 100; 204...), it assumes these are absolute distances from the root.
- **<middle_key>:** The keyboard center for the tuning (Default: 60 / Middle C).

Example: A "Purer" Major Third

```
// Flatten the Major 3rd (E) by 14 cents to make it "Just" (purer)
// C C# D D# E (flat) F ...
TEMPERAMENT "Just-ish"; 0; 0; 0; 0; -14; 0; 0; 0; 0; 0; 0; 0
```

3. Microtonal Imports (SCALA & TUN)

For complex systems (like Middle Eastern scales or 19-tone scales), you can import industry-standard files.

- **SCALA (.scl):** Defines the ratios of the scale.
- **KBM (.kbm):** Defines how those ratios map to your physical keyboard.
- **TUN (.tun):** AnaMark files containing precise frequency maps for every key.

Example: Bulk Importing a Library

```
// Scan a folder and add every tuning file found to the VST dropdown
TEMPERAMENT "C:\MySamples\TuningLibrary\"
```

4. Dynamic Switching

You can change the temperament of your instrument mid-performance using its Name or Index.

Important: Changing temperaments while a note is held will **not** cause a pitch jump. The new tuning only applies to the *next* Note ON.

```
// Triggered via a script or a UI button
TEMPERAMENT "Kirnberger III"
```

Developer's Note

If you are using **SAMPLES**, the Temperament step uses high-quality interpolation (controlled by the `QUALITY` instruction) to shift the sample pitch. This ensures that even if you are using a microtonal scale that requires a note to be sharp by only 3.5 cents, the sample will play back with phase-accurate precision and no "chipmunk" artifacts.

This instruction is for adding new temperaments to the list of user selectable temperaments. It can be put only in the `COMMON` section.

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 16384 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: a [LOCAL] MIDI message is issued, so all the processing, like local storage updating, message merging in the TAPE, linked MIDI CC, MIDI Output is performed.

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 notes 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].

NOTE: if you are using internal temperaments, i.e. not using external SCALA or TUN files, you can have a tempered stretched instrument, e.g. a stretched piano. Just set KEYTRACK to a value different than 1, e.g. 1.01. The note with the correct tuning will be A3(69) by default. To change this, change BASEF and KEYCENTER accordingly.

Examples:

1. Default Settings and Selection

By default, the engine always defines two temperaments: **HOST** (Index 0) and **EQUAL** (Index 1). You can switch between existing temperaments using their numerical index or their descriptive name.

Example: Selecting by Index or Name

```
// Sets the current tuning to the HOST temperament (Index 0)
TEMPERAMENT 0
```

```
// Alternatively, select EQUAL temperament by name
TEMPERAMENT "EQUAL"
```

If the host DAW supports detuning parameters in Note ON messages, the **HOST** setting will use that data; otherwise, it behaves identically to **EQUAL** temperament.

2. Defining Custom Scales using Cents

You can create a bespoke temperament by specifying detuning values in cents for each of the twelve semitones in an octave, starting from C.

Example: Defining a "Just Intonation" Style Scale

```
// Defines a new temperament named "PureMajor"  
// Values represent detuning in cents relative to the root note  
TEMPERAMENT "PureMajor"; 0; 10; 20; 30.5; 40; 50; 60; 70; 80; 90; 100; 110
```

The engine automatically detects if you are providing absolute cents from the root note or deviations from equal temperament based on the numerical pattern provided. You may optionally specify a <middle_key> (default is 60 or Middle C) to determine where the scale starts.

3. Importing External Tuning Files

Crescendo supports the import of **SCALA (.scl)** and **AnaMark (.tun)** files to implement complex microtonal systems.

Example: Single File Import

```
// Imports a specific SCALA file relative to the instrument path  
TEMPERAMENT "SCALES\Persian.scl"  
  
// Imports an AnaMark tuning file  
TEMPERAMENT "tunings\vintage_synth.tun"
```

If the provided string is a valid file path, the engine imports the file and creates a new temperament entry using the filename as the label.

4. Advanced Mapping and Bulk Imports

For sophisticated microtonal setups, you can combine scale files with **Keyboard Mapping (.kbm)** files or import entire directories of tuning data.

Example: Scale with Keyboard Mapping

```
// Applies an Arabian scale with a specific physical keyboard layout  
TEMPERAMENT "SCALES\Arabic.scl"; "KBM\Standard_C.kbm"
```

Example: Bulk Folder Import

```
// Scans the "GlobalScales" folder and imports all .scl and .tun files found  
TEMPERAMENT "C:\My Tuning Library\GlobalScales\"
```

Bulk imports are processed during instrument compilation, automatically populating the temperament dropdown menu in the plugin interface.

TEMPERAMENTCC <number>, [<channel>]

This instruction creates a bridge between the temperament engine and your MIDI controllers, keyswitches, or VST automation.

1. Technical Syntax

TEMPERAMENTCC <number>, [<channel>]

- **<number>**: The control source (MIDI CC, Keyswitch, or VST Var).
- **<channel>**: The MIDI channel to listen to (default is 0).

2. Control Mapping Options

The engine behaves differently depending on the range of the <number> you provide:

| Range | Control Type | Behavior |
|---------|--------------|---|
| -1 | GUI Only | Disables remote control; you must use the mouse in the VST interface. |
| 0–127 | Standard CC | Map a MIDI CC (like Bank Select). |
| 400–527 | Keyswitch | Reserves a MIDI key (0–127). Pressing this key cycles to the next temperament. |
| 600–727 | VST Variable | Links to a VST Knob. The knob label automatically changes to show the temperament name. |

3. Strategic Examples

A. The "Live Performer" Keyswitch

If you are playing a live set and need to switch from *Equal Temperament* to *Just Intonation* for a specific song section, you can map it to your lowest MIDI key.

```
// Reserve MIDI Key 0 (C-2) to cycle through your tuning list
TEMPERAMENTCC 400
```

Note: Key 0 will no longer produce sound; it is now a dedicated "tuning button."

B. The "Studio Producer" Automation

By linking the temperament to a VST Variable, you can automate tuning changes directly in your DAW's timeline.

```
// --- COMMON SECTION ---
// Link to VST Variable #1 (Knob 601)
// Place this AFTER your temperament definitions!
TEMPERAMENTCC 601
```

When a VST variable is used, the engine automatically configures the knob to a linear scale that covers exactly the number of defined temperaments. Furthermore, the knob's label is dynamically updated in the GUI to display the name of the currently active temperament.

To ensure the control range is calculated correctly, it is recommended to place this instruction after all `TEMPERAMENT` definitions in your script.

C. GUI-Only Selection (Disabling External Control)

If you wish to prevent accidental tuning changes from external MIDI data or automation, you can restrict temperament selection to the graphical user interface.

```
// Disable external MIDI/VST control for temperament selection
TEMPERAMENTCC -1
```

Setting the control number to -1 or any value outside the recognized ranges ensures that temperaments are selectable only via the mouse in the plugin interface.

4. Critical Behavioral Rules

1. **Note Persistence:** If you switch temperaments while holding a chord, the notes you are currently holding **stay in the old tuning**. Only new notes played *after* the switch will use the new temperament. This prevents "pitch-shifting" artifacts mid-note.
2. **Global Scope:** Temperament is global for the entire VST instance. You cannot have Channel 1 in *Mean Tone* and Channel 2 in *Equal* within a single Crescendo instance.
3. **Bidirectional Sync:** If you move the VST knob, the MIDI CC updates. If you send a MIDI CC, the GUI dropdown updates. They are always in sync.

Developer's Note

This instruction links the temperament drop box with an extended MIDI CC (including `KEYSWITCHes`).

It can be put only in the `COMMON` section.

<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 and the `NOTE ON` and `OFF` messages are not retransmitted in the MIDI output. 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.

- A [LOCAL] MIDI message is issued, so all the processing, like local storage updating, message merging in the TAPE, linked MIDI CC, MIDI Output is performed.

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

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.

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.

TEMPERAMENTBLEND <midicc>, <flag>,[<target b selector>]

The TEMPERAMENTBLEND instruction enables real-time interpolation between two distinct tuning states. This allows for dynamic "morphing" between a primary temperament and a secondary target.

1. Syntax

TEMPERAMENTBLEND <blend_cc>, <flag>, [<target_b_selector>]

- **<blend_cc>**: The MIDI CC or VST Var (0–727) that controls the interpolation.
 - **Value 0.0**: The output is **100% Target B** (Secondary).
 - **Value 1.0**: The output is **100% Target A** (The primary temperament selected in the GUI or via the TEMPERAMENT command).
- **<flag>**: Determines the sampling and smoothing logic.
 - **Flag 0**: Blending disabled. The actual temperament will be the current.
 - **Flag 1 (Triggered)**: Targets are sampled only at **Note On**.
 - **Flag 2 (Continuous)**: Real-time update with internal smoothing for fluid transitions.
- **[<target_b_selector>] (Optional)**: A pointer to a VST Parameter (MIDI CC 600–727 or 0–127) used to select the **Secondary Temperament (Target B)**.
 - **Mapping**: If you provide a value in the range **0–127**, the engine interprets it as a VST Parameter number (mapped to the 600–727 range).
 - **Default**: If omitted, Target B defaults to **ID 1 (Equal Temperament)**.
 - **Restriction**: Target B **cannot** be set to ID 0 (Host). It must be an internal temperament or a loaded scale (ID 1+).

2. Workflow and ID Mapping

- **The ID-Prefix System:** To facilitate scripting, the GUI dropdown prefixes every temperament with its internal ID (e.g., 1-Equal, ...).
- **Target Selection:** The value held by the VST Parameter defined in `<target_b_selector>` determines which scale is active for Target B. For example, if VST Var 1 holds the value 1.0, Target B will be *Equal Temperament*.

3. Examples

Example A: Global Morphing between two Scales Morphing from **Werckmeister III** (Target A) to **Meantone 1/4** (Target B) using VST Variables.

```
// --- COMMON SECTION ---
VSTVAR 0, 0.0, "A/B Morph", "" // VST Var 0 (MCC 600): Blend Factor
VSTVAR 1, 3.0, "Secondary ID", "" // VST Var 1 (MCC 601): Target B Selector (ID 3)

TEMPERAMENT "Werckmeister III" // Set Target A (ID 22)
TEMPERAMENTBLEND 600, 2, 601 // Blend between ID 22 and ID 3
```

Example B: Simple 12-TET Blend A script that simply blends the primary temperament with Equal Temperament.

```
// --- LAYER SECTION ---
// Target B defaults to ID 1 (Equal).
// Value 0.0 = Equal, Value 1.0 = Just Intonation.
TEMPERAMENT "Just Intonation"
TEMPERAMENTBLEND 600, 2
```

Technical Implementation Notes

- **Parameter Pointers:** The third argument is a **selector**. It does not take a Temperament ID directly; it takes the *number of the VST Parameter* that holds the ID.
- **Host Tuning Limitation:** Target B is strictly reserved for internal temperament definitions (ID 1+). This ensures a stable mathematical reference for the interpolation and avoids potential logic loops with DAW-hosted tuning.
- **Performance:** While the Global engine provides string feedback in the UI, per-layer overrides utilize direct ID mapping. This is to consume less CPU.
- **Just Intonation Test:** When the blend factor is **1.0**, intervals like the Major Third in a C Major chord will align perfectly with their harmonic ratios (e.g., the E will be approximately 14 cents lower than in 12-TET).
- **Internal Smoothing:** In Continuous mode (**Flag 2**), the engine applies a high-quality smoothing filter to the CC input to ensure that rapid automations results in a fluid pitch glide rather than discrete steps.
- **Interaction with KEYTRACK:** This instruction operates downstream of **KEYTRACK**. While **KEYTRACK** stretches the overall keyboard, **TEMPERAMENTBLEND** shifts the individual note offsets according to the selected scale file or definition.
- **Pro Tip:** For maximum precision and easier debugging, map your temperament logic to **VST Variables (MCC 600+)**. This bypasses the 7-bit MIDI limitation and uses a direct 0.0–1.0 floating-point scale, ensuring perfect alignment with historical temperament offsets.
- **Note on Frequency Processing:** **TEMPERAMENTBLEND** is applied directly to the base frequency (**FREQ0**). Any subsequent modulations defined via **FREQMOD** (such as Pitch Bend,

LFOs, or Envelopes) are calculated relative to this tempered base. This ensures that expressive performances maintain the harmonic integrity of the selected historical scale.

- **MPE & Multi-Layer Flexibility:** In MPE mode, `TEMPERAMENTBLEND` operates per-voice. If multiple layers are active, each can maintain an independent blend factor. This allows for complex textures where different timbral components of the same note can follow different tuning trajectories (e.g., a stable tempered fundamental layered with an evolving equal-tempered harmonic texture).

MIDI Instructions, stage III: 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 from 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>

This is the most fundamental instruction for any layer. It defines the physical range on the keyboard and the touch sensitivity (velocity) required to hear this specific sound.

Technical Syntax

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

- **<minkey>, <maxkey>**: The range of MIDI notes (0–127).
- **<minvel>, <maxvel>**: The range of velocity (0–127).

Note: This instruction is unique because it **can only be placed inside a LAYER section**, never in the COMMON section.

Strategic Use Cases

A. Keyboard Splits

To create a "Bass" sound on the left hand and a "Piano" on the right, you use ONNOTEON to divide the keyboard.

```
LAYER 1 // Bass Layer
ONNOTEON 0, 59, 0, 127 // Only plays below Middle C
... instructions

LAYER 2 // Piano Layer
ONNOTEON 60, 127, 0, 127 // Only plays Middle C and above
... instructions
```

B. Velocity Switching

This is the secret to realistic acoustic instruments. You can trigger a "Soft" sample when playing lightly and a "Hard" sample when striking the keys with force.

```
LAYER 1 // Soft Flute
ONNOTEON 0, 127, 0, 60
... instructions

LAYER 2 // Overblown Flute
ONNOTEON 0, 127, 61, 127
... instructions
```

ONMCCT <number>,<min>,<max>

This instruction defines a condition. As said before, if you have three ONMCCT triggers, all three must be within their specified ranges for the layer to trigger.

1. Technical Syntax

ONMCCT <number>, <min>, <max>

- **<number>**: The MIDI CC or extended CC number (0–1007).
- **<min> and <max>**: The active range (inclusive).
 - For standard MIDI CCs, use **0.0 to 127.0**.
 - For VST Variables (600–727), use the range defined in the VSTVAR declaration (usually **0.0 to 1.0**).

2. Strategic Applications

A. The "Staccato/Legato" Keyswitch

You can use a specific MIDI CC (like CC 4, often used for Foot Controllers) to switch between different articulations.

```
LAYER 1 // Legato Strings
ONMCCT 4, 0, 63 // Only triggers when pedal is UP

LAYER 2 // Staccato Strings
ONMCCT 4, 64, 127 // Only triggers when pedal is DOWN
```

B. GUI-Based Layer Enable

By linking a trigger to a VST Variable (Knob), you can let the user turn specific sound layers on or off from the plugin interface.

```
// --- COMMON SECTION ---
VSTVAR 0, 1, "Sub Oscillator", "", 0, 1, 4, "OFF", "ON"

// --- LAYER SECTION ---
LAYER 1 // Sub Bass
ONMCCT 600, 0.5, 1.0 // Only plays if VST Knob #0 is set to "ON"
```

C. Complex Texture Shifting

You can stack multiple ONMCCT triggers to create highly specific conditions. For example, a "shimmer" layer that only plays if the **Mod Wheel** is high **AND** the **Sustain Pedal** is pressed.

```
LAYER 1 // Shimmer Pad
ONMCCT 1, 100, 127 // Mod Wheel (CC 1) must be high
ONMCCT 64, 64, 127 // Sustain Pedal (CC 64) must be pressed
```

3. Critical Rules to Remember

- **Sample-at-Trigger:** The engine checks the value of the MIDI CC at the **exact millisecond** the Note ON occurs.
 - *Note:* Moving the knob *after* the note has already started will not turn a layer on or off—it only affects the *triggering* of new notes. (To fade active notes, use XFADE instead).
- **Deletions:** If you need to remove an ONMCCT trigger defined in an earlier file, use the "impossible range" trick: ONMCCT 1, 127, 0.
- **Global CC Source:** The CC value used is always the one coming from the same MIDI channel as the triggering note.

Developer's Note

Because ONMCCT supports extended CCs, you can trigger layers based on **Keyswitches** (CC 400-527). This allows you to set up a "Mode" for your instrument where pressing C-2 in the past determines which layers play now.

ONFIRST <min>,<max>

ONLEGATO <min>,<max>

These triggers allow Crescendo to distinguish between the very first note of a phrase and the subsequent notes played while holding the previous ones. This is the foundation for realistic solo instruments like violins or mono-synths.

1. ONFIRST <min>,<max>

- **Logic:** Valid only if **no other notes** are currently held down within the specified key range.
- **Use Case:** Triggering a "hard attack" or "bow start" sample at the beginning of a musical phrase.

2. ONLEGATO <min>,<max>

- **Logic:** Valid only if **at least one other note** is already held down.
- **Use Case:** Triggering "slurred" or "transition" samples that lack a sharp attack, creating a seamless connection between notes.

```
LAYER 1 // Violin - Hard Attack
ONFIRST 0, 127 // Plays only at the start of a phrase
```

```
LAYER 2 // Violin - Legato Transition
ONLEGATO 0, 127 // Plays only when notes overlap
```

Developer's Note

Release Phase: In ONFIRST and ONLEGATO calculations, notes that are in the "Release" phase (after you let go of the key but while the sound is still fading) are **not counted** as active.

AND Logic: Remember that these are added to your ONNOTEON and ONMCCT triggers. For a legato layer to play, it must be in the right key range **AND** satisfy the legato condition.

ONRROBIN <remainder>,<divisor>

Round Robin prevents the "machine-gun effect" by cycling through different samples of the same note. Even if you hit the same key repeatedly, the sound changes slightly each time.

Syntax: ONRROBIN <remainder>,<divisor>

- **<divisor>**: The total number of variations in your cycle.
- **<remainder>**: The specific "slot" this layer occupies (starting at 0).

Example: A 3-step Round Robin

```
LAYER 1 // Snare Hit A  
ONRROBIN 0, 3
```

```
LAYER 2 // Snare Hit B  
ONRROBIN 1, 3
```

```
LAYER 3 // Snare Hit C  
ONRROBIN 2, 3
```

Developer's Note

The counter advances on every Note ON for that channel, regardless of whether other triggers (like velocity) were met.

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 channel 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.

AND Logic: Remember that these are added to your ONNOTEON and ONMCCT triggers. For a Round Robin layer to play, it must be in the right key range **AND** satisfy the Round Robin condition.

ONCHANNEL <min>,<max>

If you are building a multi-timbral instrument (where different MIDI channels play different sounds within the same instance), use ONCHANNEL.

Syntax: ONCHANNEL <min>,<max>

- **Logic:** The layer only wakes up if the MIDI Note ON message comes from a channel within this range (0–15).

```
LAYER 1 // Flute on Channel 1
ONCHANNEL 0, 0
```

```
LAYER 2 // Oboe on Channel 2
ONCHANNEL 1, 1
```

Developer's Notes

AND Logic: Remember that these are added to your `ONNOTEON` and `ONMCCT` triggers. For a layer to play, it must be in the right key range **AND** satisfy the `CHANNEL` condition.

OFF Triggers

GROUP <number>

The `GROUP` system creates a social hierarchy among your layers. By assigning layers to groups, you can allow them to "talk" to each other and trigger silences.

GROUP <number>

- **Range:** -1 to 100,000.
- **Scope:** Can be set in `COMMON` (sets the default for all layers) or inside a `LAYER` (sets it only for that layer) but **NOT** in `POST` section (because pertains to `LAYERs`).
- **-1:** The layer is an "introvert" and belongs to no group.

This declaration set the group of layers, useful in triggers (see triggers above).

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.

OFFGROUP <number>,<forcedrelease>

This is a "listener." If *any* layer belonging to the specified <number> is triggered, this layer will immediately enter its release phase.

- **<forcedrelease>:** If >0, it overrides the layer's normal envelope release with this value (in seconds). This is crucial for avoiding audio "pops" during sudden stops.
- **Exclusive Mode:** If a layer has an `OFFGROUP` for the **same** group it belongs to, it creates a monophonic behavior. New notes will kill previous notes of that same layer.
- If the note just triggered has the `OFFGROUP` trigger with the same group it belongs to, 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.

```
// Example: Traditional Mono-Synth behavior
LAYER
GROUP 1
OFFGROUP 1, 0.01 // Every new note kills the previous one with a 10ms fade
ONNOTEON 0, 127, 0, 127
OUT = GAIN * OSCG("s1f0000")
```

OFFMCCT <number>, <min>, <max>, <forcedrelease>

Unlike ONMCCT, which is only checked at the moment of a hit, **OFFMCCT** is a "live watcher." It monitors your MIDI controllers or VST knobs constantly while a note is playing.

Syntax: OFFMCCT <number>, <min>, <max>, <forcedrelease>

- **Logic:** If the value of CC <number> enters the range between <min> and <max>, the note is instantly released.
- **Use Case:** A "Panic" knob, a "Kill Switch" on a mod wheel, or a sustain pedal that, when released, specifically kills an "infinite" ambient layer.
- **<forcedrelease>:** If >0, it overrides the layer's normal envelope release with this value (in seconds). This is crucial for avoiding audio "pops" during sudden stops.

```
LAYER // Ambient Pad
// If the Mod Wheel (CC 1) hits the very top (127), the pad dies.
OFFMCCT 1, 127, 127, 0.5
OUT = GAIN * OSCG("s1f0000")
```

Default -1,-1,-1.

This allows setting, modifying or deleting an OFF trigger on a MIDI CC.

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.

The MIDI channel used for the MIDI CC check is the default channel. Channel is not applicable to global MIDI CCs as usual.

Strategic Summary: Triggers & Groups

| Goal | Instruction | Logic |
|-------------------------|------------------|-----------------------------|
| Exclusive Voices | GROUP + OFFGROUP | "If A starts, B must stop." |

| Goal | Instruction | Logic |
|--------------------------|----------------------|--|
| Monophonic Layer | GROUP X + OFFGROUP X | "If I start again, kill my previous self." |
| Manual Cut-off | OFFMCCT | "If this knob turns, stop the sound." |
| Clean Transitions | <forcedrelease> | "Fade out over X seconds instead of stopping instantly." |

Developer's Note

When using OFFGROUP for percussion (like the Hi-Hat example), a forcedrelease of 0.02 to 0.05 is usually the "sweet spot." It's fast enough to sound like a choke, but slow enough to prevent the digital clicking caused by an instantaneous volume drop to zero.

Trigger Examples:

1. Organic Examples of ON Triggers

A. Velocity-Sensitive Articulation (Layering) Use the ONNOTEON instruction to split sounds across different touch intensities.

```
LAYER
// Only triggers if the key is struck with a velocity between 1 and 64
ONNOTEON 0, 127, 1, 64
OUT = GAIN * OSCG("slf0000", 200) // Soft Sample Slot
```

In this example, the layer only responds to gentle playing, allowing for a separate layer to handle harder strikes.

B. Legato-Only Transitions Use ONFIRST and ONLEGATO to differentiate between the start of a musical phrase and subsequent connected notes.

```
LAYER
// Triggers ONLY if a note is already being held (Legato playing)
ONLEGATO 0, 127
// This layer might contain a "slide" sample or a smooth-attack oscillator
OUT = GAIN * OSCG("slf0000")
```

This is typically paired with an ONFIRST layer that handles the initial "attack" sound of a phrase.

C. Round Robin (Natural Variation) Use ONRROBIN to cycle through different samples of the same drum or instrument to avoid the "machine gun effect."

```
LAYER
// Triggers on every even-numbered Note ON (0, 2, 4...)
```

```
ONRROBIN 0, 2
OUT = GAIN * OSCG("S1f0000", 300) // Sample variant A
```

```
LAYER
// Triggers on every odd-numbered Note ON (1, 3, 5...)
ONRROBIN 1, 2
OUT = GAIN * OSCG("S1f0000", 301) // Sample variant B
```

The engine maintains per-channel counters to ensure the round-robin sequence stays consistent across performances.

2. Organic Examples of OFF Triggers

A. Exclusive Groups (Hi-Hat Choking) Use the `GROUP` and `OFFGROUP` instructions to ensure that specific sounds are mutually exclusive.

```
LAYER
GROUP 10 // Assign "Open Hi-Hat" to Group 10
ONNOTEON 46, 46, 0, 127 // MIDI Key for Open Hat
OUT = GAIN * OSCG("S1f0000", 400)

LAYER
GROUP 11 // Assign "Closed Hi-Hat" to Group 11
// When this layer triggers, it will silence any active layer in Group 10
OFFGROUP 10, 0.05
ONNOTEON 42, 42, 0, 127 // MIDI Key for Closed Hat
OUT = GAIN * OSCG("S1f0000", 401)
```

The 0.05 parameter represents a forced release in seconds, allowing the "Open Hat" to be cut off abruptly with a 50ms fade to prevent clicks.

B. Controller-Driven Sound Termination Use `OFFMCCT` to silence a layer based on real-time MIDI CC or VST Variable data.

```
LAYER
// Force this layer to release if VST Knob #5 (Index 605) goes above 0.8
OFFMCCT 605, 0.8, 1.0, 0.1
OUT = GAIN * OSCG("y1f0000", 2, 0.5) // Continuous Sawtooth
```

This is useful for sound design where a specific modulation state (like a wide-open filter or a high-intensity LFO) should terminate the sound.

Crossfades:

XFADE

<type>;<mcc>;<min>;<max>;<power>

While triggers (`ONMCCT`) are "all or nothing" gates, crossfades allow you to smoothly morph between sounds. This is the difference between a light switch and a dimmer.

A crossfade monitors a MIDI CC or VST variable and scales the volume of a layer accordingly. If the resulting multiplier is **0**, the layer isn't even triggered, saving CPU.

1. Technical Syntax

```
XFADE <type>; <mcc>; <min>; <max>; <power>
```

- **<type>**: 0 = IN (starts silent, grows to full), 1 = OUT (starts full, fades to silent).
 - **<mcc>**: The controller to watch (0–1007).
 - **<min>; <max>**: The range of the controller where the fade occurs.
- 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>**: The "shape" of the curve:
 - **0 (Linear)**: A straight line.
 - **1 (Power)**: Square root curve. Perfect for keeping the volume consistent when crossfading two layers (avoids a "dip" in the middle).
 - **2 (Exponential)**: A logarithmic curve from -60 dB to 0 dB. Sounds most natural for volume swells.

2. Strategic Applications

A. The "Orchestral Swell" (Mod Wheel)

Orchestral libraries often use the Mod Wheel (CC 1) to crossfade between a quiet, intimate violin and a loud, aggressive one.

```
LAYER 1 // Soft Violin
XFADE 1, 1, 64, 127, 1 // Fades OUT as wheel goes up

LAYER 2 // Loud Violin
XFADE 0, 1, 0, 64, 1 // Fades IN as wheel goes up
```

B. Velocity Morphing

Instead of a hard "snap" between velocity layers, you can use XFADE on CC 130 (Note Velocity) to blend them.

```
LAYER 1 // Soft Piano
XFADE 1, 130, 40, 90, 1 // Fades out as you hit harder

LAYER 2 // Hard Piano
XFADE 0, 130, 40, 90, 1 // Fades in as you hit harder
```

C. The "Sweet Spot" (Band-Pass Control)

You can use multiple crossfades on the **same CC** to isolate a sound to a specific knob range.

```
LAYER // "Mid-Range" Synth
XFADE 0, 600, 0.3, 0.4, 0 // Fade IN at 0.3
XFADE 1, 600, 0.6, 0.7, 0 // Fade OUT at 0.6
// Result: Sound is only heard when VST Knob 0 is between 0.3 and 0.7
```

D. Standard Mod Wheel Crossfade (IN)

To make a layer gradually appear as the Modulation Wheel (MIDI CC 1) is moved from its lowest to its highest position, use a linear fade-in:

```
// Fade in Layer 1 as CC 1 moves from 0 to 127
LAYER
XFADE 0, 1, 0, 127, 0
OUT = GAIN * OSCG("s1f0000")
```

If the Mod Wheel is at 64, the layer's output signal will be multiplied by approximately 0.5.

E. Velocity-Based Layer Blending (OUT)

You can use the `XFADE` instruction to blend out a "soft" sample layer as you play harder. Since Note ON velocity is accessed via MIDI CC 130, you can define a power-curve fade-out:

```
// Fade out this layer as velocity moves from 64 to 127
LAYER
XFADE 1, 130, 64, 127, 1
OUT = GAIN * OSCG("S1f0000", 200) // Soft Sample Slot
```

In this scenario, if the velocity is 127, the crossfade evaluates to 0, and the layer will not be triggered.

F. Global Inheritance and Resetting

Definitions in the `COMMON` section are inherited by all layers, but they can be cleared for a specific layer if needed.

```
// COMMON SECTION
XFADE 0, 1, 0, 127, 0 // All layers now fade in with Mod Wheel

LAYER
// This layer inherits the Mod Wheel fade from COMMON

LAYER
XFADE -1 // Deletes the inherited crossfade for THIS layer only
// This layer will now play at full volume regardless of the Mod Wheel
```

3. Inheritance and Rules

- **Multiplication:** If a layer has five different crossfades, the final volume is the **product** of all of them. If any one of them is at 0, the layer is silent.
- **COMMON Inheritance:** If you put an `XFADE` in the `COMMON` section, every layer in the file will use it. This is great for a global "Master Volume" or "Expression" control.
- 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.
- **The Reset (-1):** If a layer shouldn't inherit the global crossfades, use `XFADE -1` at the top of that layer to wipe the slate clean.
- **Real-time vs. Trigger:** Unlike `ONMCCCT` (which only checks at the start), `XFADE` continues to work while the note is held. If you move the knob while a note is sustaining, the volume will change in real-time.

Developer's Note

Use `power 1` (Square Root) when you are crossfading two similar sounds. Because of the way decibels work, a linear crossfade (`power 0`) will sound like it gets quieter in the middle (at the 50/50 point). The `power 1` curve keeps the perceived energy levels steady.

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.

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 or MIDI CC messages, starting from an initial set, possibly augmented by live user generated notes and/or messages.

From now on, for short, here we refer as messages any NOTE ON/OFF couple, MIDI CC message, VST variable or Keyswitch change.

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 SEQUENCERI ACTIVATE instruction with an invalid MIDI CC number, e.g. SEQUENCERI "ACTIVATE -1", see below. By default it is deactivated.

The sequencer is able to transform in a programmable way a sequence of messages and issue them on the next stage of processing.

To do this it can process sequencer instructions in two stages: immediate mode and program mode.

In immediate mode the instructions are executed during the instrument file compiling and they determine the initial TAPE content.

In program mode the instructions are stored in the "Program". This is executed at run time, each time the TAPE cursor reaches the end of the TAPE (see below).

The command to issue immediate instructions is SEQUENCERI. The command to queue instructions in the "Program" is SEQUENCER.

Those command can be intermixed. This is useful because you can organize programs in external files, referencing them with the INCLUDE instruction.

The TAPE

The sequencer uses as its main structure an abstraction, called the TAPE.

The TAPE contains an array of messages, sorted by trigger time and two positions: the <cursor> and the <end>.

The <end> is the index of where the next produced message in immediate mode or by the "Program" would go, by default. It signals the end of the TAPE.

The <cursor> is the index of the message that will be played next, as soon as the trigger time allows it. In immediate mode is just another position in which insert messages: the cursor moving instructions work also in immediate mode.

Multiple messages can be issued at a time if they have the same trigger time.

If it is a NOTE, each one can trigger multiple layers, just like normal NOTE ON messages.

Messages that move the sequencer VST Var are not applied, but if the sequencer VST Var is linked to another MIDI CC, these moving are applied: it is advisable to not link the sequencer VST Var, but to use other means (e.g. the STATUS instruction).

The <cursor> can range from 0 to <end>.

After the program is compiled, the cursor is set to 0, before starting the normal VST processing.

As soon as it reaches <end>, there are no more messages to be issued: in this case the “Program” will be issued, that hopefully will generate new messages to play next and/or moves the cursor.

The tape is 524288 slots in the current implementation. If the TAPE becomes full, the instructions that ADD messages are ignored.

There is an instruction, called RESET, to reset the cursor and the random seed (see below). This can be called programmatically or can be performed automatically in various occasions:

- At a suspend/resume cycle.
- When the DAW song position change.
- At instrument file start.
- At sequencer activation with ACTIVATE.
- When the user loads or saves a TAPE file.
- When issuing CLEAR or CLEARF instructions.

When the “Program” generates new messages, for each instruction of the program that new messages will be put on a BAR boundary, since the start of the processing: the last message trigger time is rounded in excess to the next bar and used as the starting time of the new messages. There are commands to put messages in other places, but the starting point is always on BAR boundary. The offset to this origin can be fractional, depending on the instruction.

Due to possible randomization or loops, the “Program” will be executed up to 10 times or 2000 instructions (by default: this value can be changed), until the <cursor> is less than <end> (a program can generate new messages or move the <cursor>). The program is fully executed at least once.

After the program execution(s), all the message triggers (and releases) are shifted so that the message pointed at the current cursor is the next to be played.

When the sequencer is activated, by default the TAPE is empty.

If the VST is part of a DAW project and in a preceding session a TAPE file was loaded and the file name was stored in the DAW file, this TAPE file will be loaded and the cursor is left pointing past the last message loaded. See Tape file loading/saving below.

Otherwise the TAPE will be empty and both cursors will point at its start.

The TAPE can be filled, during instrument file compiling, by using SEQUENCER instructions. These instructions will be executed in order and can specify new messages to insert, TAPE files to load (LOADTAPE instruction) and almost any processing supported in the normal run time program mode.

When activating MERGE mode with an empty TAPE, the SNAPSHOT instruction is automatically executed (see below), to take a snapshot of the parameters when starting a recording.

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 by the user.

After the instrument file loading, the <tape_file> variable is initialized to an empty value.

The current value of <tape_file> can be seen hovering 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 tape 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. For this workflow can be useful the COUNTDOWN instructions, that performs a countdown before starting playing the tape.

Ways of resetting <tape_file> variable to empty:

- Reloading the instrument file. This also resets all the personalizations made to the VST variables.
- Shift and/or Ctrl + left click on the sequencer knob. This also empties the TAPE and resets the cursor and the seed of the random number generator., but you don’t lose the VST Var personalizations.
- CLEARF instruction in immediate mode. Has the same effect of the option above this.

If there are not SEQUENCER instructions and the <tape_file> is empty, 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 messages 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, with the interface or the LOADTAPE instruction in immediate mode.

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.

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 trigger times must be in ascending order and the first slot MUST trigger at 0.0 seconds, as the LoadTape() routine is optimized for speed assuming this.

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.

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

1. Trigger time in seconds from the start of the sequence (0.0 for the first note/message).
2. Release time in seconds from the start.
3. Note pitch in semitones (e.g., 69.0 for A3) or message MIDI CC number:
 - 0-127: actual note.
 - 1000-1327: MIDI CC message 0-327 (standard and extended MIDI CC, including temperament and Aftertouch).
 - 1600-1727: VST variables.
 - 2000-2007: Keyswitches 1-8.
4. Velocity or message value (0.0 to 127.0 for notes, unrestricted for messages).
5. Off velocity (0.0 to 127.0).
6. Channel number (0-15).

Sequencer states

The sequencer has five states, selectable by the associated VST VAR:

➤ 0 - REWIND.

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), but neither the TAPE, neither the "Program" are touched.

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 rewind.

➤ 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 and "Program" state is not modified, in particular the cursor is not moved.

➤ 2 - SEQ. ONLY.

In this state the sequencer is running: NOTES and messages are generated from the TAPE.

As soon as the <cursor> reaches <end>, the "Program" is executed.

Input NOTES are discarded for ALL channels. MIDI messages, VST Vars etc. are still applied locally but all messages are not transmitted to the MIDI Output. Only NOTES and messages coming from the TAPE are transmitted on the MIDI Output.

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 messages. 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 messages. 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.
Finally also the other messages are saved in the TAPE in the correct position, as soon as they arrive.
This means that the next time the program is issued, it will process also the user inputted messages.
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 REWIND, 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.

There exist an instruction to change the state at runtime, but the value stored in the DAW file takes precedence. See below.

Going from REWIND or PAUSE to the other states, the TAPE trigger times are moved forward to take into account the time passed in pause mode and so to trigger the messages at the correct time.

By default the notes merged into the TAPE are taken before any MIDI POST processing step (including temperament), since by default when they are taken back from the TAPE and played (and eventually put on the MIDI OUT), the post processing steps are executed. There are four instruction to change this behavior. E.g. you can put modified instructions on the TAPE (to save e.g. a randomized or quantized live performance) and play them RAW from the TAPE (to avoid reapplying again the MIDI POST steps), or the other combinations. See PLAYRAW, PLAYMOD, MERGERAW and MERGEMOD.

Main Sequencer instructions

Here follows the description of the main sequencer related instructions.
The sequencer language is explained after the SEQUENCER[I] instruction definition.

Random number generator

The random numbers generator is used during sequencer instruction processing. It is used for probabilistic execution or when a range of values is specified, to pick one at random.

The random number 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, you can change the seed offset. You can also have the option to change the offset to a pseudorandom value, but this means that your file will not be repeatable anymore.

This can be performed with this instruction:

SEED <offset>

The same positive value will give the same sequence.

There are about 4 billion of combinations.

<offset> can be -2, -1 or an unsigned 32 bit integer that will be added to the internal seed generator. Default = 0.

If it's -1, then after the file is loaded a new unsigned 32 bit integer is generated, depending on the current time. In this case each time the file is reloaded, the seed is changed, but remains constant during all the various processing: if you rewind the TAPE, you have the same random numbers.

If it's -2, then after the file is loaded and each time the sequencer is RESET or REWOUND, a new unsigned 32 bit integer is generated, depending on the current time. In this case each time the seed is changed.

SEQUENCER[I] "Instructions", <minch>, <maxch>

Here it is the most important function for the sequencer module.

SEQUENCER[I] executes immediately the **"Instructions"**, processing the TAPE messages limited to the indicated channels.

SEQUENCER queues the **"Instructions"** in the "Program", including the channel limits, for run time execution.

The channel limits allow to filter the processing to certain channels or to apply to one or all channels in the range the instructions, repeatedly. See the instruction description for the actual meaning.

All the "Instructions" share the same channels setting, so if other instructions have to have different channels setting, you must split the instructions in separate SEQUENCER[I] instructions.

The limits allow to process different channels with different instructions, e.g. a program for the bass line, another for the lead, etc.

The instructions are executed mostly in order, except if using IF, PROB, GOTO or BRANCH.

The GOTO or BRANCH targets are constrained in immediate mode to the instructions contained in the single SEQUENCER[I] instruction.

They are not constrained in "Program" mode.

General notes:

- Instructions are separated by ";". Spaces between it and the next token are ignored.
- Parameters are separated by " , "
- Token name is case insensitive.
- All spaces but the one after the token are deleted, except for some instructions (see below), so there is freedom of formatting the string for clarity.

- Comments inside the program are not supported, unrecognized token are discarded up until the “;” and an error in the log will be given. Use standard comments outside quotes.
- The line continuation character “/” works also in opened strings, so you can put the program in contiguous rows, but it is not advisable for clarity. You can just issue separate SEQUENCER[I] instructions.
- Most parameters are optional. The parser will recognize also “<TOKEN><SPACE>;”, meaning an instruction without parameters, that maintains the default values.
- Some instructions have no parameters: those can be specified with or without the space before the “;”.
- The parsing of the parameters is also 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.
- The <cursor> is the pointer to the next message to be played (the “Program” is executed when the TAPE is empty at this note).
- The <rcursor> is the trigger time of the message at the <cursor>, rounded to the next integer bar since the start of the sequencer.
- The <end> is the pointer past the last message in the TAPE. When the “Program” is invoked, <end> = <cursor> but when the processing proceeds, <end> is generally greater than <cursor>. <end> is always rounded up to the next BAR since TAPE start.
- The <origin> is the actual origin used in the instructions. The instructions listed as “<name>x” support three variants:
 - x = empty: in this case the <origin> is <rcursor>.
 - x = ‘0’: in this case <origin> is the start of the TAPE.
 - x = ‘E’ or ‘e’: in this case <origin> is <end>.
 So in any case the <origin> is rounded to the next bar.
- The message 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.
- All the non-integer values can be expressed as fractions, e.g. 3/4 or 3\4. Numerator and denominator can also be float. Where it is specified that the number must be an integer, only the numerator is picked. No error is given.
- The bar ranges are opened on the superior edge. Generally can have also negative values from the origin, because the final values are still capped between 0 and <end> in some cases (See the instruction help for details).
- A single position (<pos>) can be out of 0-<end> range. This value is typically used in insertion instructions. The TAPE will be extended and if the insertion is at the head, before the 0 position.
- The VST monitors BPM and the time partiture values: as soon as detects a variation, all the TAPE times are rescaled around the current cursor to have the next notes with the right trigger and 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.
- The <minch> and <maxch> specify on which channels operate the instructions. Most instruction will work only on the channels specified, moving/copying/modifying only the messages of the selected channels or inserting new messages for all the channels specified. Some instructions operate only on the <minch>.

- Copy, delete and general modifying messages work on both notes and MIDI messages. Most instructions are specialized. The description will specify the behavior.
- Every instruction that moves the <cursor> is capped to 0-<end> and <rcursor> is updated to the rounded up position.
- Every instruction that adds messages, increments <end> that is then rounded up.
- Every instruction that adds or deletes messages does not move <cursor>, but only <end>. It is advised to reposition the cursor, because it will point to a different message.
- Except the MOVE instruction and if they add messages before the start of the TAPE: the shift mentioned above will move the cursor in terms of next message to be played.
- Every instruction except the immediate mode only are sensible to PROB, as a whole or element for element. It is specified in the instruction description.

Here is a list of the recognized instructions:

- SEQUENCER "ACTIVATE <VSTVAR>,<max instructions>;",<minch>,<maxch>
 - Only in immediate mode. It is shown standalone to highlight the SEQUENCER prefix, but it can be issued before and after other instructions, provided that is in immediate mode.
 - Activate or deactivate the SEQUENCER and set the VST Var number for the status change.
 - Both ranges 0-127 and 600-727 are accepted and specify a VST Var.
 - If <VSTVAR> is below zero or out of the ranges above, the sequencer is disabled.
 - <max instructions> specifies the maximum number of instructions to be processed at each program invocation before giving up. Capped between 100 and 100000. Default 2000. The maximum number of consecutive program invocations is always 10.
 - <minch> and <maxch> specify the channel interval enabled in MERGE mode. It is a subset of the channels enabled with MIDICH instruction. Messages outside these channels are not merged into the TAPE. VST Vars, temperament and Keyswitches messages do not have a channel and so are ALWAYS merged.
 - This command deletes the TAPE, the "Program" and resets the cursors and the random seed.
 - If <tape_file> is not empty and the sequencer is activated, the file is loaded and the cursor is left after the last message loaded. If you want add messages along those and not after those in immediate mode, use <name>0 variants or move the cursor.
- SEQUENCER "COUNTDOWN <seconds>;"
 - Only in immediate mode. It is shown standalone to highlight the SEQUENCER prefix, but it can be issued before and after other instructions, provided that is in immediate mode.
 - This instruction sets the countdown internal parameter.
 - Default 0. Capped between 0 and 60. Channels ignored.
 - This countdown is employed when the VST Var of the sequencer is moved from REWIND or PAUSE to other states.
 - The TAPE playing and the note merging is suspended until the countdown expires.
 - During the countdown are played short C3(60) notes every second and a long A3(69) at the last second. They are obviously not merged into the TAPE, but are output on the MIDI Output (if enabled) for the benefit of downstream devices.

- If countdown is less than 2 seconds, no note are played, but the countdown time is honored.
- SEQUENCER "PLAYRAW;"
 - Only in immediate mode. It is shown standalone to highlight the SEQUENCER prefix, but it can be issued before and after other instructions, provided that is in immediate mode.
 - It sets the mode in which the notes taken from the TAPE are not modified by the MIDI POST steps and are put on the MIDI Output and played RAW.
- SEQUENCER "PLAYMOD;"
 - Only in immediate mode. It is shown standalone to highlight the SEQUENCER prefix, but it can be issued before and after other instructions, provided that is in immediate mode.
 - It sets the mode in which the notes taken from the TAPE are modified by the MIDI POST steps before being put on the MIDI Output and played.
- SEQUENCER "MERGERAW;"
 - Only in immediate mode. It is shown standalone to highlight the SEQUENCER prefix, but it can be issued before and after other instructions, provided that is in immediate mode.
 - It sets the mode in which the notes MERGED into the TAPE are not modified by the MIDI POST steps and are put on the TAPE RAW.
- SEQUENCER "MERGEMOD;"
 - Only in immediate mode. It is shown standalone to highlight the SEQUENCER prefix, but it can be issued before and after other instructions, provided that is in immediate mode.
 - It sets the mode in which the notes MERGED into the TAPE are modified by the MIDI POST steps before being MERGED into the TAPE.
- PROB <x>;
 - Prefix for probabilistic execution.
 - Applies a probability <x> to the next instruction.
 - For most instructions this means that 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 MODyyyx this is the probability of modify of a single message. 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%.
- DEBUG Message;
 - Queue the message in the debug log, if enabled.
 - Maximum 255 characters.
 - Spaces are not deleted here. The message, from after the space after DEBUG to the ';' excluded or up until the end of the string, is put verbatim on the log. So the message cannot contain ';'.
 - Supports PROB and IF and immediate and "Program" mode.
 - Useful to see when, if and how often a branch is reached.
- LOADTAPEx File.TAP;
 - Load and insert the TAPE file messages in the TAPE, starting from <origin>. This means that the offset in the TAPE file is added to <origin> to detect where to insert the message.
 - The messages are inserted in the correct position: if the TAPE is not empty and the <origin> lies where there are other messages, the TAPE file is "layered" on top the old messages. This can be used to layer multiple tape files.

- Maximum 255 characters for the file name.
- Spaces are not deleted here. The string, from after the space after LOADTAPE_x to the ‘;’ excluded or up until the end of the string, is used verbatim as file name. So the file name cannot contain ‘;’.
- If the first character of the file name is an asterisk, a Load File dialog is shown to the user, with the string after the asterisk used as message or a default message if it is too short.
Note that all LOADTAPE_x *, immediate or not are analyzed at file compiling stage and the files are asked in sequence. If your DAW uses a single thread to initialize all the plugins, then the initialization can be slowed down, waiting for the user to select the TAP file. Experiment with your DAW.
- The <cursor> is left just after the last message loaded. If the file was loaded in a not empty TAPE, the last loaded message can also be not the last message in the TAPE and so the <cursor> can point to a valid message.
- Message are filtered by <minch> and <maxch>. VST Vars, keyswitches and temperament are not filtered per channel.
- Can be used in immediate and program mode.
- Supports PROB and IF before it. E.g. probabilistic or conditional load of a tape file.
- 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 <minch> channel, if applicable (e.g. VST Vars don’t have channel number).
 - <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.
- RESET; or REWIND;
 - Rewind the TAPE (<cursor> = 0) and resets the random seed.
 - Ignores the countdown if not in REWIND or PAUSE state.
- CLEAR;
 - Like RESET, but clears also the TAPE (but not the “Program”).
- CLEARF;
 - Like CLEAR, but clears also the <tape_file> variable.
- MOVE <minbars>,<maxbars>;
 - Let <x> be an integer random number between <minbars> and <maxbars>.
 - <minbars> and <maxbars> must be both integers and can be negative.
 - This instruction moves the <cursor> by <x>.
 - The <cursor> is capped between 0 and <end>.
 - The TAPE is not touched.
- SNAPSHOT;
 - This instruction saves in the TAPE, at the current cursor position (unrounded) the values of all MIDI CCs, VST Vars and Keyswitches different from zero.
 - This is useful to have the same parameters when replaying the TAPE file.
 - Items that must have a value of zero must be set manually (e.g. with ADDMCC instruction) in the TAPE.
 - This to avoid a storm of messages in the TAPE because all MIDI CC defaults to zero.

- This instructions ignores <minch> and <maxch>: all channels are saved.
- STATUS <value>;
 - This instruction modifies the Sequencer status.
 - It can be called both at runtime and immediate mode.
 - It supports PROB and IF.
 - It modifies the status as if the user has turned the knob: linked MIDI CC are moved, MIDI CC are sent on the MIDI Output and the countdown is honored if passing from REWIND or PAUSE to other statuses.
 - <value> can be: 0 for REWIND, 1 for PAUSE, 2 for SEQ.ONLY, 3 for LIVE, 4 for MERGE.
 - ACTIVATE resets the status, so in immediate mode STATUS must be put after ACTIVATE.
 - If reloading a DAW project, the VST Var value may be loaded from the DAW file, that is applied after file loading. So an immediate STATUS may be overwritten. But if the instrument file was not changed, the value saved in the DAW file should be the same of the STATUS instruction.
- EXIT <seconds>;
 - Stop the program after at least <seconds> seconds.
 - To further randomize the song duration, prefix with PROB and a low probability.
 - When the instruction is finally executed, if the timer is expired, the program is also stopped: no instructions after EXIT are executed.
 - The notes in the Tape will be played until the end, so the <cursor> is left to <end> after the execution of the instruction, but no new instructions are executed, even when the <cursor> reaches <end>.
 - The only ways to start processing again instructions are the reset with the VST Var knob or the reload of the instrument file or a RESET, CLEAR or CLEARF instruction.
- GOTO <min>,<max>;
 - Let <x> be an integer random number between <min> and <max>.
 - If <max> is missing, it is assumed <max> = <min> so it is a deterministic jump.
 - 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 to level 3 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 <max_instructions> (See ACTIVATE), and the total number of program executions is limited to 10, 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.
 - In immediate mode the target instruction is limited to be one of the fellow instructions in the SEQUENCER instruction. The numbering does not start from 0, so enable the debug to level 3 to be sure. In immediate mode is better to use BRANCH.
- BRANCH <min>,<max>;
 - Let <x> be an integer random number between <min> and <max>.
 - If <max> is missing, it is assumed <max> = <min> so it is a deterministic jump.

- 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 to level 3 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 <max_instructions> (See ACTIVATE), and the total number of program executions is limited to 10, 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.
- In immediate mode the target instruction is limited to be one of the fellow instructions in the SEQUENCER instruction.
- Example: execute 50% of the time <instr1> and 50% of the time <instr2>, then execute always <instr3> and following:
 SEQUENCER " ... PROB 50;BRANCH 3;<instr1>;BRANCH 1;<instr2>;<instr3>; ... ",...
- APPENDx <minbars>,<maxbars>,<minbars>,<maxbars>;
 - Let <x> be an integer random number between <minbars> and <maxbars>.
 - Let <y> be an integer random number between <minbars> and <maxbars>.
 - Copy the messages that trigger from the bar <origin>+<x> to the bar <origin>+<x>+<y> and append them after <end>.
 - Only messages coming from the channels <minch>-<maxch> are copied. Messages without channel (e.g. VST Vars) are always copied.
 - 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 <origin>+<x>+<y> is greater than <end> the copy can still proceed, provided that there is something to copy (<origin>+<x> is not greater than <end>).
 - <x> is capped so <origin>+<x> is between 0 and <end>.
 - <minbars> and <maxbars> must be at least 1.
 - <y> is not capped. It's at least 1 because of the previous requirement. The copy is always performed, so if <y> is huge and <origin>+<x> is less than <end> the tape can become full. In this case the copy is stopped.
- DELx <minbar>,<maxbar>;
 - Delete all the messages that triggers between <origin>+<minbar> and <origin>+<maxbar>.
 - <minbar> and <maxbar> can be fractional. If <maxbar> <= <minbar> then no messages will be deleted.
 - Only messages coming from the channels <minch>-<maxch> are deleted. Messages without channel (e.g. VST Vars) are always deleted.
 - Note: the following messages are NOT moved back to fill the hole. Use MODx and MODCCx to move manually the messages.
- DELODDx <minbar>,<maxbar>;
 - Same as DEL, but only messages in ODD BARs are deleted.
- DELEVENx <minbar>,<maxbar>;
 - Same as DEL, but only messages in EVEN BARs are deleted.
- DOUBLEx <minbar>,<maxbars>;

- 2x time pattern: for all messages that trigger between <origin>+<minbars> and <origin>+<maxbars>, stretch 2x the pattern. Move accordingly the following messages.
 - Only messages coming from the channels <minch>-<maxch> are moved. Messages without channel (e.g. VST Vars) are always moved.
- HALFx <minbar>,<maxbars>;
 - 0.5x time pattern: for all messages that trigger between <origin>+<minbars> and <origin>+<maxbars>, stretch 0.5x the pattern. Move accordingly the following messages.
 - Only messages coming from the channels <minch>-<maxch> are moved. Messages without channel (e.g. VST Vars) are always moved.
- ADDx <pos>,<pitch>,<duration>,<velocity>,<offvel>;
 - Add a note at the <origin>+<pos> bar.
 - <pos> can be any value: the TAPE will be enlarged if the trigger is out of bounds.
 - All the parameters are optional.
 - If missing: <pos> = 0, <pitch> = 69, <duration> = 1 bar, <velocity> = 100, <offvel> = <velocity>.
 - To add chords or arpeggios, use multiple ADDx instructions.
 - To have random parameters, put suitable modify instructions afterwards.
 - Multiple messages for each enabled channels are created.
- ADDMCCx <pos>,<mcc>,<value>;
 - Add a message at the <origin>+<pos> bar.
 - <pos> can be any value: the TAPE will be enlarged if the trigger is out of bounds.
 - To have random parameters, put suitable modify instructions afterwards.
 - Multiple messages for each enabled channels are created. If the <mcc> is of a channel agnostic message, only one message will be created.
- SETx <minbar>,<maxbar>,<pitch>,<duration>,<velocity>,<offvel>;
 - Modify notes: for all notes that trigger between <origin>+<minbars> and <origin>+<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 set to the values specified in the instruction.
 - Negative values can be used to skip the parameter, e.g. to modify only the duration, set the <pitch> to -1. Missing parameters are not modified.
 - <minbar> and <maxbar> can be fractional. If <maxbar> <= <minbar> then no notes will be set.
 - Only notes coming from the channels <minch>-<maxch> are set.
- SETMCCx <minbar>,<maxbar>,<mccmin>,<mccmax>,<value>;
 - Modify messages: for all messages that trigger between <origin>+<minbars> and <origin>+<maxbars> and have MIDI CC number in the interval, 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 value is set to the value specified in the instruction.
 - <minbar> and <maxbar> can be fractional. If <maxbar> <= <minbar> then no messages will be set.
 - Only messages coming from the channels <minch>-<maxch> are set. Messages without channel (e.g. VST Vars) are always set.
- MODx <minbars>,<maxbars>,<dtrigmin>,<dtrigmax>,<ddurmin>,<ddurmax>,<dpitchmin>,<dpitchmax>,<dvelmin>,<dvelmax>,<doffvelmin>,<doffvelmax>;
 - Modify notes: for all notes that trigger between <origin>+<minbars> and <origin>+<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. If <maxbar> <= <minbar> then no notes will be modified.
- Only notes coming from the channels <minch>-<maxch> are modified.
- 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 <end> is rounded up to the next bar.
- Analogously if the first note is moved before the TAPE start, the TAPE is enlarged.
- Not quantized. Use QUANTIZE_x to quantize after the modify.
- Missing parameters are set to 0.
- MODMCC_x <minbars>, <maxbars>, <mccmin>, <mccmax>, <dvalmin>, <dvalmax>;
 - Modify messages: for all messages that trigger between <origin>+<minbars> and <origin>+<maxbars> and have MIDI CC number in the interval, 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 value is modified by a random number between <dvalmin> and <dvalmax> (can be negative).
 - <minbar> and <maxbar> can be fractional. If <maxbar> <= <minbar> then no messages will be modified.
 - Only messages coming from the channels <minch>-<maxch> are modified. Messages without channel (e.g. VST Vars) are always modified.
 - Note: for each message is generated a different random number.
 - Not quantized. Use QUANTIZEMCC_x to quantize after the modify.
- QUANTIZE_x <minbar>, <maxbar>, <qtrigger>, <qduration>, <qpitch>, <qstrength>;
 - Quantize notes: for all notes that trigger between <origin>+<minbars> and <origin>+<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 quantized as specified below.
 - <minbar> and <maxbar> can be fractional. If <maxbar> <= <minbar> then no notes will be modified.
 - Only notes coming from the channels <minch>-<maxch> are modified.
 - 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.
 - <qstrength> is an optional parameter, defaulting to 100 (%), setting the percentage of quantization. A value below 100% means a partial quantization, e.g. the final value is a weighted mean between the original value and the quantized value. The value is capped between 0 and 100.
- QUANTIZEMCC_x <minbar>, <maxbar>, <mccmin>, <mccmax>, <qvalue>, <qstrength>;

- Quantize messages: for all messages that trigger between <origin>+<minbars> and <origin>+<maxbars> and have MIDI CC number in the interval, 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 value is quantized as specified below.
- <minbar> and <maxbar> can be fractional. If <maxbar> <= <minbar> then no notes will be modified.
- Only messages coming from the channels <minch>-<maxch> are modified. Messages without channel (e.g. VST Vars) are always modified.
- If <qvalue> <=0 do not quantize.
- If <qvalue> = <n> then quantize to the nearest number divisible by <n>. Can be fractional to quantize with different interval.
- <qstrength> is an optional parameter, defaulting to 100 (%), setting the percentage of quantization. A value below 100% means a partial quantization, e.g. the final value is a weighted mean between the original value and the quantized value. The value is capped between 0 and 100.
- INTERPx <pos>,<n>,<glide>,<dur>;
 - Take the nearest note at <origin>+<pos> and the next one, on the channel <minch>.
 - If there are not at least two notes after <origin>+<pos> until the end of the tape, then do nothing (you must have two notes to interp).
 - If <n> < 3, do nothing.
 - Substitute the two notes with <n> smaller notes to fill the interval. The interval covers the trigger of the first note, until the release of the last note.
 - The channel on which the notes are created is <minch>.
 - 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...
- INTERPMCCx <minbars>,<maxbars>,<n>,<exp>,<mcc>,<v1>,<v2>;
 - Create <n> interpolated messages between <origin>+<minbars> and <origin>+<maxbars> on the MIDI CC <mcc>, channel <minch>.
 - If <n> < 3, do nothing.
 - The messages have a value interpolated among <v1> in the first point and <v2> in the second point.
 - If <exp> = 1 the interpolation is linear, otherwise is quadratic or square root.
 - Examples: if you want increase the volume at linear power, you must set <exp> =2. If you want to decrease the volume at linear power, you must set <exp> = .5.
 - Substitute the two notes with <n> smaller notes to fill the interval. The interval covers the trigger of the first note, until the release of the last note.
 - The channel on which the messages are created is <minch>.
- ROLLsx <minbar>,<maxbar>,<type>;
 - Modify velocity for all the notes between <origin>+<minbars> and <origin>+<maxbars>, of the channels <minch>-<maxch>.
 - <type> = 0 rolls 0->127, 1 rolls 127->0.
- C2Ax <minbars>,<maxbars>,<dir>;
 - Chord to Arpeggio.

- All notes that trigger between $\langle \text{origin} \rangle + \langle \text{minbars} \rangle$ and $\langle \text{origin} \rangle + \langle \text{maxbars} \rangle$ on the channel $\langle \text{minch} \rangle$ are spread in the interval depending on $\langle \text{dir} \rangle$ option:
 - $\langle \text{dir} \rangle = 0$ 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).
 - The duration of the notes is not modified. If it's a long chord, probably you may want to modify it with MODx or SETx.
- A2Cx $\langle \text{minbars} \rangle, \langle \text{maxbars} \rangle;$
 - Arpeggio to chord.
 - All notes that trigger between $\langle \text{origin} \rangle + \langle \text{minbars} \rangle$ and $\langle \text{origin} \rangle + \langle \text{maxbars} \rangle$ on the channel $\langle \text{minch} \rangle$ are put at $\langle \text{origin} \rangle + \langle \text{minbars} \rangle$.
 - Duplicate notes are deleted. The pitch must be almost exact (difference < 1 cent).
 - The notes are put at $\langle \text{origin} \rangle + \langle \text{minbars} \rangle$, which can be fractional. Use MODx or SETx to move or increase duration.
- REVx $\langle \text{minbar} \rangle, \langle \text{maxbars} \rangle, \langle \text{ref} \rangle;$
 - Reverse pitch pattern around a reference note.
 - For all notes that trigger between $\langle \text{origin} \rangle + \langle \text{minbars} \rangle$ and $\langle \text{origin} \rangle + \langle \text{maxbars} \rangle$, of the channels $\langle \text{minch} \rangle - \langle \text{maxch} \rangle$, modify the pitch as follows: $\langle \text{pitch} \rangle = \langle \text{ref} \rangle - (\langle \text{pitch} \rangle - \langle \text{ref} \rangle) = 2 * \langle \text{ref} \rangle - \langle \text{pitch} \rangle$.
 - Default $\langle \text{ref} \rangle = 69$.
 - The function is reversible: applied 2, 4 etc. times reverts to the original notes.
- INVx $\langle \text{minbar} \rangle, \langle \text{maxbars} \rangle;$
 - Inverse pattern.
 - For all notes that trigger between $\langle \text{origin} \rangle + \langle \text{minbars} \rangle$ and $\langle \text{origin} \rangle + \langle \text{maxbars} \rangle$ on the channel $\langle \text{minch} \rangle$, swap positions.
- RNDx $\langle \text{minbar} \rangle, \langle \text{maxbars} \rangle;$
 - Random pattern.
 - For all notes that trigger between $\langle \text{origin} \rangle + \langle \text{minbars} \rangle$ and $\langle \text{origin} \rangle + \langle \text{maxbars} \rangle$ on the channel $\langle \text{minch} \rangle$, randomly shuffle positions.
- SHIFTx $\langle \text{minbar} \rangle, \langle \text{maxbars} \rangle, \langle \text{dir} \rangle;$
 - Shift notes.
 - For all notes that trigger between $\langle \text{origin} \rangle + \langle \text{minbars} \rangle$ and $\langle \text{origin} \rangle + \langle \text{maxbars} \rangle$ on the channel $\langle \text{minch} \rangle$, shift according to $\langle \text{dir} \rangle$, first or last notes are duplicated.
 - $\langle \text{dir} \rangle > 0$ shift left 1 position only, < 0 , shift right 1 position only
 - Example for 4 notes.
 - $\langle \text{dir} \rangle = -1$, $\langle a \rangle \langle b \rangle \langle c \rangle \langle d \rangle \implies \langle a \rangle \langle a \rangle \langle b \rangle \langle c \rangle$ ($\langle d \rangle$ is lost)
 - $\langle \text{dir} \rangle = 1$, $\langle a \rangle \langle b \rangle \langle c \rangle \langle d \rangle \implies \langle b \rangle \langle c \rangle \langle d \rangle \langle d \rangle$ ($\langle a \rangle$ is lost)
- ROTx $\langle \text{minbar} \rangle, \langle \text{maxbars} \rangle, \langle \text{dir} \rangle;$
 - Rotate notes.
 - For all notes that trigger between $\langle \text{origin} \rangle + \langle \text{minbars} \rangle$ and $\langle \text{origin} \rangle + \langle \text{maxbars} \rangle$ on the channel $\langle \text{minch} \rangle$, rotate according to $\langle \text{dir} \rangle$, first or last notes go to the other end.
 - $\langle \text{dir} \rangle > 0$ shift left 1 position only, < 0 , shift right 1 position only
 - Example for 4 notes.
 - $\langle \text{dir} \rangle = -1$, $\langle a \rangle \langle b \rangle \langle c \rangle \langle d \rangle \implies \langle d \rangle \langle a \rangle \langle b \rangle \langle c \rangle$
 - $\langle \text{dir} \rangle = 1$, $\langle a \rangle \langle b \rangle \langle c \rangle \langle d \rangle \implies \langle b \rangle \langle c \rangle \langle d \rangle \langle a \rangle$

- 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 `<origin>+<x>` on the channel `<minch>`.
 - If there is not a suitable note after `<origin>+<x>` until the end of the tape, then do nothing.
 - Delete all notes between `<origin>+<x>` and `<origin>+<x>+<y>` on the channel `<minch>`.
 - If `<n> = 1`, create a note equal to the selected one, but stretched to duration of `<y>` bars.
 - If `<n> > 1`, fill the interval `<origin>+<x> - <origin>+<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...
 - The new notes are created on the channel `<minch>`.

Example 1: Activation and Environment Setup Before a sequencer can function, it must be activated and linked to a VST Variable (knob) that controls its state (Rewind, Pause, Seq. Only, Live, or Merge).

```
// --- COMMON SECTION ---
// Activate sequencer linked to VST Var #0 (Index 600),
// max 10000 instructions per cycle, acting on all channels.
SEQUENCER "ACTIVATE 600, 10000", 0, 15

// Set a 5-second countdown before the tape starts playing/merging
SEQUENCER "COUNTDOWN 5"
```

The `ACTIVATE` command resets the TAPE and random seed during compilation.

Example 2: Initializing a TAPE with a Melodic Pattern You can use `SEQUENCER` to hard-code an initial melody into the instrument file so it is ready to play as soon as the plugin loads.

```
// Move cursor to start, then add a C major seventh arpeggio (1/32nd notes)
SEQUENCER "MOVE 0,0; ADD 0,60,1/32; ADD 1/4,64,1/32; ADD 2/4,67,1/32; ADD
3/4,71,1/32;", 0, 0
```

In this example, notes are added at 0, 1/4, 2/4, and 3/4 bar positions on channel 0.

Example 3: Creating a Looping Runtime Program To make a sequence repeat indefinitely, use the `SEQUENCER` instruction with the `RESET` command. This ensures that when the last note finishes, the cursor returns to the start.

```
// Program mode: When the tape ends, rewind to the beginning
SEQUENCER "RESET;"
```

Example 4: Probabilistic Variation and Branching The sequencer supports complex logic, including probability-based execution and conditional jumps, allowing for generative music patterns.

```
// 50% chance to execute instruction 1, else execute instruction 2
SEQUENCER "PROB 50; BRANCH 3; <instr1>; BRANCH 1; <instr2>; <instr3>;", 0, 15
```

The `BRANCH` command moves the instruction pointer relative to the current command.

Example 5: Dynamic TAPE Modification You can program the sequencer to periodically modify its own content, such as duplicating a bar or humanizing existing notes.

```
// Program mode: 20% chance to copy the last bar and append it to the end  
SEQUENCER "PROB 20; APPEND 0,1,1,1;"
```

```
// Randomly shift the pitch and velocity of notes in the first 4 bars  
SEQUENCER "MOD 0, 4, 0, 0, 0, 0, -2, 2, -10, 10;"
```


Sample declaration and prefiltering Instructions

Sample Slots

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 <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>;
<crossfadefbw>

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 synthesized 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 π .

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 synthesized 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. The only limitation is that the detuning is fixed (i.e. constant). To have variable detuning you have to use multiple oscillators and sum it (see example files).

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 directly 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.

For unsupported file types FFMPEG is used if available (see above).

<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 Appendix A.

1. Example: Importing a Sampled Waveform (Audio File)

This is the most common use case, where an external uncompressed audio file (such as a .WAV or .AIFF) is assigned to a slot.

Code Snippet:

```
// Assign a piano sample to slot 200
// File: "samples\Piano_C3.wav"
// Normalization: 4 (RMS separately for L/R)
// Center Note: 60 (Middle C)
SAMPLE 200; "samples\Piano_C3.wav"; 4; 60
```

- **Filename:** The path is relative to the instrument file's location.
- **Normalization (4):** The engine scales the audio to an RMS value of -6dB separately for each channel to ensure consistent volume across different samples.
- **Center Note (60):** This tells the engine that the pitch in the recording corresponds to MIDI note 60 (C3), allowing it to transpose the sample correctly across the keyboard.

2. Example: Defining a Synthesized Waveform (Harmonics)

Instead of a file, you can define a waveform mathematically by specifying up to 32 harmonics. This is ideal for creating "ever-running" synth sounds or detuned unison effects.

Code Snippet:

```
// Define a rich synth bass in slot 100
// Harmonic 1: Sine (0), Amplitude 1.0, Freq 1.0, Phase 0
// Harmonic 2: Square (1), Amplitude 0.5, Freq 2.0, Phase 1.57 (PI/2)
SAMPLE 100; 0; 1; 0.5; 2.0; 1.57
```

- **Numeric Codes:** 0 represents a sine wave, and 1 represents a square wave.
- **Harmonic Parameters:** For each additional harmonic, you must provide its type, relative amplitude, relative frequency, and phase offset in radians.
- **Mono Output:** Note that all synthesized synth data is processed as mono, though the OSCG function can still produce a stereo image during playback through detuning.

3. Example: Advanced Looping and Separate Release

This example demonstrates how to configure a sample for infinite sustain with a dedicated "release" portion found at the end of the file.

Code Snippet:

```
// Slot 300: Violin sample
// Normalization: 0 (None), Center Note: 69 (A3)
// Loops: 0 (Infinite), Direction: 0 (Forward)
// Start: 0, End: 0 (Auto-detect file end)
// Release Start: -10000 (Use last 10,000 samples for release)
// Loop Start: 5000, Loop End: 15000, Crossfade: 50
SAMPLE 300; "Violin.wav"; 0; 69; 0; 0; 0; 0; -10000; 5000; 15000; 50
```

- **Infinite Loops (0):** The sample will repeat the loop section as long as the key is held.
- **Loop Boundaries (5000, 15000):** The engine will cycle between these two sample positions.
- **Crossfade (50):** To prevent audible clicks, the engine performs a 50-sample fade between the end and the start of the loop.
- **Separate Release (-10000):** When the key is released, the engine will crossfade the sustain loop with the audio found in the final 10,000 samples of the file, simulating a natural instrumental decay.

4. Example: Administrative Slot Deletion

You can use a simplified syntax to explicitly free up memory or ensure a layer is silenced by deleting a previously declared slot.

Code Snippet:

```
// Delete the sample data currently stored in slot 200
SAMPLE 200
```

- In "Play mode," if an oscillator references an empty slot, it will produce no sound, and the layer may not be triggered at all.

**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, RENDER uses the current resample quality.

For synth slots, RENDER will oversample the synth data by the last parameter of the QUALITY instruction, capped at 257x.

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, 64 // 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, 64 // 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>,[<resonance>]

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>, [<resonance>],
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 resonance is automatically calculated based on the order of the filter, to have maximum flatness.

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.

Examples:

1. Fixed Filter Prefixes: FILTER and FILTERSEMI

The FILTER and FILTERSEMI instructions allow you to apply standard filter types to a sample slot.
 The primary difference is that FILTER uses frequency in Hertz, while FILTERSEMI uses semitones where 69.0 represents 440 Hz.

Syntax and Technical Logic:

- **Syntax:** FILTER <type>, <frequency_hz>, <resonance> SAMPLE/SAMPLES/RENDER

- **Filter Types:** These use the same coding as the real-time FILT function, such as 0 for 12dB/oct LP or -1 for 6dB/oct LP.
- **Scope:** When applied as a prefix to the SAMPLE instruction, the filtering only affects sampled data and is ignored for synth data.

Example: Low-Pass Filtering a Kick Drum

```
// Apply a 12dB/oct low-pass filter at 1000Hz with 0.5 resonance
FILTER 0, 1000, 0.5 SAMPLE 200; "kick.wav"; 0; 60
```

In this scenario, the kick drum sample is filtered once when the instrument loads, so the oscillator does not need to calculate the filter for every sample during playback.

2. Equalization Prefixes: EQ3DB and EQ1DB

The EQ3DB (3-band) and EQ1DB (1-band) prefixes enable precise spectral shaping. Unlike spatial prefixes, multiple instances of these can be stacked to create complex equalization curves.

Technical Capabilities:

- **Stacking:** You can combine multiple EQ1DB prefixes to create a multi-band equalizer (e.g., five consecutive prefixes for a 5-band EQ).
- **EQ Types:** The <type> parameter specifies the filter order, where 1 is a 6dB/oct pole and 2 is a 12dB/oct two-pole filter.

Example: High-Shelf and Mid-Scoop Stacking

```
// Boost frequencies above 5000Hz and cut the 400-800Hz range
EQ1DB 1, 5000, 20000, 6, 0.7, 0.7 EQ1DB 1, 400, 800, -3, 0.7, 0.7 SAMPLE 1;
"pad.wav"; 0; 69
```

This example processes the "pad.wav" file through two distinct equalizer stages before it is normalized and stored in memory.

3. Spatial Processing Prefix: WIDEPAN

The WIDEPAN instruction applies stereo widening and panning to the final waveform.

Operational Rules:

- **Single Instance:** Unlike the filter and EQ prefixes, WIDEPAN is limited to a single instance per sample declaration.
- **Channel Conversion:** If the original sample is mono and a non-zero pan is applied, the engine automatically converts the sample to stereo.
- **Mono Optimization:** If both widening and pan are set to 0, the final sample is forced to mono, which allows the use of faster mono oscillator variants.

Example: Panned and Widened Piano

```
// Widen the stereo image by 1.5 and pan it 50% to the left
WIDEPAN 1.5, -50 SAMPLE 300; "piano.wav"; 0; 60
```

The resulting data in slot 300 will incorporate these spatial properties permanently in memory.

4. Combining Multiple Prefixes

All prefiltering instructions can be combined into a single declaration line to perform comprehensive preprocessing. The engine processes these prefixes in the order they are written, typically culminating in normalization.

Comprehensive Example: Processed Vocal Sample

```
// Apply a high-pass filter, a 3-band EQ, and pan the result
FILTER -2, 200, 0 EQ3DB 2, 500, 3000, 0, -2, 4 WIDEPAN 0, 25 SAMPLE 400;
"vox.wav"; 4; 72
```

1. The sample is high-passed at 200Hz.
2. It then passes through a 3-band EQ with a 2dB dip in the mids and a 4dB boost in the highs.
3. The final audio is panned 25% to the right and converted to stereo if necessary.
4. Finally, the audio is RMS normalized (mode 4) before being stored in slot 400.

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.

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.

Defaults Instructions

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. It is important to note that the default modulations, formulas, envelopes and LFOs apply only to the simplified oscillators functions. You have the option of using just the less simplified oscillator functions, that have more parameters and are thus slightly slower, if you need a custom formula for some parameter.

- **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 stretched instruments, alternative tunings or 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`.

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, `FREQ_LFO` 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.

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.

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. The existence of local layer overriding of the default values, avoid the use of custom formulas, and thus slower OSCG variants, in most cases. You must resort to custom automation only in few cases, e.g. different automations in a single layer or a custom automation not attainable with the default formulas (e.g. a custom envelope curve or a custom gain or frequency formula or a custom LFO waveform, e.g. a sampled one).

Interaction and Merging

When both global and local modulation instructions for the same parameter and MIDI CC are present, the local setting takes precedence, see below.

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 (and generally are) 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.

Instructions:

BASEG

This 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

This 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 below 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 parameters 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 gives descending sawtooth, >=1 give ascending sawtooth and 0.5 triangular waveform. The other values give a sawtooth with unequal slopes.

3 = Smooth sawtooth with formula $(x/\pi) - \text{POWABS}(x/\pi, \text{<param>})$. <param> should be >>1, but no check is made. The more is <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.

Examples:

1. BASEG: Base Gain Initialization

The `BASEG` instruction defines the global or per-layer base multiplier for an oscillator's amplitude. By default, this value is set to 1.0, and it supports a range from 0.001 up to 1000.0.

- **Fixed Example:** `BASEG 0.5` sets the starting gain to half of the default amplitude for every note.
- **Dynamic Example:** `BASEG $600` samples the value of VST Variable #0 (Index 600) at trigger time to use as the base gain multiplier.
- **Layer Overriding:** When used in a `LAYER` section, a positive value overwrites the global setting for that specific layer, while a negative value instructs the layer to revert to the global common value.

2. VELTRACK: Velocity Tracking Sensitivity

The `VELTRACK` instruction sets the exponent used to calculate gain based on the incoming Note ON velocity,. The default value is 2.0, providing a natural exponential response to touch.

- **Standard Formula:** The engine calculates the default gain as $GAIN = BASEG * (ONVEL / 127)^{VELTRACK}$.
- **Implementation Example:** `VELTRACK 3.0` increases the "hardness" of the instrument, requiring a significantly stronger strike to reach maximum volume,.
- **Automation:** Using `VELTRACK $600` allows the velocity sensitivity curve to be adjusted at trigger time via a GUI knob or MIDI CC.

3. GAINMOD: Real-Time Decibel Modulation

The `GAINMOD` instruction allows for linear modulation of the gain, expressed in decibels, using specified MIDI Control Change (CC) messages,.

- **Standard Syntax:** `GAINMOD <MCC>, <MOD>` links a controller to a gain shift.
- **Practical Example:** `GAINMOD 1, -20` ensures that as the Modulation Wheel (CC 1) moves from 0 to 127, the gain is attenuated by up to 20 dB.
- **Stacking Rules:** `GAINMOD` declarations in the `COMMON` section are inherited by all layers, but local layer declarations for the same MIDI CC will take precedence.
- **Complex Modulators:** The sensitivity parameter can be defined as `$<MCC>`, allowing one MIDI CC to scale the modulation depth of another MIDI CC in real-time.

4. GAINENV: Multiplicative Gain Envelope

The `GAINENV` instruction applies a sample-accurate multiplicative envelope to the default gain. It is highly efficient for standard ADSR-style amplitude shaping.

- **Technical Syntax:** `GAINENV <amount>, <delay>, <atk>, <hold>, <decay>, <sustain>, <release>.`

- **Organic Example:** `GAINENV 1, 0, .1, 0, 10, 0, .5` creates a piano-like response with a 0.1s attack, a long 10s decay to silence (0 sustain), and a 0.5s release phase.
- **Parameter Automation:** Every parameter can be fixed (0.1) or linked to a controller using `$<MCC>` (sampled at trigger time) or `&<MCC>` (sampled continuously).
- **Curve Selection:** A positive time value (e.g., 0.5) results in an exponential curve, while a negative value (e.g., -0.5) forces a linear transition,.

5. GAINLFO: Periodic Amplitude Modulation (Tremolo)

The `GAINLFO` instruction introduces a periodic Low-Frequency Oscillator to the gain, with the modulation amount specified in decibels,.

- **Technical Syntax:** `GAINLFO <num>, <param>, <freq>, <amount>, <delay>, <atk>, <release>.`
- **Configuration Example:** `GAINLFO 0, 0, 5, 6, 0, 0, 0` creates a basic tremolo using a sine wave (type 0) at 5 Hz, fluctuating the volume by +/- 6 dB,.
- **Tempo Sync:** Setting the frequency to a negative value (e.g., -1) syncs the LFO speed to the DAW's tempo in bars (1 bar per cycle).
- **Waveform Options:** The `<num>` parameter supports various shapes: 0 for Sine, 1 for Square (with PWM), 2 for Triangle, 3 for Smooth Sawtooth, and 5 for White Noise,.
- **Evolution:** The `<atk>` and `<release>` parameters allow the tremolo intensity to fade in or out over a specified duration following the Note ON or Note OFF events,.

BASEF

This 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

This 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

This 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 below 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.

FREQ LFO <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 gives descending sawtooth, >=1 give ascending sawtooth and 0.5 triangular waveform. The other values give a sawtooth with unequal slopes.

3 = Smooth sawtooth with formula $(x/\pi) - \text{POWABS}(x/\pi, \text{<param>})$. <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.

Examples:

1. BASEF: Base Frequency

The `BASEF` instruction sets the global reference frequency, measured in Hertz, which anchor's the instrument's tuning.

- **Default State:** The factory default is 440 Hz.
- **Static Example:** `BASEF 432.0` (This implements the "Verdi tuning" standard across all notes).
- **Dynamic Example:** `BASEF $600` (This samples the position of VST Variable #0 at trigger time to set the reference pitch).
- **Constraints:** The engine supports values ranging from 0.1 Hz to 10,000 Hz.

2. KEYTRACK: Keyboard Sensitivity

The `KEYTRACK` instruction determines the pitch relationship between consecutive MIDI keys.

- **Default State:** The default is 1.0, representing standard 12-tone equal temperament.
- **Static Example:** `KEYTRACK 0.5` (This creates a "shrunk" keyboard where it takes two octaves on the physical keys to produce one musical octave).
- **Stretched Tuning Example:** `KEYTRACK 1.01` (Often used in piano instruments to implement stretched tuning across the keyboard range).
- **Constraints:** Values are accepted between 0 and 10.

3. KEYCENTER: Reference Pitch Anchor

The `KEYCENTER` instruction specifies which MIDI key corresponds exactly to the `BASEF` frequency.

- **Default State:** The default is 69.0, which is the MIDI note for A3.
- **Static Example:** `KEYCENTER 60.0` (This re-anchors the reference pitch to Middle C, making C3 the note that plays at 440 Hz if `BASEF` is at default).
- **Fine Detuning:** Fractional values like `KEYCENTER 69.12` are supported to perform precise global detuning in cents.

4. FREQMOD: Linear Frequency Modulation (Cents)

The `FREQMOD` instruction allows for the linear modulation of an oscillator's frequency using MIDI Control Change (CC) messages.

- **Standard Syntax:** `FREQMOD <MCC>, <MOD>`.
- **Standard Pitch Bend Example:** `FREQMOD 135, 200` (This configures the Pitch Bend wheel to have a range of +/- 200 cents, or two semitones).
- **Mod Wheel Vibrato Setup:** `FREQMOD 1, 100` (This causes the Modulation Wheel to shift the frequency by up to 100 cents as it is moved from 0 to 127).
- **Variable Sensitivity:** Using `FREQMOD 1, $605` allows VST Variable #5 (Index 605) to continuously scale the depth of the modulation applied by the Mod Wheel.

5. FREQENV: Pitch Envelopes

The `FREQENV` instruction applies a multiplicative envelope that shifts the pitch over the duration of a note.

- **Technical Syntax:** `FREQENV <amount>, <delay>, <atk>, <hold>, <decay>, <sustain>, <release>`.

- **"Blip" Attack Example:** `FREQENV 1200, 0, 0.01, 0, 0.1, 0, 0.1` (Creates a 1200-cent/one-octave pitch spike that rapidly decays to the root note, useful for synthesized brass or percussion).
- **Parameter Linking:** `FREQENV &600, 0, .1, 0, .5, 0, .1` (Uses the continuous value of VST Variable #0 to determine the maximum pitch shift amount).

6. FREQLFO: Frequency LFO (Vibrato)

The `FREQLFO` instruction introduces periodic frequency fluctuations, primarily for creating vibrato effects.

- **Technical Syntax:** `FREQLFO <num>, <param>, <freq>, <amount>, <delay>, <atk>, <release>.`
- **Standard Vibrato Example:** `FREQLFO 0, 0, 5, 20, 0.5, 0.2, 0` (Creates a sine-wave vibrato at 5 Hz with a depth of +/- 20 cents, starting after a 0.5-second delay and fading in over 0.2 seconds).
- **Tempo Syncing:** `FREQLFO 0, 0, -1, 50, 0, 0, 0` (Setting the frequency to -1 syncs the LFO cycle to exactly one BAR of the DAW's tempo).
- **Waveform Modulation:** The `<num>` parameter selects shapes: 0 for Sine, 1 for Square (PWM), 2 for Triangle, 3 for Smooth Sawtooth, and 5 for White Noise.

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 further 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 below 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 gives descending sawtooth, >=1 give ascending sawtooth and 0.5 triangular waveform. The other values give a sawtooth with unequal slopes.

3 = Smooth sawtooth with formula $(x/\pi) - \text{POWABS}(x/\pi, \text{<param>})$. <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.

Examples:

1. FCMOD: Base Cutoff and MIDI Modulation

The FCMOD instruction establishes the starting frequency for the filter and allows for linear modulation in cents via MIDI Control Change (CC) messages.

Technical Syntax: FCMOD <BASE_FREQ> <MCC1>, <MOD1>, ..., <MCCN>, <MODN>

- **<BASE_FREQ> Options:**
 - **0 (or < -10):** The filter perfectly tracks the oscillator frequency (FREQ).
 - **1 to 10:** The filter tracks the keyboard but is offset by a multiplier (e.g., 2 means $2 * FREQ$).
 - **> 10:** The filter uses a fixed frequency in Hertz, independent of the note played.
- **<MCC>, <MOD>:** Links a MIDI CC to a shift in the cutoff frequency measured in cents (100 cents = 1 semitone).

Example: Fixed Cutoff with Mod Wheel Control

```
// Sets a fixed base cutoff of 1000Hz
// Mod Wheel (CC 1) can increase the cutoff by 2400 cents (2 octaves)
FCMOD 1000, 1, 2400
```

In this scenario, the filter does not move when different keys are played unless the Mod Wheel is moved.

Example: Classic Keyboard Tracking with Aftertouch

```
// Base frequency tracks the played note (FREQ)
// Channel Aftertouch (CC 133) adds 1200 cents (1 octave) of brightness
FCMOD 0, 133, 1200
```

This ensures the filter always stays relative to the pitch of the note, providing consistent timbre across the keyboard.

2. FCENV: Filter Cutoff Envelope

The FCENV instruction applies a multiplicative envelope to the DEFAULTFC value, allowing the filter to "sweep" or "pluck" over the duration of a note.

Technical Syntax: FCENV <amount>, <delay>, <atk>, <hold>, <decay>, <sustain>, <release>

- **<amount>:** The peak modulation depth in cents.
- **Time Parameters:** Positive values create exponential curves, while negative values force linear transitions.

Example: "Acid" Synth Filter Pluck

```
// Quickly sweeps the filter up by 3600 cents (3 octaves)
// 0.01s attack, 0.3s decay to a sustain level of 0 (closed filter)
FCENV 3600, 0, 0.01, 0, 0.3, 0, 0.2
```

This creates a sharp, percussive filter movement common in electronic bass sounds.

3. FCLFO: Filter Cutoff LFO

The FCLFO instruction introduces periodic fluctuations to the cutoff frequency, typically used to create "wah-wah" effects or rhythmic tonal movement.

Technical Syntax: FCLFO <num>, <param>, <freq>, <amount>, <delay>, <atk>, <release>

- **<num>:** Selects the waveform (0=Sine, 1=Square, 2=Triangle, etc.).
- **<freq>:** Frequency in Hz (positive) or BARS (negative).
- **<amount>:** Peak modulation depth in cents.

Example: Rhythmic "Wah" Effect

```
// Sine wave LFO (0) at a rate of 1/4 bar (-.25)
// Fluctuates the cutoff by +/- 1200 cents (1 octave)
FCLFO 0, 1, -.25, 1200, 0, 0, 0
```

Because the frequency is set to `-.25`, the filter sweep will stay perfectly in sync with the DAW's tempo.

4. Implementation: Using the Default Cutoff

To apply these modulations to a sound, you must use the `FILT0` function within a `LAYER`. This function is hard-wired to use the `DEFAULTFC` value generated by the instructions above.

Comprehensive Implementation Example:

```
// COMMON SECTION
// 1. Set base frequency to track keyboard
FCMOD 0
// 2. Add an envelope pluck
FCENV 2400, 0, 0.05, 0, 0.4, 0.2, 0.3
// 3. Add a subtle 5Hz vibrato/wah
FCLFO 0, 1, 5, 200, 0, 0, 0

LAYER
// Generate sound
O = OSCG("sg")
// Apply filter using the DEFAULTFC calculated above
// Type 0 is a 12dB/oct Low Pass filter, resonance is set to 0.7
OUT = FILT0(0, 0, 0.7)
```

In this example, the resulting audio will have keyboard tracking, an automated envelope sweep, and a constant LFO oscillation all affecting the single filter instance.

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.

Note that the frequency must not be low. This is just an FM modulation, with almost fixed modulating waveform.

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 gives descending sawtooth, >=1 give ascending sawtooth and 0.5 triangular waveform. The other values give a sawtooth with unequal slopes.

3 = Smooth sawtooth with formula $(x/\pi) - \text{POWABS}(x/\pi, \text{<param>})$. <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.

Example:

In digital synthesis, modulating the phase of a signal is the fundamental method for achieving true Frequency Modulation (FM). Using PHASELFO allows for "fixed" FM effects—such as periodic vibrato or timbral shifts—without the need for complex carrier-modulator chains in every layer.

1. Technical Logic and FM Relationship

While Crescendo supports dynamic FM via the OSCG phase parameter, the PHASELFO instruction is optimized for speed when a constant, periodic modulation is sufficient.

- **For Synth Oscillators:** The modulation is applied in **radians**.
- **For Sampled Oscillators:** The modulation is applied in **seconds**.
- **FM Result:** Periodic variation of the phase translates directly to frequency variation. Because the starting phase is linear, a PHASELFO effectively acts as a phase-based vibrato or FM modulator.

2. Technical Syntax

PHASELFO <num>, <param>, <freq>, <amount>, <delay>, <atk>, <release>

- **<num> (Waveform Type):** Selects the LFO shape: 0 (Sine), 1 (Square), 2 (Triangle), 3 (Smooth Sawtooth), or 5 (White Noise).
- **<param>:** Adjusts the shape (e.g., PWM for square waves or slope for triangles).
- **<freq>:** The speed of the modulation in Hz (positive) or rhythmic BARS (negative).
- **<amount>:** The peak dephasing intensity (radians for synth, seconds for samples).
- **<delay>, <atk>, <release>:** Timers that control when the FM effect begins and how it fades in or out relative to the note trigger.

3. Implementation Example: Periodic FM Vibrato

To implement a fixed FM effect where the phase oscillates at 6 Hz with a depth of π radians (causing a distinct pitch wobble), follow this structure:

Step 1: Configuration (COMMON Section)

```
// Sine LFO (0), 6 Hz Speed, 3.14 radians depth (~half cycle), no delay
PHASELFO 0, 1, 6, 3.14, 0, 0, 0
```

Step 2: Activation in Layer To apply this LFO, you must use an OSCG format string where the **5th character** is set to something other than '0' (typically 'L' for LFO).

```
LAYER
// Format string: 4th char 'L' enables PHASELFO
// s = Sine, 1 = fixed gain, f = default frequency, 0 = no autophase, L =
PHASELFO ON
OUT = OSCG("slfL00")
```

4. Advanced Use: Tempo-Synced FM

By setting the frequency to a negative value, you can sync the FM timbral shifts to the DAW's tempo.

Example Syntax:

```
// Sine LFO (0) oscillating once every BAR (-1)
// Modulating the phase of a sampled oscillator by 0.05 seconds
PHASELFO 0, 1, -1, 0.05, 0, 0, 0
```

```
LAYER
// 'S' = Sampled Slot #200, 'L' = PHASELFO active
OUT = OSCG("SlfL00", 200)
```

5. Operational Notes

- **Efficiency:** Using PHASELFO is computationally faster than manually calculating FM formulas because the engine uses optimized internal routines to apply the shift.
- **Mono vs. Stereo:** PHASELFO is a mono modulation source, but it is applied to the starting phase which can be processed stereophonically by the oscillator.

- **Inheritance:** Like other default instructions, a `PHASELFO` defined in the **COMMON** section is inherited by all layers unless a specific layer defines its own `PHASELFO` to overwrite the global setting.

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 **COMPLEXMOD**s specified in the **COMMON** section are added to the **COMPLEXMOD**s 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 of the modify formula 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>:

| Code | Target Parameter | Effect Type |
|------|--------------------------------|------------------------------|
| 0 | Default Frequency | Adds/subtracts MOD cents. |
| 1 | Default Gain | Adds/subtracts MOD decibels. |
| 2 | Default Cutoff Frequency | Adds/subtracts MOD cents. |
| 3 | Frequency Envelope | Multiplies depth by MOD. |
| 4 | Frequency LFO (Vibrato) | Multiplies depth by MOD. |
| 6 | Gain Envelope | Multiplies depth by MOD. |
| 7 | Gain LFO (Tremolo) | Multiplies depth by MOD. |
| 9 | Cutoff Envelope (Filter sweep) | Multiplies depth by MOD. |
| 10 | Cutoff LFO (Wah) | Multiplies depth by MOD. |
| 12 | Default Phase LFO | Multiplies depth 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.

Examples:

Example 1: Mod Wheel-Controlled Vibrato Depth If you have a `FREQ_LFO` defined for vibrato, you can use `COMPLEXMOD` to ensure it only becomes audible when the Modulation Wheel (CC 1) is moved.

```
// Multiplies the Frequency LFO depth (Dest 4) by Mod Wheel (CC 1)
// Amount 1.0 means full depth at Mod Wheel 127
COMPLEXMOD 4, 1, -1, 1.0
```

Because <MCC2> is -1, it is ignored (treated as 1). The result is Vibrato * MCC2(1) * 1.

Example 2: Dual-Controller "Wah" Intensity You can modulate the depth of a filter LFO (`FCLFO`) using both the Modulation Wheel (CC 1) and a VST knob (e.g., VST Var #0, Index 600) simultaneously.

```
// Modulates Cutoff LFO depth (Dest 10) by CC 1 AND VST Var 0
// The LFO depth is multiplied by the product of both controllers
COMPLEXMOD 10, 1, 600, 1.0
```

In this scenario, if either the Mod Wheel or the VST knob is at zero, the LFO modulation is silenced.

Example 3: Three-Way Scaling (Advanced) This example scales the Gain Envelope intensity based on the Expression pedal (CC 11), Channel Aftertouch (CC 133), and a GUI multiplier knob (Index 605).

```
// Multiplies Gain Envelope depth (Dest 6) by CC 11, CC 133, and VST Var #5
COMPLEXMOD 6, 11, 133, $605
```

This calculates $Env * MCC2(605) * MCC2(11) * MCC2(133)$.

Example 4: Deleting a Global Modulation If a modulation is defined in the COMMON section but you wish to disable it for a specific LAYER, use the "reset" values for <AMOUNT>.

```
// COMMON SECTION
COMPLEXMOD 1, 1, -1, 20 // Mod Wheel adds 20dB globally

LAYER
// Local override: setting amount to 0 for Dest 0-2 deletes the modulation
COMPLEXMOD 1, 1, -1, 0

LAYER
// For modulators (Dest 3-12), use an amount of 1 to "neutralize" the multiplier
COMPLEXMOD 10, 1, -1, 1
```

MIDI CCs and keywords list

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. $\text{Volume} = (\text{MCC}(7) + \text{MCC}(39)) / 128 / 128$. |
| KEY/KEY1 | 128 | Y | PER LAYER | Key pressed after remapping, derived rounding KEYF. 69 is A3. 0-127. |
| KEYF | 129 | Y | PER LAYER | Key pressed including MIDI post-processing, remapping and temperaments. 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 $\frac{3}{4}$. |
| DEN | 138 | N | GLOBAL | Time partiture denominator. E.g. 4 for $\frac{3}{4}$. |
| 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: $\text{GAIN} = \text{BASEG} * (\text{ONVEL} / 127) ^ \wedge \text{VELTRACK}$. 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: $\text{FREQ} = \text{SEMI}(\text{BASEF}, \text{KEYTRACK} * (\text{KEYF} - \text{KEYCENTER})) * \langle \text{gliding_factor} \rangle$. 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: $\text{GAIN0} = \text{BASEG} * (\text{ONVEL} / 127) ^ \wedge \text{VELTRACK}$. 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: $\text{FREQ0} = \text{SEMI}(\text{BASEF}, \text{KEYTRACK} * (\text{KEYF} - \text{KEYCENTER})) * \langle \text{gliding_factor} \rangle$. 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. |
| INZ | 156 | N | GLOBAL | Current value of INZ variable. INZ keyword is an alias. |
| NOISE | 157 | N | N/A | Random number between 0 and 1, changing at each invocation. Useful for random execution with EXECIF. |
| <NOT ASSIGNED> | 158-159 | 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. $\text{MCC2}(200 + \text{KEY1})$ 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. |
| N/A | 1000-1007 | N | GLOBAL | These values are used to access in a simpler way the Keyswitches values, especially for modifying them via assignments. Moreover these are the extended MIDI CC numbers used in MIDI output. |

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). Note: MCC2 and MCC3 are always mono and are not suitable for inputs, outputs and sends. MCC syntaxes can be used, because they are substituted in compilation stage with the exact real slot number of the corresponding keyword. MCC2 and MCC2, instead, are standard functions that use the internal functions to access MIDI CC, that are mono.

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.

For the VST variables exist an alternate function, named VAR, that accepts VST VAR numbers in the range 0-127 and 600-727. It is like MCC: a plain substitution at compile time.

Another way to access the MIDI CC or the VST Variables is with the MCCnnnn and VARnnn keywords.

All these mechanisms, MCC(nnnn), VAR(nnn), MCCnnnn, VARnnn, can be used also on the left of the = sign, to create assignment instructions that modify in real time the given MIDI CC.

Also MCC(nnnn) and MCCnnnn, with nnnn = 1000-1007 is the only way to modify the Keyswitches in real-time. The range 400-527 works only on the right of the equal sign.

MIDI CC Keywords

Here follows a detailed description of the keywords defined for some MIDI CCs and system variables.

Keywords that can be used in LAYERs and in POST step:

PROGRAM keyword is the current MIDI program selected. 0 - 127. It is an integer. Continuously variable.

BANK keyword is the current MIDI bank selected. 0 - 16383. It is an integer. Continuously variable.

SAMPLERATE or SAMPLEFREQ keyword is the current sample rate. Useful in custom filters.

BPM, NUM, DEN keywords are the current Beats per minute and time signature specification. Continuously variable. Sampled at each sample block.

WHEEL or PBEND keyword is the current pitch bend. [-1, +1[. Floating point with 14 bit precision. Continuously variable.

TEMPERAMENT 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.

INZ is the VST input signal (INPUT # 1). Stereo. See the sections above for details.

IN is the VST input signal (INPUT # 2). Stereo. See the sections above for details.

IN0 – IN99 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.

OUT and OUT0 – OUT99 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.

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 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. If you need only the layers sum, you can subtract INZ from OUT as a first step and then use the new OUT and INZ as you wish, for example as a sidechain.

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
```

VAR0 – VAR127 or VAR600 – VAR727

Used to access the VST VARs.

MCC0 – MCC1007

Used to access standard or extended MIDI CC, KEYSWITCH values, VST VARS, Poly aftertouch values.

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). See the SMOOTH instructions for details.

The default MIDI channel is used to assess the MIDI CC value.

See above table for details on number assignment.

Keywords that can be used ONLY in LAYERs:

GAIN is an extended MIDI CC that accesses the default gain of the fast oscillator functions.

It can be used in expressions if you want to calculate a more complex GAIN for use with the full oscillator function, based on the default gain.

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 accesses a possible gain of the fast oscillator functions.

It can be used in expressions if you want to calculate a more complex gain for use with the full oscillator function, based on the default gain devoided of the modulations, envelope and LFO.

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 accesses the default frequency of the fast oscillator functions.

It can be used in expressions if you want to calculate a more complex FREQ for use with the full oscillator function, based on the default frequency, so to support gliding.

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 accesses a possible frequency of the fast oscillator functions.

It can be used in expressions if you want to calculate a more complex frequency for use with the full oscillator function, based on the default frequency, devoided of the modulations, envelope and LFO, so to support gliding.

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 accesses 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 above).

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.

KEYF keyword is the actual current note to be played, after applying the full MIDI Post processing, keyboard remapping and corrected for temperament. 0 - 127. 69.0 is A3. Fractional part multiplied by 100 is the cents of detuning. Fixed at trigger time (excluding gliding).

The actual pitch played by the layer depends on the oscillator definitions. FREQ is based on this value.

This is the steady state value, because during gliding or portamento, this value is varied between two values.

Note that KEYF is recalculated from the current frequency (FREQ0) assuming BASEF = 440 Hz, KEYCENTER = 69.0 and KEYTRACK = 1, so if you plan to use this value to perform a custom mapping, take this into account.

KEY or KEYI keyword is the effective note number currently to be played in the layer. It is calculated rounding KEYF 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.

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.

The value is the same of ONVEL during the early stages of the note. In release stage it suddenly becomes the OFF velocity, if available. Otherwise it remains set to the ON velocity.

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.

NOISE keyword is a random number between 0 and 1, that varies each time is accessed.
Useful for EXECIF when every time the result must be different, like PROB of the SEQUENCER..

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.

Assignments and Expressions Engine

Having established the global parameters and structural foundations in the **COMMON** section, we now transition to the core functional power of the Crescendo engine: the **Assignment and Expression System**. These features serve as the primary building blocks for sound generation, MIDI manipulation, and audio processing across all stages of the pipeline.

1. The Assignment: The Core Unit of Action

In Crescendo, an assignment is a mathematical statement used to programmatically modify a variable, a MIDI CC, a VST parameter, or an audio output. This is the fundamental mechanism through which sound is produced and processed.

- **Syntax:** All assignments follow the standard programming format: `<variable> = <expression>`.
- **Sample-Accurate Processing:** By default, every assignment is evaluated **sample-by-sample**, ensuring that modulations and signal changes are perfectly smooth and devoid of digital artifacts like zipper noise.
- **Implicit Declaration:** Unlike many programming languages, Crescendo does not require explicit variable declarations. You define a new variable simply by using it on the left side of an assignment operator.
- **Target Flexibility:** You can assign values to user-defined variables, system keywords (such as `OUT` or `SENDS<i>`), or even MIDI CC numbers to modify internal storage in real-time.

2. Expressions: Defining Relationships

An expression is a combination of constants, variables, operators, keywords, and functions that resolves to a single value. They allow you to define complex, dynamic relationships between your control signals and the resulting audio.

- **The Ingredients of an Expression:**
 - **Constants:** Fixed 32-bit floating-point numbers (e.g., `440` or `0.5`).
 - **Keywords:** Reserved system identifiers that provide access to real-time data, such as `KEY` (the note being played), `ONVEL` (velocity), or `BPM` (tempo).
 - **Operators:** Standard mathematical symbols for addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (^).
 - **Functions:** Built-in processing routines, including oscillators (`OSCG`), envelope generators (`ENV`), and filters (`FILT`).

3. Recursive Complexity and Automation

The most powerful aspect of the Crescendo engine is its support for **unlimited complex formulas**.

- **Nested Automation:** Every automatable parameter can be driven by a formula, which can itself contain parameters driven by other formulas. This allows for an infinite depth of modulation where, for example, a filter's resonance could be modulated by an LFO, whose

frequency is in turn modulated by a complex mathematical relationship with the note's velocity.

- **CPU-Based Limits:** There are no hard-coded limits on the number of assignments you can include in a layer or the post-processing stage; the only practical constraint is the processing power of your host CPU.

4. Variable Scope and Data Management

Understanding where a variable "lives" is critical for managing signal flow.

- **Local Scope:** Variables defined within a **LAYER** or the **POST** section are local to that specific instance and cannot be accessed by other parts of the instrument.
- **Global Integration:** Variables declared in the **COMMON** section are global and are automatically inherited (replicated) by all layers and the post-processing section.
- **Optimization:** The engine includes an "intelligent compiler" that identifies "dead values"—instructions that do not contribute to the final audio output or a MIDI CC—and skips their execution to preserve CPU resources.

By mastering these concepts, you can transcend the limits of traditional fixed-function synthesizers, creating instruments where every sonic detail is governed by your own bespoke mathematical models.

Sound Generation and Evolution

- **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

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

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 **OSC** 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

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

- **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("s1v0A0", 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("s1v000", 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.

Expressions

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 a conditional execution construct, `IF...GOTO/EXIT`, which add further flexibility. You can make decisions based on specific conditions, leading to more dynamic and responsive instruments.
- **Assignments:** you can modify in real time any MIDI CC with a custom expression, evaluated sample for sample.
- 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.

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.

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 compatible, but more general than SoundFont 2.04 modulation, because it supports custom curves for distortions and not only the predefined ones. Moreover in Crescendo, MOD2, like all the other functions, can be used with any data, including oscillators, producing nonlinear ring modulations, while in the SoundFont file is used only to combine two MIDI CCs.

Filters and Equalizers

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

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

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 (`LUTs`), 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

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.

- **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.
- **CURVE Function:** This function creates custom lookup tables (LUTs) that define mappings between input values and output values. These LUTs can then be applied to various parameters.
- **Transcendental Functions:** 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`, and `ATAN` and Hyperbolic counterparts):** These functions provide standard trigonometric operations, enabling the creation of periodic signals and implementing various signal processing techniques.

Oscillator Functions

OSCG("format string", <parameters>...)

General oscillator function.

The `OSCG` function is the primary sound generation engine of Crescendo. It is a highly optimized, "all-in-one" function that handles waveform generation, gain/envelope shaping, frequency modulation, and integrated multi-mode filtering.

The Format String

The "format string" is a case-sensitive string of **0 to 8 characters**. Each character defines a specific component of the oscillator's signal chain. The characters over the 8th are ignored.

- **Default String:** "s1f00000"
- **Missing Characters:** If the string is shorter than 8 characters, the missing slots take their values from the default string.
- **Invalid Characters:** If an unrecognized character is used, the function reverts to the default character for that slot.
- **Parameter Order:** The parameters following the string must appear in the exact order of the components defined by the string (from character 1 to 8). See below for the details.

1st Character: Oscillator Type

Defines the waveform source and the initial phase logic.

| Code | Type | Parameters | Description |
|------|-------------|-----------------------------|---|
| s | Sinusoid | None | Standard stereo sine wave. |
| y | Synth | <num>, <param> | Single synth waveform. |
| S | Stereo Slot | slot | Plays a sample or a synth waveform from the specified slot number (Mono and sampled at trigger time). |
| L | Slot + Loop | slot, l_start, l_end, l_cnt | Sampled osc with real-time loop control, stereo and continuous (see <code>SAMPLE</code> instruction for details). <code>UNDEFINED</code> behavior if the slot is of synth type. |

| Code | Type | Parameters | Description |
|----------|-------------|--|---|
| V | Full Sample | <code>slot, s, e, l_s, l_e, l_c</code> | Complete control over start, end, and loop points. Stereo and continuous (see SAMPLE instruction for details). UNDEFINED behavior if the slot is of synth type. |
| O | Mono Sine | None | High-performance mono sine. |
| m | Mono Synth | <code><num>, <p></code> | High-performance mono synth waveform. |
| M | Mono Slot | <code>slot</code> | Plays a sampled slot in mono mode. |

‘O’, ‘m’, ‘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.

For ‘y’, ‘m’ types, here are the specifics:

- `<num> = 0.`
 - Sine wave distorted with power `<p>` like the function **POWABS**. With `<p>` near 0 the waveform approaches the square wave.
- `<num> = 1.`
 - Square wave with $\text{PWM}(\%) = \text{<p>} * 100$. If `<p> <= 0`, PWM is zero. If `<p> >= 1`, PWM is 100%.
- `<num> = 2.`
 - Triangular wave with slope width given by `<p>`: `<=0` gives descending sawtooth, `>=1` give ascending sawtooth and 0.5 triangular waveform. The other values give a sawtooth with unequal slopes.
- `<num> = 3.`
 - Smooth sawtooth with formula $(x/\text{PI}) - \text{POWABS}(x/\text{PI}, \text{<p>})$. `<p>` should be `>>1`, but no check is made. The more is `<p>`, the more the high frequency content.
- `<num> = 4.`
 - Triangular distorted with power `<p>` like the function **POWABS**. With `<p>` near 0 the waveform approaches the square wave.
- `<num> = 5.`
 - White noise. `<p>` ignored.

`<num>` is mono and sampled at trigger time, `<p>` is stereo and can be continuous variable.

NOTE ON POWER OVERSAMPLING: for synth data, the Power oversampling is applied. It allows you to set a low oversampling factor (1x is sufficient for sampled data if `ww` is high enough) and have a good quality on synth data. In power oversampling mode the actual oversampling factor is given by the last parameter of the **QUALITY** instruction, capped at 257x.

Performance measures show that a 10x oversampling (the default mode) costs only less than double CPU occupation than a synth oscillator at 1x, but this gives artifact free saw or square waves up to C7. You can further increase the quality if needed.

2nd Character: Gain (VCA) Formula

Defines how the amplitude is calculated.

| Code | Logic | Parameters |
|--------------|------------|--|
| 1 | Fixed Gain | None (Gain = 1.0) |
| C | Automation | Result = <C> (Stereo continuous) |
| g | Variable | None (Uses global <code>GAIN</code> variable) |
| k | Scaled | <k> (Result = $GAIN * k$) (Stereo continuous) |
| d | Decibels | <dB> (Result = $GAIN * 10^{(dB/20)}$) (Stereo continuous) |
| G/K/D | Same | Same as above, but uses <code>GAIN0</code> (See above). |

3rd Character: Frequency (Pitch) Formula

Defines the base frequency and tuning logic.

| Code | Logic | Parameters |
|----------|------------|---|
| A | Fixed | None (Fixed at 440Hz) |
| v | Automation | Result = <F> (Direct frequency in Hz) (Stereo continuous) |
| f | Variable | None (Uses global <code>FREQ</code> variable) |
| k | Scaled | <k> (Result = $FREQ * k$) (Stereo continuous) |

| Code | Logic | Parameters |
|-------|-------|--|
| c | Cents | $\langle \text{cents} \rangle$ (Result = $\text{FREQ} * 2^{(\text{cents}/1200)}$) (Stereo continuous) |
| F/K/C | Same | Same as above, but uses <code>FREQ0</code> (See above). |

Note: If the final frequency is negative, than means that are BARS and Crescendo automatically converts it to Hz. E.g.: -1 means 1 BAR. This is useful for synched oscillators, e.g. LFOs.

4th, 5th, & 6th Characters: Phase & FM Modulation

These characters control the starting phase (**AUTOPHASE**), the fixed LFO (**PHASELFO**), and the new internal **FM Modulation Engine**. All phase offsets are additive. A disabled term adds zero.

4th Character: Starting Phase (AUTOPHASE)

- **0: Disable AUTOPHASE.** The oscillator starts at a defined phase (usually zero, or specified by the 6th character). This is typically used for **LFOs** to ensure predictable modulation patterns or for **Sampled Oscillators**.
- **Any other character: Enable AUTOPHASE.** This emulates the behavior of continuously running oscillators in analog synthesizers. The plugin calculates a "random" phase offset based on the elapsed time since the engine started, ensuring each note trigger sounds organic and slightly different.

5th Character: PHASELFO

- **0: Disabled.**
- **Any other character: Enabled.** Adds the global **PHASELFO** automation to the phase.

Note: Use this for fixed-frequency FM (modulation that does not track the note pitch). Since only one PHASELFO is available per layer, use the 6th character options for note-tracking FM or creating multiple distinct modulations.

6th Character: Internal FM Engine & Manual Offset

This character defines the FM algorithm and determines how many additional parameters are consumed from the parameter list.

| Char | Mode | Params | Description |
|------|------|--------|----------------|
| 0 | Off | 0 | No modulation. |

| Char | Mode | Params | Description |
|----------|---------------|-------------------------|--|
| P | Manual | 1 (<ph>) | Adds a manual offset/automation. Required for Self-Modulation, External FM (modulating one OSC with another, even sampled), or custom phase starts for sampled oscillators. |
| s | sine | 1 (<k>) | Internal Sine FM. Amplitude fixed at 1.0 . $F = k * \text{FREQ}$. |
| S | SINE | 2 (<a>, <k>) | Internal Sine FM. Amplitude <a>, $F = k * \text{FREQ}$. |
| f | fm | 2 (<k>, <t>) | Type <t> FM. Amplitude fixed at 1.0 , $F = k * \text{FREQ}$. |
| F | FM | 3 (<a>, <k>, <t>) | Type <t> FM. Amplitude <a>, $F = k * \text{FREQ}$. |
| g | fm2par | 3 (<k>, <t2>, <p>) | Type <t2> FM with extra param <p>. Amplitude fixed at 1.0 , $F = k * \text{FREQ}$. |
| G | FM2par | 4 (<a>, <k>, <t2>, <p>) | Type <t2> FM with extra param <p>. Amplitude <a>, $F = k * \text{FREQ}$. |

Waveform codings:

1 parameter:

- 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 $(x/\text{PI}) - (x/\text{PI})^y$, with y set at 9, 19, 39 and 59, giving increasing high frequency content.

2 parameters:

- <t2> = 0.

- Sine wave distorted with power <p> like the function POWABS. With <p> near 0 the waveform approaches the square wave.
- <t2> = 1.
 - Square wave with $\text{PWM}(\%) = \text{<p>} * 100$. If <p> <= 0, PWM is zero. If <p> >= 1, PWM is 100%.
- <t2> = 2.
 - Triangular wave with slope width given by <p>: <=0 gives descending sawtooth, >=1 give ascending sawtooth and 0.5 triangular waveform. The other values give a sawtooth with unequal slopes.
- <t2> = 3.
 - Smooth sawtooth with formula $(x/\text{PI}) - \text{POWABS}(x/\text{PI}, \text{<p>})$. <p> should be >>1, but no check is made. The more is <p>, the more the high frequency content.
- <t2> = 4.
 - Triangular distorted with power <p> like the function POWABS. With <p> near 0 the waveform approaches the square wave.
- <t2> = 5.
 - White noise. <p> ignored.

<t2> is mono and sampled at trigger time, <p> is stereo and can be continuous variable.

Performance Tip: Lowercase characters (s, f, g) are more CPU-efficient as they bypass the amplitude parameter fetch and multiplication.

Usage Examples

Example 1: Manual FM / Sampled Start (Character 'P')

Used when you need to modulate the phase using an external source or need a specific starting point for a sampled oscillator.

```
OSCG ("xxxxxxP00", ..., 0.5)
```

- The phase is shifted by **0.5** radians. If this parameter is automated by another oscillator's output, it creates **External FM**.

Example 2: Advanced FM Modulation (Character 'F')

Creating a classic FM growl using a Triangle wave as a modulator.

```
OSCG ("xxxxxxF00", ..., 0.8, 2.0, 3
```

- <a> = **0.8**: Modulation intensity (Depth).
- <k> = **2.0**: The modulator runs at twice the fundamental frequency.
- <t> = **3**: Modulator waveform type (e.g., Triangle).

Example 3: Complex Global FM (Character 'G')

Using a variable-pulse or specialized waveform to modulate a carrier.

OSCG ("xxxxxxG00" ..., 0.6, 1.5, 1, 0.2

- **<a> = 0.6:** Modulation depth.
- **<k> = 1.5:** Non-integer ratio for inharmonic, metallic bell-like tones.
- **<t> = 1:** Advanced waveform type (e.g., Pulse Width Modulated wave).
- **<p> = 0.2:** Specific parameter for waveform <t> (e.g., 20% Pulse Width).

Technical Notes on Ratio <k>

There is no safety check on the frequency ratio <k>.

- **k > 0:** Standard FM.
- **k < 0:** Inverts the modulator phase. This is mathematically valid and useful for complex phase-cancellation when layering multiple OSCG instances.

7th Character: Integrated GAIN Envelope

Applies an internal envelope to the Gain stage. This is useful if in a layer there are more than one oscillator and they must have different envelopes for GAIN. You have to set a null GAINENV (which is the default) and use a different envelope setting for each oscillator. E.g. for an LFO just set gain = 1 and envelope of LFO type. **See the Envelope functions below for details on the parameters meaning, particularly ENV for the “opt” parameter.**

| Code | Type | Parameters |
|----------|----------------|--|
| 0 | None | None |
| L | Attack/Release | atk, rel (LFO style, 100% sustain) (Mono continuous) |
| S | ADSR | atk, dec, sus, rel (Simple envelope) (Mono continuous) |
| D | Double Decay | atk, d1, d1_lvl1, d2, sus, rel (Mono continuous) |
| F | Full (DAHDSR) | opt, del, atk, hld, dec, sus, rel (Mono continuous) |

The 8th character: filter topology & type

The 8th character defines the integrated filter topology, default frequency and type. If missing, it defaults to ‘0’ (No filter).

| Char | Topology | Cutoff Source (Fc) | P. added to OSCG |
|------|----------|--------------------|------------------|
|------|----------|--------------------|------------------|

| Char | Topology | Cutoff Source (Fc) | P. added to OSCG |
|------|--|--------------------|-----------------------|
| '0' | No Filter | - | None |
| 'f' | Classic (Filt -> Gain) | Internal DEFAULTFC | type, reso, drive |
| 'F' | Classic (Filt -> Gain) | External Parameter | type, fc, reso, drive |
| 'd' | Drive (Gain -> Filt) | Internal DEFAULTFC | type, reso, drive |
| 'D' | Drive (Gain -> Filt) | External Parameter | type, fc, reso, drive |

Oscillator Signal Flow Topology

The OSCG function allows you to toggle the internal routing between the **Filter** and the **Gain/Envelope (VCA)** stage. Use the following table to choose the best mode for your sound:

| Topology Mode | Logic | Best For | Technical Characteristics |
|-------------------------------------|-------------------------------|---|---|
| Pre-Gain Filter
(Classic) | Output = GAIN * FILTER (Wave) | Standard subtractive synthesis, Lead, Pads, Plucks. | The filter receives a constant 0dB signal. Resonance and saturation remain consistent regardless of the envelope level. |
| Post-Gain Filter
(Drive) | Output = FILTER (GAIN * Wave) | Aggressive sounds, Acid bass, Industrial, Lo-fi. | The input level of the filter is modulated by the envelope. This creates dynamic distortion in non-linear filters (like the Moog Ladder). |

Quick Selection Guide

| If you want... | Use this configuration: |
|----------------|--|
| Moog "Growl" | Drive + High Resonance + High Gain |

| If you want... | Use this configuration: |
|-----------------|--|
| Clean Sine Subs | Classic + Low Cutoff |
| Biting FM Leads | Classic + Phase Modulation (FM) |
| Soft, Warm Pads | Drive + Low Gain + Low Cutoff |

Implementation Guide & Best Practices

1. Handling Resonance and Tails

- **Classic Mode:** This is the safest bet. When the amplitude envelope (GAIN) hits zero, the sound (including any high-resonance "whistle" from the filter) is cut off immediately.
- **Drive Mode:** Be careful with high resonance. If the filter is in self-oscillation, it might continue to produce sound even if the input Wave is silent, as long as the GAIN is high.

2. Non-Linear Filters (Moog and others)

If you are using the **non linear** filter model:

- In **Post-Gain (Drive)** mode, the "Growl" and "Grit" of the filter will follow your envelope. The sound will be distorted during the *Attack* and clean up during the *Sustain* or *Release*.
- In **Pre-Gain (Classic)** mode, the saturation is static. Use this if you want a predictable, professional-grade analog tone.

3. Modulation & Anti-Aliasing

- **Smoothing:** If your GAIN or DEPHASE (FM) modulations are very fast or "jagged," placing the **Filter after the Gain** (Post-Gain) can act as a natural low-pass, smoothing out potential digital clicks or aliasing artifacts before they reach the output.
- **FM Precision:** When using heavy Phase Modulation, filtering *after* the gain stage helps contain the massive harmonic expansion that occurs at high modulation indexes.

4. CPU Optimization

The integrated filter in OSCG is highly optimized. Using the internal filter is significantly faster than piping the output of a "dry" oscillator into a separate filter function.

Filter Type Table (Parameter _{type})

The type parameter is an integer that defines the slope, shape and distortion (Linear vs Moog and other) of the filter.

The Base Mode & Slope Matrix

The base value of type determines the filter's slope and type. Unlike standard filters, this engine includes ultra-lean **6dB/oct** modes for subtle tone balancing. Reso parameter is ignored for -6dB filters.

| Base Code | Slope | Filter Type | Stages |
|----------------|----------------|-------------|--------|
| -2 | 6 dB/oct | High-Pass | 1/2 |
| -1 | 6 dB/oct | Low-Pass | 1/2 |
| 0 - 3 | 12 - 48 dB/oct | Low-Pass | 1-4 |
| 4 - 7 | 12 - 48 dB/oct | High-Pass | 1-4 |
| 8 - 11 | 12 - 48 dB/oct | Band-Pass | 1-4 |
| 12 - 15 | 12 - 48 dB/oct | Notch | 1-4 |
| 16 - 19 | 12 - 48 dB/oct | All-Pass | 1-4 |

The Saturation Offsets (The "Drive" Logic)

By adding an offset to the base codes, you change the internal mathematics of the filter stages. In saturated modes (98+), the **DRIVE** parameter controls the intensity of the non-linear processing. In linear mode the DRIVE just multiplies the input signal.

| Range | Type | Logic | Character |
|-------------------|-------------------|---|---|
| -2 to 19 | Linear | Standard SVF | Transparent, no harmonic coloring. |
| 98 to 119 | Moog Tanh | $\tanh f(\text{in} * \text{drv})$ | Creamy analog warmth. Odd harmonics. |
| 198 to 219 | Hard Clip | $\text{clamp}(\text{in} * \text{drv}, -1, 1)$ | Aggressive, industrial, squared-off. |
| 298 to 319 | Tube/Valve | $2x - x^2$ (Quadratic)* | Fat even harmonics, mimics vacuum tubes. |
| 398 to 419 | Cheby 2 | $T_2(\text{in} * \text{drv})$ blended* | Harmonic Octave. Adds a pure upper octave. |

| Range | Type | Logic | Character |
|------------|---------|-----------------------|--|
| 498 to 519 | Cheby 3 | T3(in * drv) blended* | Harmonic Fifth. Nasal, hollow, aggressive. |
| 598 to 619 | Cheby 4 | T4(in * drv) blended* | Glassy/Chime. High-frequency reinforcement. |

Note on Saturation Physics: *The internal filters of **OSCG** share the same non-linear engine as the **MOOGG** function. For a detailed technical breakdown of how **Tube (300)** and **Chebyshev (400+)** modes affect the harmonic spectrum and signal phase, please refer to the **MOOGG** documentation.

Chebyshev Harmonic Saturation in OSCG

The **400, 500, and 600 series** transforms the internal OSCG filter from a subtractive tool into a **harmonic generator**.

Interacting with Topology (Character 8)

The behavior of Chebyshev saturation changes drastically based on your chosen routing:

- **Pre-Gain (Classic) Filter ('f', 'F'):**

The saturation is constant. This is ideal for creating "Locked Harmonic" sounds. For example, using **Cheby 2 (Offset 400)** on a simple Sine wave with high resonance will make the oscillator sound like a stable dual-oscillator rank (Fundamental + Octave).

- **Post-Gain (Drive) Filter ('d', 'D'):**

This is where the Chebyshev modes truly shine. Since the drive is modulated by the internal envelope (GAIN), the harmonic content evolves over time.

- **The "Harmonic Bloom":** During the Attack phase, as the GAIN increases, the upper harmonics generated by the Chebyshev polynomial "bloom" and become more prominent.
- **Dynamic Resonant Tails:** On long releases, the resonance will shift from a complex harmonic chord back to a pure sine tone as the signal level drops below the saturation threshold.

Technical Note: Drive vs. Harmonics

In Chebyshev modes, the **DRIVE** parameter acts as a "Harmonic Exciter" control.

- At low **DRIVE** (0.1 - 0.5), the filter behaves almost linearly.
- At high **DRIVE** (> 1.0), the chosen harmonic (Octave, Fifth, etc.) becomes dominant, effectively "re-tuning" the filter's resonance peak.

For other implementation considerations, see the help of the MOOGG function below. For advanced examples see the MOOGG function help or the tutorial and FAQ section.

Examples:

1. Synthesized Waveforms

A. Basic Sine Wave Oscillator Utilizes the default frequency (FREQ) and gain (GAIN) variables defined in the COMMON section.

```
// Character 1: 's' (Sine), 2: '1' (Gain 1), 3: 'f' (Default FREQ)
OUT = OSCG("s1f00000")
```

Note: This is equivalent to the default internal string "s1f00000".

B. Smooth Synth Waveform with Manual Control Generates a sawtooth wave with manual gain (G) and frequency (F) inputs.

```
// Character 1: 'y' (Synth), 2: 'k' (GAIN * k), 3: 'v' (Manual Freq)
// Parameters: 2 (Sawtooth Type), 0.9 (Smoothness), G (Gain), F (Freq)
O = OSCG("ykvL0000", 2, 0.9, G, F)
```

The 'L' in character 4 enables Autophase, simulating an "ever-running" analog oscillator.

2. Sample-Based (Slotted) Oscillators

A. Simple Sample Playback Plays the audio data stored in a specific sample slot.

```
// Character 1: 'S' (Stereo Slot), 2: '1' (Gain 1), 3: 'f' (Default FREQ)
// Parameter: 200 (Sample Slot Index)
OUT = GAIN * OSCG("S1f00000", 200)
```

In this example, the internal gain is fixed at 1, but the output is scaled by the global GAIN keyword.

B. Slotted Oscillator with Integrated Envelope Applies a simple ADSR envelope directly within the function for efficiency.

```
// Character 7: 'S' (Simple Envelope: Attack, Decay, Sustain, Release)
// Parameters: Slot 200, Attack 0.01s, Decay 0.1s, Sustain 0.8, Release 0.2s
OUT = OSCG("Sgf000S0", 200, .01, .1, .8, .2)
```

The 'g' in character 2 automatically applies the default GAIN variable.

3. Low-Frequency Modulation (LFOs)

The distinction between an oscillator and an LFO is primarily defined by frequency (typically < 20 Hz) and the disabling of Autophase to ensure a consistent starting phase.

A. Standard Sine LFO

```
// 5 Hz Sine wave with Autophase disabled (Character 4: '0')
FreqMod = OSCG("s1v00000", 5)
```


B. Performance LFO with Delay and Attack

```
// Character 7: 'L' (LFO Envelope: Attack + Release)
// Parameters: 5 Hz speed, 0.5s Attack fade-in, 0.1s Release fade-out
LFO = OSCG("slv000L0", 5, 0.5, 0.1)
```

4. Using the filter step: The "Digital Pipe" Organ

Using Chebyshev 3 (Offset 500) to create a hollow, pipe-like sound with a dynamic harmonic strike.

```
// OSCG Configuration:
// Character 8 = 'D' (Post-Gain Filter, External Params)
// Saturation = 500 (Cheby 3 / Fifth) + Base 1 (24dB LP) = 501

E = ENV0(0.01, 0.4, 0.0, 0.2) // Percussive envelope
OUT = OSCG("SAW...D", 440, E, 501, 1200, 0.8, 1.5)
```

In this example, the 'Fifth' harmonic (Offset 500) will be strongest at the start of the note (high GAIN), giving the sound a sharp, nasal pipe-organ "chiff" before settling into a smoother filtered tone.

5. Advanced Implementation: Phase/FM Modulation

Crescendo supports true Frequency Modulation by modulating the phase parameter (Character 6).

A. Operator-Style FM (Carrier and Modulator)

```
// Step 1: Create a modulator (Sine, Unitary Amplitude, 2x Frequency)
Modulator = OSCG("skk00000", 1.0, 2.0)

// Step 2: Use modulator to shift the phase of the carrier (Character 6: 'P')
OUT = OSCG("slf00P00", Modulator)
```

This provides high-fidelity FM synthesis without frequency-stepping artifacts.

B. Advanced operator style frequency modulation.

You need phase modulation sinusoidal waves, of unitary amplitude:

OSCG("slf00000"): the frequency is FREQ Hz. Useful for earliest modulators.
OSCG("slf00P00", <phase>): the frequency is FREQ Hz and the phase is <phase> radians.
OSCG("slk00P00", <freqm>, <phase>): the frequency is <freqm> * FREQ Hz and the phase is <phase> radians.
OSCG("slv00P00", <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>)

The **SUPERSAW** family of functions consists of specialized oscillators designed to generate lush, stereo sawtooth waveforms with multiple detuned harmonics. These functions are ideal for recreating the classic "supersaw" sound popularized by vintage hardware synthesizers like the Roland JP-8000. Crescendo provides three variants: the full **SUPERSAW** function for maximum control and the simplified **SUPERSAW0** and **SUPERSAW1** variants for rapid implementation.

While you could technically build a Supersaw by summing multiple individual **OSCG** sawtooth oscillators, Crescendo's specialized **SUPERSAW** functions are significantly more CPU-efficient and offer native stereo spreading that would be tedious to program manually.

A Supersaw consists of one central sawtooth wave and several pairs of detuned sawtooth "sidebands." These create the thick, "reedy" texture famous in Trance, EDM, and cinematic pads.

1. The Core Parameters

- **<detune>**: The distance between the harmonics.
 - **Positive values**: Use non-linear spacing (harmonic clusters).
 - **Negative values**: Use linear spacing (mathematically even).
 - *Tip: Use a stereo signal here (e.g., `JOIN(0.02, -0.02)`) to create a wide stereo field.*
- **<n>**: The number of harmonic pairs. The total number of oscillators is $2|n| + 1$.
 - **Max value is 15** (total of 31 sawtooths).
 - **AUTOPHASE**: If **<n>** is positive, the oscillators use "free-running" phase (organic variation). If negative, they are phase-locked.
- **<mix>**: The volume of the sidebands.
 - **Positive values**: All sidebands have the same volume.
 - **Negative values**: Volume decreases as they get further from the center frequency.

2. The Three Variants

A. SUPERSAW0 (Rapid Setup)

The "quick-start" version. It uses the current note's `FREQ`, has the filter enabled, and sets `<n>` to 3 (7 total oscillators).

```
OUT = GAIN * SUPERSAW0(<detune>, <mix>)
```

B. SUPERSAW1 (Density Control)

Identical to `SUPERSAW0`, but lets you choose how many harmonics (`<n>`) to include.

```
OUT = GAIN * SUPERSAW1(<detune>, <mix>, <n>)
```

C. SUPERSAW (Full Control)

The full laboratory version. You manually define the frequency, phase, and whether the internal 6dB/oct high-pass filter is active.

```
OUT = GAIN * SUPERSAW(<freq>, <phase>, <detune>, <mix>, <n>, <filter>)
```

3. Strategic Examples

The "Trance Lead" (Wide & Sharp)

To get that classic high-energy lead, use `SUPERSAW1` with a high harmonic count and a stereo-spread detune.

```
LAYER
// 15 oscillators (n=7), wide stereo detune, sidebands at 60% volume
OUT = GAIN * SUPERSAW1(JOIN(0.05, -0.05), 0.6, 7)
```

The "Organic Pad" (Decreasing Amplitude)

For a softer, more "vintage" feel, use a negative `<mix>` so the outer harmonics are quieter than the inner ones.

```
LAYER
// Negative mix (-0.7) creates a smoother, less "buzzy" texture
OUT = GAIN * SUPERSAW1(0.03, -0.7, 4)
```

The "Sub-Heavy" Supersaw

The built-in filter in `SUPERSAW0/1` is a 6dB/oct high-pass. This is vital because 31 detuned sawtooth waves can create massive, muddy low-end build-up that ruins a mix.

```
// SUPERSAW0 has the filter ON by default to keep the low-end clean
OUT = GAIN * SUPERSAW0(0.04, 0.5)
```

Classic Lush Pad (`SUPERSAW0`) The `SUPERSAW0` function is pre-configured with the filter enabled, phase at zero, frequency tied to the default `FREQ`, and exactly 7 harmonics (`n=3`).

```
LAYER
// Standard lush detuned sound
// Detune: 0.05 (stereo spread), Mix: 0.5 (amplitude of side harmonics)
OUT = GAIN * SUPERSAW0(0.05, 0.5)
```

High-Density Stereo Lead (SUPERSAW1) The SUPERSAW1 variant allows you to specify the number of harmonics while keeping the filter and frequency defaults of SUPERSAW0.

```
LAYER
// Dense 11-harmonic lead (n=5)
// Detune: 0.02, Mix: 0.4, 5 couples of harmonics
OUT = GAIN * SUPERSAW1(0.02, 0.4, 5)
```

Manual Precision Control (Full SUPERSAW) Use the full function when you need custom frequencies, manual phase management, or to disable the internal high-pass filter.

```
LAYER
// Frequency: 220Hz, Phase: 0, Detune: 0.1, Mix: -0.8 (decreasing amplitude),
// 3 couples (n=3), Filter: 0 (disabled)
OUT = GAIN * SUPERSAW(220, 0, 0.1, -0.8, 3, 0)
```

"Ever-Running" Synth with AUTOPHASE By ensuring the <n> parameter is positive, the function simulates an analog oscillator that is always running, creating organic variations every time a note is triggered.

```
LAYER
// Positive n=7 enables AUTOPHASE for a more organic, varying detuned unison
OUT = GAIN * SUPERSAW1(0.08, 0.6, 7)
```

Developer's Note: Phase and Texture

If you want the sound to be identical every time you hit a key (punchy and consistent), use a **negative** <n>. If you want it to feel like a real hardware synth that is "always running" (varying and lush), keep <n> **positive** to engage **AUTOPHASE**.

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 distance 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.

Features:

- **Stereo Processing:** All `SUPERSAW` functions generate stereo output by default. To enhance the stereo image, use stereo signals for the `<detune>` or `<mix>` parameters.
- **Aliasing Mitigation:** Sawtooth waveforms are harmonically rich. The built-in 6dB/octave high-pass filter (enabled by default in `SUPERSAW0/1`) helps clean up low-frequency clutter in dense detuned patches.
- **Power oversampling by default:** the oversampling factor is given by the last parameter of the `QUALITY` instruction, capped at 257x.
- **Efficiency:** These specialized functions are more CPU-efficient than manually summing individual oscillators to create unison effects.

SINC(<op1>)

The `SINC` function generates a stereo "Whittaker–Shannon" waveform, also known as the sampling function. In digital signal processing, the `SINC` function is the mathematical representation of a perfect low-pass filter. In *Crescendo*, it functions as a unique oscillator that produces a rich, evolving spectrum that is distinct from standard geometric waves like Saws or Squares.

1. The Mathematical Foundation

The `SINC` instruction follows the standard formula:

$$\text{Sinc}(x) = \sin(x) / x$$

In the context of the *Crescendo* engine, x is defined as $\text{op1} * T$, where T is the time elapsed since the Note ON trigger.

- **Trigger Behavior:** At $T = 0$, the function evaluates to exactly **1.0**, providing a consistent, punchy start for every note.
- **Time Evolution:** As T increases, the waveform oscillates but decays in amplitude over time.
- **Spectral Result:** Unlike a sine wave (single frequency) or a saw wave (infinite harmonics), a `SINC` wave has a concentrated energy burst at the start that smooths out, creating a "pulsing" or "rippling" tonal character.

2. Strategic Applications

A. The "Bell-Like" Transient

Because the `SINC` function is strongest at $T=0$ and naturally decays, it is excellent for creating the initial "strike" of a bell, mallet, or electric piano.

```
LAYER
// A high-frequency SINC pulse creates a sharp, metallic "tine" sound
OUT = GAIN * SINC(2000)
```

B. Spectral Sweeping (The "Laser" Effect)

If you modulate the input `<op1>` using an envelope or a VST knob, you create a sweeping effect that sounds more organic and "liquid" than a standard filter sweep.

```
LAYER
// Use an envelope (E1) to sweep the Sinc frequency
// Creates a "Pew-Pew" laser or a liquid synth drop
OUT = GAIN * SINC(FREQ * E1)
```

C. Wide Stereo Texture

Since `SINC` can accept a stereo input, you can create a massive sense of space by slightly offsetting the frequency between the left and right channels.

```
LAYER
// Frequency is slightly higher in the left ear than the right
// Result: A wide, shimmering spectral wash
OUT = GAIN * SINC(JOIN(440,442))
```

D. Frequency-Linked Sinc Oscillator

In this scenario, the `SINC` function is driven by the default frequency (`FREQ`) calculated in the `COMMON` section, but scaled by a GUI knob (VST Variable #0) to create a "spectral sweep" effect.

```
// --- COMMON SECTION ---
VSTVAR 0, 1.0, "Sinc Scale", "", 0.1, 10, 1

// --- LAYER SECTION ---
LAYER
// Generate the sinc waveform
// Scale the default frequency by the VST knob
// Resulting spectral content shifts as ScaleFactor or FREQ changes
// Apply global gain and output
OUT = GAIN * SINC(FREQ * VAR(600))
```

3. Comparison: SINC vs. Sine

| Feature | SINE | SINC |
|---------------|---|--|
| Harmonics | Pure fundamental only. | Wide spectrum at start, narrowing over time. |
| Initial Phase | Can be varied. | Always 1.0 (Sin(0)/0) at trigger. |
| Amplitude | Constant (unless shaped by ENV). Naturally decaying (1/T behavior). | |
| Best Used For | Sub-bass, pure tones. | Transients, liquid leads, bells. |

Developer's Note

The SINC function is computationally inexpensive but very sensitive to the `<op1>` value. High values result in extremely fast oscillations that can lead to aggressive, "noisy" textures, while low values result in a slow, booming "thud" that eventually settles into a DC offset. Experiment with values between **20** and **2000** for the most musical results.

WAVETABLE(<num>,<op1>)
OR
WAVETABLE01(<num>,<op1>)
OR
WAVETABLES(<num>,<op1>)

The `WAVETABLE` family of functions in Crescendo provides sample-accurate control over the playback "playhead" of a sample slot. Unlike a standard oscillator that plays at a set pitch, these functions allow you to treat a sample as a **lookup table (LUT)** or a **custom waveform** that you can "scrub" or scan through in real-time.

1. Function Definitions

There are three variants based on how they interpret the "position" parameter:

- **WAVETABLE(slot, time)**: The position is defined in **seconds** (from 0 to the end of the sample).
- **WAVETABLE01(slot, position)**: The position is normalized from **0.0 to 1.0**, representing the start to the end of the sample.
- **WAVETABLES(slot, position)**: The position is normalized from **-1.0 to 1.0** (often used for bipolar signals like sine waves or audio inputs).

2. Technical Logic

`WAVETABLExx(<num>, <op1>)`

- **<num>**: The index of the sample slot (loaded via `SAMPLE`, `SAMPLES` or `RENDER`). Mono and sampled at trigger time.
- **<op1>**: The playback position. This parameter can be stereo and modulated at the sample rate.
 - In **seconds** for `WAVETABLE`. Range from 0 to `<sample duration in seconds>`
 - Between 0 and 1 for `WAVETABLE01`, spanning the whole sample.
 - Between -1 and 1 for `WAVETABLES`, spanning the whole sample.
 - For `<op1>` out of range the function will evaluate as zero.
- **Interpolation**: Crescendo uses the current `QUALITY` setting (up to `SINC` interpolation) to calculate the perfect value.

3. Practical Examples

Example A: The "Scrubbing" Sampler (Granular Logic)

You can use a VST knob or an LFO to manually move the playhead through a recording. This allows for time-stretching or "freezing" a sound without affecting its pitch.

```
// --- LAYER SECTION ---
LAYER
// Scrub through Slot 300 using VST Knob #0 (VAR 600)
// Because VAR(600) is 0 to 1.0, WAVETABLE01 is the easiest choice.
// It automatically maps the knob range to the full length of the sample.
OUT = GAIN * WAVETABLE01(300, VAR(600))
```

Example B: The "Phase-Locked" Custom Oscillator

Standard samplers can sometimes have slight phase drift. By using `WAVETABLE` with a phase ramp (as seen in Example A), you create an oscillator that is perfectly stable. This is the foundation of **Wavetable Synthesis**.

- **Scanning:** If you have a long sample containing different "shapes," you can add a "Position" knob to scan through those shapes while the note is playing.

Example C: Custom Periodic Oscillator

To turn a short sample into a playable synth oscillator, you feed it a **Phase Ramp** (a sawtooth wave that resets every cycle). This ensures the sample plays at the correct frequency (`FREQ`).

```
// --- LAYER SECTION ---
LAYER
// 1. Create a phase ramp that goes 0.0 -> 1.0 at the current note's frequency
Phase = FRAC(TIME * FREQ)

// 2. Use that phase to cycle through a waveform in Slot 200
// WAVETABLE01 ensures the single-cycle wave fills exactly one period.
WarpedWave = WAVETABLE01(200, Phase)

OUT = GAIN * WarpedWave
```

Example D: Custom Harmonic Distortion (Wave-Shaping)

You can use a sample (like a recorded tube saturation curve) as a transfer function. By feeding an incoming audio signal (like `IN0`) into `WAVETABLES`, the input signal determines the playhead position.

```
// --- POST SECTION ---
POST
// Drive an input signal into Slot 201 (a saturation curve).
// Since audio signals are usually bipolar (-1 to +1),
// WAVETABLES maps the input perfectly across the lookup table.
DistortedAudio = WAVETABLES(201, IN0)

OUT = DistortedAudio
```

4. The Precision Problem: Why TIME is Risky in Example C

In digital audio, floating-point numbers lose precision as they get larger.

- **In a LAYER:** A note usually lasts only a few seconds. The loss of precision is negligible because the math is still operating on relatively small numbers.
- **In the POST section:** The `TIME` variable represents the total seconds since the plugin was initialized. After several minutes (or hours) of a DAW session, the value of `TIME` becomes so large that the decimal precision required to calculate `FRAC(TIME * FREQ)` accurately at 44.1kHz or 48kHz begins to break down. This results in audible "phase jitter" or a noisy, "crackly" oscillator.

The Robust Solution: `WAVETABLES` + `OSCG`

Instead of calculating phase manually, we use the engine's built-in sawtooth oscillator to drive the lookup. The `OSCG` function uses an internal double precision phase accumulator, ensuring the sound remains crystal clear regardless of how long your DAW has been running.

Since a sawtooth oscillator outputs a bipolar signal from **-1.0 to 1.0**, we use `WAVETABLES` to match that range perfectly.

Example: Precision-Safe Custom Oscillator

```
// --- LAYER SECTION ---
LAYER
// 1. Generate a high-precision Sawtooth phase ramp (-1 to 1)
// We use a gain of 1.0 and no envelope here as it's just a phase driver.
// 2 = asymmetric triangle, 1 = ascending Sawtooth
PhaseRamp = OSCG("y1f0000", 2, 1)

// 2. Drive the Wavetable using the Sawtooth
// WAVETABLES expects the -1 to 1 range, making it the perfect partner for OSCG.
// This will never lose precision, even after hours of playback.
CustomTone = WAVETABLES(200, PhaseRamp)

// 3. Apply the actual Gain and Envelope to the result
OUT = CustomTone * GAIN * ENV0(0.01, 0.2, 0.5, 0.4)
```

Comparing the Methods

| Method | Implementation | Stability | Best Used For... |
|----------------------|---|-------------------------------|--------------------------------------|
| Manual Phase | <code>WAVETABLE01(..., FRAC(TIME*F))</code> | Low (Decays over time) | Quick tests or very short layers. |
| Bipolar Drive | <code>WAVETABLES(..., OSCG("y1..."))</code> | High (Sample-accurate) | Production-ready instruments. |

Summary of Benefits

1. **No Jitter:** By avoiding global `TIME` for phase calculations, you eliminate the "crunchy" artifacts that appear in long sessions.

2. **Phase Control:** Since you are using a real `OSCG`, you can use all the standard oscillator flags (like `p` for phase modulation) to warp the wavetable playback dynamically.
3. **Post-Section Safety:** This is the only reliable way to implement wavetable-based effects (like custom distortions) in the `POST` section where precision loss is most aggressive.

5. Strategic Comparison

| Feature | WAVETABLE | WAVETABLE01 | WAVETABLES |
|---------------------|-----------------------------|--------------------------|---------------------------|
| Input Unit | Seconds (Absolute) | 0.0 to 1.0 (Relative) | -1.0 to 1.0 (Relative) |
| Best For | Granular Scrubbing / Delays | Phase-locked Oscillators | Wave-shaping / Distortion |
| Out of Range | Returns 0 | Returns 0 | Returns 0 |

Developer's Note: Always ensure the `QUALITY` setting in your `settings.ini` is high when using these functions for synthesis. Because the playhead can move at any speed, low-quality interpolation can introduce aliasing or noise.

6. WAVETABLE vs. OSCG

| Feature | OSCG (Sampler) | WAVETABLE |
|--------------------|-----------------------------------|-------------------------------|
| Control | Pitch-based (<code>FREQ</code>) | Time-based (Seconds) |
| Looping | Supports complex loops/release | Single loop or One-shot only |
| Flexibility | Set and forget | Highly programmable |
| CPU Speed | Fast | Slightly faster for one-shots |

Developer's Note: The "Tiny Frequency" Trick

If you need the power of `WAVETABLE` but require complex looping or the "separate release" features of a standard sampler, you can "fake" it with a standard oscillator.

By setting the frequency to a near-zero value (e.g., `1e-30`) and modulating the **Phase** parameter, you can achieve manual scrubbing on a complex sampled sound.

However, for single-cycle oscillators or custom distortion curves, the `WAVETABLE` function is the cleaner, more professional choice.

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.

The interpolation is performed with the current QUALITY settings, so it supports windowed sync interpolation, but ignores OVERSAMPLING.

GRAINSYNTH(<num>, <size>, <crossfade>, <position>)

The GRAINSYNTH function is a high-performance granular oscillator. It decomposes a sample into fragments ("grains") and reassembles them. Unlike standard playback, it allows you to freeze time, create lush clouds, or perform "scrubbing" through a sample without changing its pitch.

1. Technical Syntax

GRAINSYNTH(<num>, <size>, <crossfade>, <position>)

- **<num> (Sample Slot):** The index of the sample slot (loaded via SAMPLE, SAMPLES or RENDER). Mono and sampled at trigger time.
- **<size> (Grain Size):** The duration of each grain in samples.
- **<crossfade> (Fade Duration):** The length of the sine-based equal-power crossfade at the start/end of each grain (in samples).
- **<position> (Absolute Position):** The specific starting sample index within the source file.

2. Operational Logic

- **Trigger Initialization:** At Note-On, the engine samples the initial <position> and begins the first grain.
- **Boundary Resampling:** To ensure phase stability and prevent metallic artifacts, the <position> parameter is **resampled only at the end of a grain cycle**.
- **Absolute Offset:** The <position> is an absolute index. If your sample is 44,100 samples long (1 second), setting <position> to 22,050 will always start grains from the middle of the file, regardless of the grain size.
- **Equal-Power Windowing:** The function uses a **Sine-based crossfade** to ensure constant perceived volume during grain transitions, eliminating the "dipping" effect of linear ramps.

3. Practical Examples

Example A: The "Time-Freeze" Scanner

Map a VST knob to the position to manually "scrub" through a sound. Because the position is absolute, the movement is smooth and intuitive.

```
// VST Var 0 (Index 600) scans from sample 0 to 100,000
// Grain Size: 2000 samples, Crossfade: 400 samples
LAYER
OUT = GRAINSYNTH(300, 2000, 400, VAR(600))
```

Example B: Randomized "Stochastic Cloud" (Using MCC 157)

Use the **Volatile Noise (157)** to pick a new random location in the file every time a grain finishes. This creates a dense, non-repeating texture.

```
// Every time a grain ends, a new position is chosen randomly
// within the first 50,000 samples.
LAYER
OUT = GRAINSYNTH(300, 1500, 300, MCC(157) * 50000)
```

Example C: Stable Multi-Note Textures (Using MCC 139)

Use the **Latched Random (139)** to give each MIDI note a unique but stable starting point.

```
// Each note picks a spot in the file and stays there,
// providing a different "timbre" for every key pressed.
LAYER
OUT = GRAINSYNTH(300, 3000, 500, MCC(139) * 80000)
```

4. Developer's Notes

- **Performance:** GRAINSYNTH handles complex buffer management internally. It is significantly more efficient than using multiple layers to simulate granulation.
- **Quality:** The interpolation honors the global QUALITY settings (Windowed Sinc), ensuring high-fidelity results even when grains are very short.
- **Stereo:** Parameters are stereo-independent. You can use different positions for Left and Right to create massive stereo width.

WAVESCAN(<num>, <size>, <position>, <frequency>, <dephase>)

The WAVESCAN function is a hybrid oscillator that combines granular logic with wavetable synthesis principles. It treats a segment of audio as a single periodic cycle, allowing you to turn any sample into a playable synthesizer oscillator. By scanning through the sample, you can achieve dynamic timbral movement, vocal-like formants, and complex digital textures.

1. Technical Syntax

WAVESCAN(<num>, <size>, <position>, <frequency>, <dephase>)

- **<num> (Sample Slot):** The index of the sample slot (loaded via SAMPLE, SAMPLES or RENDER). Mono and sampled at trigger time.
- **<size> (Grain Size):** The length of the periodic window in samples. This determines the "wavetable" resolution.
- **<position> (Absolute Position):** The specific starting sample index in the source file where the scan begins.
- **<frequency> (Hz):** The target playback pitch.
- **<dephase> (Offset):** A continuous phase offset in samples, allowing for phase-distortion and PWM-like effects.

2. Operational Logic

Periodic Waveform Assumption (Playback Pitch)

The core mechanic of WAVESCAN is that **a single grain of size <size> is assumed to be a single complete cycle of the waveform.** Unlike standard sample playback which moves at a fixed rate, the scanning speed of the grain is dynamic: it is calculated based on the requested **<frequency>** and the system **Sample Rate**. To produce a cycle at 440Hz, the engine will scan the entire length of the <size> window 440 times per second.

Phase Stability & Resampling

To ensure pitch integrity, the **<position>** (the "where" in the file) is **resampled only when the internal phase completes a full cycle** (crosses the grain border). This prevents the pitch from "warbling" or mangling when you move the scanning knob.

Continuous Dephase

The **<dephase>** parameter is applied in real-time. It shifts the readout point within the sample *without* affecting the oscillator's main frequency timer. This creates a "sliding" window within the grain, enabling smooth, aggressive "growling" modulations.

3. Practical Examples

Example A: Manual Timbre Scanning

Map a VST knob to the position to sweep through a vocal or instrument sample.

```
// A grain of 1024 samples is played at the current keyboard
Frequency.
// Moving VAR(600) scans through the file's timbres without
changing pitch.
LAYER
OUT = WAVESCAN(200, 1024, VAR(600), FREQ, 0)
```

Example B: Harmonic Jitter (Using MCC 157)

Use **Volatile Noise (157)** to add "analog" instability. Since the position is resampled at the end of every cycle, each wave cycle will look slightly different.

```
// Each cycle pulls a slightly randomized wave shape from the
sample.
JitterPos = 10000 + (MCC(157) * 50)
LAYER
OUT = WAVESCAN(200, 512, JitterPos, FREQ, 0)
```

Example C: Phase Distortion Growl

Modulate the **<dephase>** parameter for aggressive, modern bass textures.

```
// Rapidly shift the phase within the grain for a "growling"
effect.
LFO_Mod = 200 * OSCG("s1v0000", 8)
LAYER
OUT = WAVESCAN(200, 2048, 5000, FREQ, LFO_Mod)
```

4. Developer's Notes

- **Character:** If the source sample does not have perfect zero-crossings at the grain boundaries, WAVESCAN will produce a gritty, digital character. This is a hallmark of early wavetable synthesis.
- **Quality:** The engine uses **Windowed Sinc interpolation** based on your global QUALITY settings, which is vital for maintaining clarity when scanning small grains at high frequencies.
- **Stereo Width:** Since all parameters are stereo, using a slight <dephase> or <position> offset between the Left and Right channels creates an incredibly wide and lush stereo image.
- **Playback speed:** If the grain size is 1024 samples and the frequency is 440Hz, the "effective" playback speed is much higher than a standard sampler, as it must traverse those 1024 samples within a single 440Hz period.

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 HAHD SHR 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) / 128 of beat (not quantized).

2001 - 3000: sync. The envelope retriggers every (<option> - 2000) / 128 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).

Example A: Complex Exponential Envelope with Pre-Attack Delay This example uses the full ENV function to create a sound that waits before starting and has a distinct "hold" at its peak.

```
LAYER
// Option 0: Normal behavior
// Levels: Start 0 -> Peak 1.0 -> Sustain 0.7 -> End 0
// Holds: 0.05s delay (H1), 0.1s peak hold (H2), 0s sustain hold (H3)
// Curves: 0.1s Exp Attack, 0.3s Exp Decay, 0.6s Exp Release
E = ENV(0, 0, 1.0, 0.7, 0, 0.05, 0.1, 0, 0.1, 0.3, 0.6)
OUT = GAIN * E * OSCG("s1f0000")
```

Example B: Looped "LFO-Style" Envelope By setting the <option> to 1, the envelope will automatically re-trigger its attack phase once the decay stage finishes, acting as a custom-shaped LFO.

```
LAYER
// Option 1: Looped
// Ramps from 0 to 1 over 0.2s linear, then back to 0 over 0.2s linear
L = ENV(1, 0, 1, 0, 0, 0.001, 0.001, 0.001, -0.2, -0.2, 0.1)
// Use as a pitch modulator (2400 cents depth)
OUT = GAIN * OSCG("s1v00P0", SEMI(FREQ, L * 2))
```

Example C: Beat-Synced Rhythmic Retriggering Setting the <option> between 2001 and 3000 synchronizes the envelope re-triggering to the DAW's tempo. The rate is calculated as (option - 2000) / 128 beats.

```
LAYER
// Option 2128: (2128 - 2000) / 128 = 1 beat retrigger
// Creates a rhythmic "pulsing" effect every 1 beat
Pulse = ENV(2128, 0, 1, 0, 0, 0.001, 0.001, 0.001, -0.05, -0.1, 0.1)
OUT = Pulse * GAIN * OSCG("s1f0000")
```

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

The ENVCURVE function provides a highly flexible method for sound shaping by using a user-defined Lookup Table (LUT) to determine the levels of an envelope over time. Unlike standard ADSR envelopes with fixed linear or exponential curves, ENVCURVE allows you to define complex, non-linear trajectories for the attack, decay, and release stages by referencing a specific CURVE slot.

This function produces a 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 ($\langle\text{option}\rangle - 1000$) / 32 of beat (not quantized).

2001 - 3000: sync. The envelope retriggers every ($\langle\text{option}\rangle - 2000$) / 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.

$\langle\text{index}\rangle$ 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.

$\langle\text{H1}\rangle$: initial hold time (seconds). Before attack. Minimum 0.001.

$\langle\text{atk}\rangle$: attack time (seconds). Minimum 0.001.

$\langle\text{H2}\rangle$: final attack value hold time (seconds). Before decay. Minimum 0.001.

$\langle\text{dcy}\rangle$: decay time (seconds). Minimum 0.001.

$\langle\text{H3}\rangle$: H3, sustain value hold time (seconds). Before release. Minimum 0.001.

$\langle\text{rel}\rangle$: release time (seconds). Minimum 0.001.

1. Step One: Defining the Envelope Shape (CURVE Instruction)

Before using `ENVCURVE`, you must define the mapping of levels in the **COMMON** section using the `CURVE` instruction. For the envelope to function correctly, the LUT must contain a strictly increasing sequence of X-values that covers at least the interval from **0.0 to 3.0**.

The X-Axis Mapping Logic:

- **X = 0.0**: Represents the starting level and the value maintained during the `Hold1` period.
- **X = 0.0 to 1.0**: This range is linearly swept during the **Attack** phase.
- **X = 1.0**: Represents the peak attack level and the value maintained during the `Hold2` period.
- **X = 1.0 to 2.0**: This range is linearly swept during the **Decay** phase.
- **X = 2.0**: Represents the **Sustain** level and the value maintained during the `Hold3` period.
- **X = 2.0 to 3.0**: This range is linearly swept during the **Release** phase.

Example CURVE Definition:

```
// COMMON SECTION
// Define a "Double Peak" envelope shape in Slot 50
// x=0 (Start): 0.0
// x=0.5 (Mid-Attack): 1.0 -> creates a rapid initial rise
// x=1.0 (Peak): 0.8 -> peak level
// x=1.5 (Mid-Decay): 0.9 -> a slight swell during decay
// x=2.0 (Sustain): 0.4 -> sustain level
// x=3.0 (End): 0.0 -> final release level
CURVE 50, 0, 0, 0.5, 1, 1, 0.8, 1.5, 0.9, 2, 0.4, 3, 0
```

2. Step Two: Implementing the ENVCURVE Function

Once the curve is defined, the `ENVCURVE` function is called within a **LAYER** (or assigned in **COMMON** for replication) to drive a parameter such as volume or filter cutoff.

3. Complete Implementation Example

The following code demonstrates a complete instrument file using a custom-shaped volume envelope.

```
// --- COMMON SECTION ---
// Define a complex pluck-and-swell shape in CURVE slot 10
// Start 0 -> Peak 1.0 -> Swell 1.2 -> Sustain 0.5 -> End 0
CURVE 10, 0,0, 1,1.0, 1.5,1.2, 2,0.5, 3,0

// --- LAYER SECTION ---
LAYER
// Standard Note Trigger
ONNOTEON 0, 127, 0, 127

// Apply the custom ENVCURVE
// Option 0: Normal mode
// Index 10: Reference the CURVE above
// H1: 0.05s delay before attack
// Atk: 0.1s to reach peak
// H2: 0.01s hold at peak
// Dcy: 0.6s to decay to sustain
// H3: 0s hold before release
// Rel: 1.2s release time
MyEnv = ENVCURVE(0, 10, 0.05, 0.1, 0.01, 0.6, 0.001, 1.2)

// Use the envelope to scale a sine wave oscillator
OUT = MyEnv * GAIN * OSCG("s1f0000")
```

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).

Example A: Standard Linear ADSR For standard volume shaping where advanced hold times are unnecessary, use the `ENV0` variant.

```
LAYER
// Attack 0.1s, Decay 0.2s, Sustain 0.5 (50%), Release 0.5s
// Note: negative times indicate linear curves
Amplitude = ENV0(-0.1, -0.2, 0.5, -0.5)
OUT = Amplitude * OSCG("s1f0000")
```

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.

Example:

In this scenario, we define an instrument where the sound waits for a short duration after a key press and then holds its peak for emphasis before decaying. This demonstrates how ENV1 provides more rhythmic control than the standard ENV0 variant.

Code Snippet:

```
// --- COMMON SECTION ---
// Define VST Variables to allow the user to adjust timing in real-time
VSTVARS 2
VSTVAR 0, 0.25, "Pre-Delay", "s", 0.001, 1.0, 1 // VST Var #0 (Index 600)
VSTVAR 1, 0.1, "Peak Hold", "s", 0.001, 0.5, 1 // VST Var #1 (Index 601)

// --- LAYER SECTION ---
LAYER
// Apply the ENV1 function
// atk: 0.01s exponential (0.01)
// dcy: 0.4s exponential (0.4)
// sustain: 0.3 (30% level)
// rel: 0.5s linear (-0.5)
// delay: Linked to VST Var #0 (&600 for continuous sampling)
// hold: Linked to VST Var #1 (&601 for continuous sampling)
AmpEnv = ENV1(0.01, 0.4, 0.3, -0.5, &600, &601)

// Generate a sine wave scaled by the envelope and global gain
OUT = AmpEnv * GAIN * OSCG("s1f0000")
```

ENV2(<atk>, <dcy1>, <dcy1 level>, <dcy2>, <sustain>, <rel>)

The ENV2 function is a sample-accurate temporal shaping tool designed to create sophisticated AD1D2SR (Attack, Decay 1, Decay 2, Sustain, Release) envelopes. This variant introduces a

"Double Decay" stage, allowing developers to define a sound that has two distinct decay slopes before reaching the sustain level—a characteristic often found in instruments like the Yamaha DX7 or acoustic pianos.

This function generates a simple linear/exponential AD1D2SR envelope.

All parameters are mono and continuously automatable.

The envelope is a mono expression.

A "Double Decay" option is also available directly within the `OSCG` function's format string (using 'D' as the 7th character) for maximum execution speed.

<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).

<dcyl>: 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).

<dcyl_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).

Example: The "Complex Percussive Decay"

This example demonstrates a sound that features a very sharp initial "pluck" (Decay 1) followed by a slower, more melodic "tail" (Decay 2) that eventually settles into a low sustain level.

Code Snippet:

```
// --- COMMON SECTION ---
// Define a VST knob to allow the user to adjust the secondary decay tail
VSTVARS 1
VSTVAR 0, 1.5, "Tail Time", "s", 0.001, 5.0, 1 // VST Var #0 (Index 600)

// --- LAYER SECTION ---
LAYER
// Apply the ENV2 function
// atk: 0.01s exponential rise
// dcyl: 0.15s sharp exponential decay
// dcyl_level: Drops from peak (1.0) to 0.5 quickly
// dcy2: Linked to VST Var #0 (&600 for continuous sampling)
// sustain: 0.15 (15% volume)
// rel: 0.8s linear release (-0.8)
ComplexEnv = ENV2(0.01, 0.15, 0.5, &600, 0.15, -0.8)

// Generate a sawtooth wave scaled by the complex envelope and global gain
OUT = ComplexEnv * GAIN * OSCG("y1f0000", 2, 0.5)
```

Filter Functions

FILT0(<num>,<input>,<res>)

See **FILT** below, with `fc = DEFAULTFC`.

Practical Example: Performance-Linked High-Pass Filter This example demonstrates a filter that follows the keyboard and responds to a global envelope pluck.

```
// --- COMMON SECTION ---
// Set base cutoff to track the keyboard (FREQ)
FCMOD 0

// Add an envelope pluck of 2 octaves (2400 cents)
FCENV 2400, 0, 0.01, 0, 0.4, 0, 0.2

// --- LAYER SECTION ---
LAYER
Source = GAIN * OSCG("y1f0000", 2, 0.5)

// Apply a 12dB/oct High Pass filter (type 4)
// The cutoff is automatically handled by the instructions above
OUT = FILT0(4, Source, 0.5)
```

By using `FILT0`, the developer avoids writing complex mathematical formulas for the cutoff inside the layer.

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 $1/(4(1-\text{<res>}))$, 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. If <res> < 0, it is automatically calculated based on the order of the filter, to have maximum flatness when summing complementar bands.

<num> is mono, integer, sampled at NOTE ON time and specifies the filter type. Reso parameter is ignored for -6dB filters.

| Code | Slope | Filter Type | Stages |
|------|----------|-------------|--------|
| -2 | 6 dB/oct | High-Pass | 1/2 |

| Code | Slope | Filter Type | Stages |
|----------------|----------------|-------------|--------|
| -1 | 6 dB/oct | Low-Pass | 1/2 |
| 0 - 3 | 12 - 48 dB/oct | Low-Pass | 1 - 4 |
| 4 - 7 | 12 - 48 dB/oct | High-Pass | 1 - 4 |
| 8 - 11 | 12 - 48 dB/oct | Band-Pass | 1 - 4 |
| 12 - 15 | 12 - 48 dB/oct | Notch | 1 - 4 |
| 16 - 19 | 12 - 48 dB/oct | All-Pass | 1 - 4 |

For implementation details, applicable also to **FILT**, see **MOOGG** below.

Practical Example: Sweeping a 24dB Low-Pass Filter In this example, an LFO is used to modulate the cutoff frequency of a sawtooth wave.

```
LAYER
// Generate a 5Hz sine LFO for the sweep
SweepLFO = 1000 + 800 * OSCG("s1v0000", 5)

// Generate source audio
Source = GAIN * OSCG("y1f0000", 2, 0.5)

// Apply a 24dB/oct Low Pass filter (type 1)
OUT = FILT(1, Source, SweepLFO, 0.7)
```

In this configuration, the cutoff frequency oscillates between 200 Hz and 1800 Hz.

MOOGG0(<mode>,<signal>,<res>,<drive>)

See **MOOGG** below, with `fc = DEFAULTFC`.

MOOGG(<mode>, <signal>, <fc>, <res>, <drive>)

The **MOOGG** function is a non-linear, multi-stage filter engine. Unlike the standard **FILT** function, **MOOGG** applies saturation algorithms inside each filter stage. This creates a "cascaded" distortion effect where the harmonics generated by one stage are reshaped by the next.

Algorithm Offsets

The `mode` parameter determines both the filter slope/type and the saturation physics.

| Range | Algorithm | Formula | Character |
|-----------|------------|---------------------------------------|--|
| -2 - 19 | Moog Tanh | <code>tanhf(in * drive)</code> | Classic analog warmth; creamy odd harmonics. |
| 98 - 119 | Hard Sat | <code>clamp(in * drive, -1, 1)</code> | Aggressive, square-edged industrial grit. |
| 198 - 219 | Tube/Valve | $2x - x^2$ (Quadratic) | Thick, "fat" even harmonics; mimics vacuum tubes. |
| 298 - 319 | Cheby 2 | <code>T2(in * drive) blended*</code> | Octave. Pure 2nd harmonic; doubles the resonance frequency. |
| 398 - 419 | Cheby 3 | <code>T3(in * drive) blended*</code> | Fifth. 3rd harmonic; adds nasal "bite" and edge. |
| 498 - 519 | Cheby 4 | <code>T4(in * drive) blended*</code> | Super-Bright. 4th harmonic; metallic and glassy textures. |

*See CHB paragraph below.

The Base Mode & Slope Matrix

The base value of type determines the filter's slope and type. Unlike standard filters, this engine includes ultra-lean **6dB/oct** modes for subtle tone balancing. Reso parameter is ignored for -6dB filters. To select a filter, take the **Base Code** from the table below and add the **Algorithm Offset** (0, 100, 200, 300, 400, or 500).

| Base Code | Slope | Filter Type | Stages |
|-----------|----------|-------------|--------|
| -2 | 6 dB/oct | High-Pass | 1/2 |
| -1 | 6 dB/oct | Low-Pass | 1/2 |

| Base Code | Slope | Filter Type | Stages |
|----------------|----------------|-------------|--------|
| 0 - 3 | 12 - 48 dB/oct | Low-Pass | 1 - 4 |
| 4 - 7 | 12 - 48 dB/oct | High-Pass | 1 - 4 |
| 8 - 11 | 12 - 48 dB/oct | Band-Pass | 1 - 4 |
| 12 - 15 | 12 - 48 dB/oct | Notch | 1 - 4 |
| 16 - 19 | 12 - 48 dB/oct | All-Pass | 1 - 4 |

All-Pass Modes & Phase Modulation (FM)

Modes **16 to 19** do not change the frequency balance of the input signal but shift its **phase**. This is particularly powerful when used on complex signals like **Phase Modulation (FM)**:

- **Harmonic Alignment:** An All-Pass filter can "rotate" the phase of FM sidebands. If an FM lead sounds "thin" when layered with another track, shifting the All-Pass fc can align the phases to restore the fundamental's power.
- **Dispersive FM:** High-slope All-Pass filters (Code 19) cause "harmonic smearing." In percussive FM sounds (bells, metallic hits), this creates a dispersive effect where different frequencies reach the output at slightly different times, mimicking the physical properties of glass or thick metal.
- **Non-Linear Phase Distortion:** Because MOOGG is non-linear, using high drive on an All-Pass mode introduces phase-distortion without altering the EQ. This adds "analog hair" to a signal while keeping its spectral balance perfectly intact.

Algorithm Implementation

The MOOGG engine utilizes two distinct mathematical approaches to process audio, selectable via the `mode` parameter:

A. Non-Linear Cascaded Ladder — (Offsets 0-200)

- **Topology:** 4-stage cascaded integrators with a feedback loop.
- **Physics:** Each stage includes a saturation function (`tanh`, `clamp`, or `quadratic`).
- **The "Growl":** The classic analog "growl" comes from the fact that resonance is generated *through* these saturation stages. As you increase `res`, the peak is constantly reshaped and limited, preventing digital clipping while adding warmth.

B. Harmonic-Function Shaping (Chebyshev) — (Offsets 300-500)

- **Topology:** Polynomial Waveshaping within the feedback path.

- **Mathematics:** Unlike standard clipping, which creates a broad spectrum of noise, Chebyshev polynomials map the input to a specific harmonic interval:
 - $T_2(x) = 2x^2 - 1$ (Second Harmonic / Octave)
 - $T_3(x) = 4x^3 - 3x$ (Third Harmonic / Fifth)
 - $T_4(x) = 8x^4 - 8x^2 + 1$ (Fourth Harmonic / Double Octave)
- **Result:** This is "mathematical distortion." It doesn't just saturate; it literally **re-synthesizes** the resonance peak into a musical interval. For filter stability the polynomial is blended dynamically with the input (see CHB paragraph below).

Parameters

- **signal:** The input audio signal.
- **fc:** Cutoff frequency in Hz.
- **res:** Resonance (0.0 to 1.0). High values create a sharp resonant peak. If $res < 0$, it is automatically calculated based on the order of the filter, to have maximum flatness when summing complementary bands.
- **drive:** Internal gain multiplier. In Chebyshev modes, *drive* controls the intensity of the harmonic generation and the blending rather than just "clipping" the signal.

"Adaptive Gain & Multi-Slope Consistency"

The **Crescendo** filter engine is meticulously calibrated to provide a uniform response across all slopes and saturation models.

- **Consistency:** Thanks to our **Iterative Power Scaling**, switching from a 12dB to a 48dB slope maintains a cohesive output level. The internal drive is dynamically adjusted to compensate for the natural gain accumulation of high-order resonant stages.
- **Saturation Voicing:** Whether you choose the classic **Moog Tanh** or the aggressive **Chebyshev Harmonics**, the engine ensures that the saturation remains musical. At high Drive settings, all models converge to a controlled, powerful peak, allowing for extreme sound design without unpredictable volume jumps.
- **Headroom Management:** Even at maximum Resonance (0.75+) and Drive (5.0+), the filter transitions into a rich, self-oscillating character that stays within a professional gain range.

Mode 200+: The Quadratic "Exciter" (V-MOSFET & Push-Pull Tube)

While Mode 0 (Tanh) emulates the classic bipolar transistor (BJT) behavior found in standard Moog ladders, **Mode 200+** shifts the architecture to simulate the "**Square Law**" characteristics of Vacuum Tubes and V-MOSFETs (Vertical Metal-Oxide-Semiconductor Field-Effect Transistors).

The V-MOSFET / Tube Physics

In a standard transistor, the transition from linear to saturated is relatively abrupt. However, Tubes and V-MOSFETs possess a wider "transfer curve." Even at low input levels, the quadratic nature of the parabola ($2x - x^2$) begins to subtly reshape the waveform.

- **V-MOSFET Emulation:** V-MOSFETs are prized in high-end audio for their "tube-like" transfer characteristics but with faster transient response. Mode 200+ captures this by

providing a sharp, "bright" attack that feels more immediate and "hi-fi" than the darker, compressed Tanh mode.

- **Push-Pull Tube Configuration:** By applying this quadratic curve symmetrically to both the positive and negative cycles of the wave, we simulate a **Class AB Push-Pull Power Stage**. In this configuration, the even harmonics of each individual component are phase-canceled, but the resulting "folding" of the waveform creates a unique harmonic profile that is rich in upper-mid clarity.

Sonic Characteristics: Why it sounds "Brilliant"

The primary difference between Mode 200+ and other modes is its **Harmonic Excitation**:

1. **Low-Level Excitement:** Because the quadratic curve starts deviating from a straight line immediately, it adds a "sheen" to the sound even at `Drive = 0.5`. It acts as a harmonic exciter, making the filter output sound "expensive" and "open."
2. **Harmonic Bandwidth:** The parabolic shape generates a wider spread of harmonics compared to the "squashing" effect of Tanh. This preserves the "air" of the original signal while adding body.
3. **Zero-Crossing Detail:** The transition at the zero-crossing between the two parabolic arcs provides a subtle "bite" that helps the sound cut through dense mixes without needing extra EQ.

Usage Recommendation

- **Mode 0 (Tanh):** Use for "Vintage" sounds, thick bass, and instances where you want the filter to "eat" the transients for a smoother, liquid feel.
- **Mode 200+ (Quadratic):** Use for "Modern" leads, aggressive pads, or when you need the filter to remain "Bright" and "Present" regardless of the cutoff frequency. It is the ideal choice for strings and plucked sounds where high-frequency detail is paramount.

Mode 300+: Chebyshev Harmonic Saturation

Unlike Tanh (Offset 0) or Tube (Offset 200), which rely on sigmoidal curves to compress the signal, the **300+ Series** uses **Chebyshev Polynomials**. This is a mathematical approach to saturation that transforms the input signal into specific harmonic intervals.

The Physics of Chebyshev Filtering

When used inside a resonant filter loop, Chebyshev saturation doesn't just "squash" the peak; it creates a **Harmonic Lock**:

- **Mode 300 (2nd Order):** The filter resonance generates a tone exactly one octave above the cutoff frequency. This adds a massive, "dual-oscillator" feel to a single source.
- **Mode 400 (3rd Order):** The resonance creates a perfect fifth (plus an octave). This is the secret for "hollow" or "nasal" lead sounds that cut through a mix without needing high-gain distortion.
- **Mode 500 (4th Order):** Generates two octaves above the cutoff. It creates a crystalline, "chime-like" resonance that feels more like an exciter than a filter.

Sonic Characteristics: "Resonance Shaping"

1. **Non-Compressive Peak:** Standard saturation modes lower the resonance peak as you increase drive. Chebyshev modes maintain the "whistle" of the resonance but change its pitch and character.
2. **Dynamic Intervals:** Because the polynomial is applied to the feedback loop, the intensity of the harmonic depends on the input level. Struck sounds (like a `PLUCK`) will have a bright harmonic "ping" that decays into a pure tone.
3. **No Low-End Loss:** While Tanh can sometimes soften the low-end "thump" due to compression, Chebyshev modes keep the fundamental frequency transients intact while adding "sheen" on top.

The Chebyshev Harmonic Blending (CHB)

Unlike standard saturation that clips the signal at the output, the **Crescendo** engine implements **Chebyshev Harmonic Blending** directly within the filter's feedback path. This unique approach allows you to "re-voice" the filter's resonance by injecting specific even or odd harmonics (2nd, 3rd, or 4th order) into the loop. Using directly the Chebyshev polynomials into the filter loop causes instability. An intelligent blend between the original signal, multiplied by the drive, is employed:

- **Dynamic Morphing:** As the **Drive** control increases, the feedback loop seamlessly transitions from a linear signal to a pure Chebyshev polynomial. This transforms the filter from a transparent, surgical tool into a rich, growling harmonic exciter.
- **Acoustic Stability:** The blending is intelligently scaled based on the filter slope. While the **12dB mode** allows for aggressive, raw saturation, the **48dB mode** employs a refined blend ratio to maintain structural integrity and prevent digital clipping, ensuring a silky-smooth analog warmth even at extreme resonance settings.
- **Harmonic Focus:**
 - **Cheby 2:** Adds "tubey" even-order warmth and sub-octave thickening.
 - **Cheby 3:** Delivers a classic "tape-like" odd-order bite, perfect for aggressive lead sounds.
 - **Cheby 4:** Provides complex, modern textures with a dense harmonic spread that cuts through any mix.

Implementing the MOOGG Engine: Cascaded SVF & One-Pole Architectures

The **MOOGG** function combines two distinct filter topologies to cover the full spectrum of analog-style filtering, from subtle tone-shaping to aggressive resonant sweeps.

1. The 6dB/oct Module (Modes -1, -2)

For gentle slopes, the engine utilizes a **Non-Linear One-Pole Topology**. This module calculates a frequency-dependent α coefficient to provide a smooth, 1st-order response.

- **Integrated Saturation:** Unlike linear filters, the input signal is saturated *before* entering the state-register (`state[0]`). This creates a "warm" threshold where the low-pass and high-pass outputs react dynamically to the `drive` level.
- **Phase Integrity:** The High-Pass mode (-2) is derived by subtracting the Low-Pass state from the saturated input, ensuring perfect phase reconstruction and a natural roll-off.

2. The Cascaded SVF Module (Modes 0 to 19)

For steeper slopes (12dB to 48dB), the engine employs a **Cascaded State-Variable Filter (SVF)** architecture, inspired by the research of **Antti Huovilainen** and **Vadim Zavalishin**.

- **Internal Stage Saturation:** Each 12dB section is processed sequentially. The saturation function (`sat`) is embedded within the state integration path of each stage. This emulates the inter-stage clipping of classic analog hardware, where resonance and drive interact to "shape" the filter's growl.
- **Dephase Mode (LP-HP):** In the Dephase modes (16-19), the engine utilizes the complex phase relationships between the Low-pass and High-pass nodes. By subtracting these nodes, it creates unique, frequency-dependent phase cancellations. When pushed with high Drive and Resonance, this produces "liquid" phaser-like textures and vocal formants that are not possible with standard linear filters.

3. Stability & Precision

Across all modes, the algorithm maintains stability through:

- **Sine-Compensated Mapping:** Frequency is mapped using `sinf` to preserve tuning accuracy near the Nyquist limit.
- **Dynamic Damping:** The damp parameter is automatically constrained to prevent digital "blow-up" at extreme resonance settings.
- **Iterative Chaining:** By chaining up to four SVF sections, the filter achieves 24, 36, or 48dB/octave slopes with the characteristic "weight" and harmonic density of a professional-grade Virtual Analog filter.

A final note: the linear filters (FILT and FILT0) employ the same algorithms, but without the non linearities and the drive parameter.

Implementation Tips for Users

- **Use Tanh (Offset 0)** for classic subtractive synthesis and liquid, vintage feels.
- **Use Hard Sat (Offset 100)** for aggressive, industrial grit and "destroyed" resonance.
- **Use Tube (Offset 200)** for warming up thin sounds with asymmetric, fat harmonics.
- **Use Chebyshev (Offsets 300-500)** when you want the filter to act as an **harmonic generator**. Ideal for "modern" sound design, futuristic basslines, and bell-like resonant sweeps.
- **For Bass:** Use **Code 1 (24dB)**. It has the perfect balance of resonance "bite" and low-end retention.
- **For Pads:** Use **Code 0 (12dB)**. It is smoother and allows the high-end air to breathe through even when the filter is partially closed.
- **For Special FX:** Use **Code 11 (48dB BP)**. The two resonance peaks from the dual-ladder architecture create a unique "vocal" quality that sounds like a robotic "Wah."
- **For Mastering/Warming:** Use **Code -1 (6dB LP)** with a drive of 1.2. It acts as a subtle "Tilt EQ" with analog saturation that glues the sound together without sounding "filtered."

Examples:

Classic 24dB Moog Low-Pass:

```
OUT = MOOGG(1, OUT, 1000, 0.7, 2.0)           // Base 0 + Offset 1
```

Industrial 48dB Hard-Clipping Low-Pass:

```
OUT = MOOGG(103, OUT, 500, 0.8, 1.2)           // Base 3 + Offset 100
```

Warm 12dB Tube-Style Band-Pass:

```
OUT = MOOGG(208, OUT, 1500, 0.5, 3.0)           // Base 8 + Offset 200
```

Harmonic "Octave" 24dB Low-Pass:

```
OUT = MOOGG(301, OUT, 800, 0.8, 1.5)           // Base 1 + Offset 300
```

Acid "Fifth" 12dB Band-Pass:

```
OUT = MOOGG(408, OUT, 1200, 0.9, 2.0)           // Base 8 + Offset 400
```

The "Acid Bass" (TB-303 Style):

The secret to the "Acid" sound is a steep filter, high resonance, and a dynamic envelope that drives the internal saturation. By modulating the drive with the envelope, the "grit" of the saturation changes alongside the frequency. This mimics how analog circuits behave when hit with a strong control voltage, making the filter "quack" more realistically.

```
E = ENV0(0.1, 0.2, 0.5, 0.5) // Quick decay envelope
CUTOFF = 400 + (E * 2000)      // Modulate cutoff with envelope
DRV = 1 + E * .5               // Drive increases on the attack!
OUT = MOOGG(1, GAIN * SUPERSAW0(0.05, 0.5), CUTOFF, .85, DRV)
```

Sequential Filtering:

You can "stack" filters for incredible textures. For example, a **Notch** followed by a **Low Pass** creates a classic "Space-Phaser" lead:

```
TEMP = MOOGG(12, IN, 2000, 0.5, 1.0)
OUT = MOOGG(1, TEMP, 1000, 0.3, 1.2)
```

MOOG1(<input>,<Fc>,<drive>)

6 Db/oct low pass MOOG filter simulation, on sample <input> and cutoff frequency <Fc>.

All the signals inside the simulation are multiplied for <drive> before being used in the non-linear function used to simulate the MOOG filter, that is hyperbolic tangent.

The values of <Fc> and <drive> are continuously sampled, can be stereo and are applied separately for both channels. They should be both strictly greater than zero but no check is made.

Example: `OUT = MOOG1(Source, 1000, 2.0)` applies a gentle 6dB slope with significant saturation.

MOOG4F(<input>,<Fc>,<res>,<drive>)

4 stage (24 Db/oct) low pass MOOG filter simulation, on sample <input>, cutoff frequency <Fc> and feedback/resonance <res>.

All the signals inside the simulation are multiplied for <drive> before being used in the non-linear

function used to simulate the MOOG filter, that is hyperbolic tangent.

The values of <Fc>, <drive> and <res> are continuously sampled, can be stereo and are applied separately for both channels. <res> should be between 0 and 1 inclusive and the others should be both strictly greater than zero but no check is made.

Practical Example: Saturating a Lead Sound

```
LAYER
// Source audio with high harmonic content
O = GAIN * OSCG("y1f0000", 1, 0.1) // Square wave

// Apply the 24dB Moog simulation
// Fc: 1500Hz, Res: 0.8, Drive: 5.0 (heavy saturation)
OUT = MOOG4F(O, 1500, 0.8, 5.0)
```

The high drive value introduces non-linear distortion characteristic of overdriven analog filters.

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), 3, 4 and 5 mean a 4, 6 and 8 pole filter. The resonance is automatically calculated based on the order of the filter, to have maximum flatness. Use EQ3DB2 to be able to set also the resonance.

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.

Example: Real-Time Harmonic Shaping In this scenario, a 3-band equalizer is used to "scoop" the mids of a sawtooth lead to make room for other instruments in a mix.

```
LAYER
// Generate source audio
O = GAIN * OSCG("y1f0000", 2, 0.5)

// Apply 12dB/oct (type 2) EQ
// Low band (<300Hz): +4dB boost
// Mid band (300Hz-3000Hz): -8dB cut
// High band (>3000Hz): +2dB boost
OUT = EQ3DB(2, O, 300, 3000, 4, -8, 2)
```

EQ3DB2(<type>, <input>, <lowfreq>, <highfreq>, <lowgain>, <midgain>, <highgain>, <reso>)

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 <reso>. 3, 4 and 5 mean a 4, 6 and 8 pole filter, with resonance <reso>

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.

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 $(\text{DB2LIN}(\text{<gain>}) - 1)$ 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.

Example: Real-Time Presence Boost This example adds "air" to a pad by boosting high-mid frequencies.

```
LAYER
O = GAIN * OSCG("s1f0000")
// Boost the 4000Hz-8000Hz range by 10dB using 12dB/oct filters (type 2)
OUT = EQ1DB(2, O, 4000, 8000, 10, 0.7, 0.7)
```

BIQUADEQDB(<order>, <input>, <fmin>, <fmax>, <gain>)

Syntax: result = BIQUADEQDB(order, input, fmin, fmax, gain)

Description: BIQUADEQDB is a high-precision, multi-mode equalizer based on **Biquad Direct Form I** filters. It is designed for transparent audio processing and is mathematically idempotent, making it ideal for pre-emphasis/de-emphasis chains (e.g., noise reduction or "Reverse Dolby" techniques).

Unlike the standard EQ1DB, this function utilizes serial processing and does not introduce the phase interference typical of parallel summing equalizers.

Parameters:

- **order:** The number of filter stages in cascade (from **1 to 8**). Increasing the order steepens the filter slope (approx. 6dB/octave per stage). The total **gain** is automatically distributed across all stages to ensure numerical stability.
- **input:** The audio sample to be processed.
- **fmin / fmax:** Frequency boundaries in Hz. These parameters determine the filter mode:
 - **Low Shelf:** Triggered if **fmin** < 10 Hz. The filter acts on **fmax**.
 - **High Shelf:** Triggered if **fmax** > 24000 Hz. The filter acts on **fmin**.
 - **Peaking (Bell):** Triggered if both frequencies are within range. The filter centers on the geometric mean of **fmin** and **fmax**, with the **Q factor** automatically calculated based on the bandwidth (in octaves) between the two values.
- **gain:** The boost or attenuation in **decibels (dB)**.

Technical Notes:

- **Efficiency:** Filter coefficients are only recalculated when a parameter changes, minimizing CPU load on real-time automations.
- **Phase Linearization:** When used in a mirror configuration (e.g., -12dB followed by +12dB with identical frequencies), the original signal is reconstructed with near-bit-perfect accuracy.

Example: Surgical High-Frequency Restoration

This example attenuates high frequencies by 12dB using a 2nd-order High Shelf before a noise reduction stage, then restores them afterward.

```
// 1. Pre-processing: Darken the signal to assist NLM
OUT = BIQUADEQDB(2, OUT, 4000, 25000, -12)

OUT = NLMH (OUT, 1, 300, 10)

// 2. Post-processing: Restore original brightness exactly
OUT = BIQUADEQDB(2, OUT, 4000, 25000, 12)
```

Advanced Signal Processing and Effects

The true power of the Crescendo engine lies in its ability to transcend basic sound generation and enter the realm of professional-grade signal conditioning and spatial design. This section details the comprehensive suite of **Effect Functions**—a modular toolkit designed to sculpt the dynamics, frequency response, and spatial characteristics of your instruments and audio streams.

Whether you are working within a **LAYER** to define the character of a single note or in the **POST** section for global mix processing, Crescendo's effects are built for both high-performance efficiency and extreme flexibility. The architecture covers five primary domains:

1. **Temporal & Spatial Processing:** A deep family of **Delays and Echoes** ranging from simple linear interpolating lines to complex, prime-number-based **Reverb** networks. These tools allow you to simulate everything from tight metallic "slapbacks" to infinite, lush atmospheric spaces.
2. **Dynamics Control:** High-precision stereo **Compressors, Expanders, and Noise Gates**. These functions utilize variable attack and decay ramps to manage signal levels, glue mixes together, or surgically remove background noise. Another type of filter that can be categorized as a noise gate is the **Non Local Means (NLM)** filter. See below for details.
3. **Pitch & Frequency Manipulation:** Real-time **Pitch Shifters** (`PSHIFT`) that allow for octave shifts and detuning without altering playback speed, as well as a 32-stage **Phaser** for classic swirling spectral movement.
4. **Modulation Mastery:** An evolution of high-density engines including **Chorus, Flanger, and the hybrid CHOFLA series**. These functions support up to 8 voices per channel and introduce "Asynchronous Modulation," allowing the left and right stereo fields to breathe and drift independently.
5. **Non-Linear Geometry:** Advanced operators like `CHOFLA2` and `CHOFLA3` introduce **WAVE_PARAM** control, giving you the power to morph the actual geometry of LFOs—warping sines into "shelves" or triangles into "saw-slopes" for non-linear, "liquid" textures.

Every effect in this section is **sample-accurate** and features **continuously automatable parameters**. By utilizing the `JOIN()` function, most of these processors can be decoupled into dual-mono units, allowing for asymmetrical stereo sound design that is impossible in standard "hard-wired" VSTs.

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 (because the internal delays use prime numbers, that are integer).*

DELAY(<input>,<DELAY>)

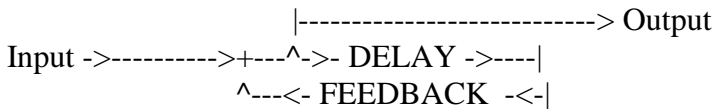
Input ->----- DELAY -----> Output

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 MAXDELAY seconds (See MAXDLY instruction above).

DELAYF(<input>,<DELAY>,<FEEDBACK>)

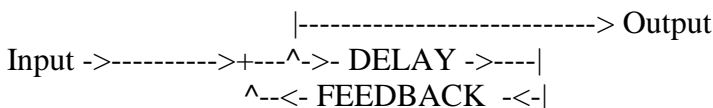


Signal <input> delayed <DELAY> seconds, with feedback signal multiplied by <FEEDBACK>. <DELAY> and <FEEDBACK> can be stereo and are continuously sampled.

<DELAY> is in seconds or in BARS (if < 0, converted in seconds) and is clipped between 0 and MAXDELAY seconds (See MAXDLY instruction above).

<FEEDBACK> should be between -1 and 1.

DELAYFF(<input>,<DELAY>,<FEEDBACK>,<FC>)



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 MAXDELAY seconds (See MAXDLY instruction above).

<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>)



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 MAXDELAY seconds (See MAXDLY instruction above).

<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 MAXDELAY seconds (See MAXDLY instruction above).

<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 MAXDELAY seconds (See MAXDLY instruction above).

<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 MAXDELAY seconds (See MAXDLY instruction above).
 <FEEDLF> and <FEEDHF> should be between -1 and 1.
 The difference with **DELAYFF3** is the point where the output signal is taken.

DELAYFF5(<input>,<DELAY>,<FEEDLF>,<FEEDMF> <FEEDHF>,<FCLO>,<FCHI>)

Input \rightarrow ----- \rightarrow + ----- \rightarrow DELAY \rightarrow - | ----- \rightarrow Output
 $\quad \quad \quad \wedge$ - < - FEEDBACK - < - |

Signal <input> delayed <DELAY> seconds with feedback multiplied by <FEEDLF> for low frequencies, <FEEDMF> for mid frequencies and <FEEDHF> for high frequencies, with <FCLO> and <FCHI> as the cutoff of the filters that separates the bands (6dB/oct). <DELAY>, <FEEDLF>, <FEEDMF>, <FEEDHF>, <FCLO> and <FCHI> 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 MAXDELAY seconds (See MAXDLY instruction above).

<FEEDLF>, <FEEDMF> and <FEEDHF> should be between -1 and 1.

DELAYFF5D(<input>,<DELAY>,<FEEDLF>,<FEEDMF>,<FEEDHF>,<FCLO>,<FCHI>,<DRIVE>)

Input ->----->+----->- DELAY ->-|----->- Output
 ^<- FEEDBACK <-|

Signal <input> delayed <DELAY> seconds with feedback multiplied by <FEEDLF> for low frequencies, <FEEDMF> for mid frequencies and <FEEDHF> for high frequencies, with <FCLO> and <FCHI> as the cutoff of the filters that separates the bands (6dB/oct). The feedback is then saturated with a TANH saturation with the given DRIVE.

<DELAY>, <FEEDLF>, <FEEDMF>, <FEEDHF>, <FCLO>, <FCHI> and <DRIVE> 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 MAXDELAY seconds (See MAXDLY instruction above).

<FEEDLF>, <FEEDMF> and <FEEDHF> should be between -1 and 1.

DELAYFF4X(<input>,<DELAY>,<FEEDLF>,<FEEDHF>,<FC>)

Input \rightarrow ----- \rightarrow + ----- \rightarrow DELAY \rightarrow -|----- \rightarrow Output
 \wedge -<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 MAXDELAY seconds (See MAXDLY instruction above).

<FEEDLF> and <FEEDHF> should be between -1 and 1.

PPDELAY(<input>,<DELAY>,<FEEDLF>,<FEEDHF>,<FC>)



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.

The latter 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.

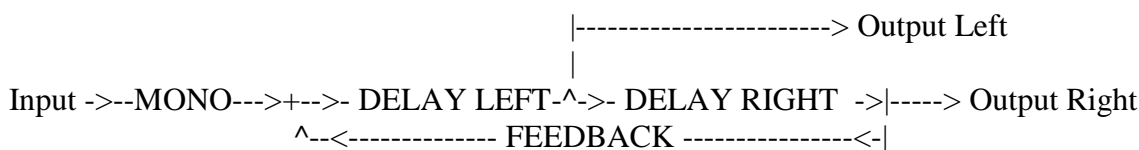
<DELAY> is in seconds or in BARS (if < 0, converted in seconds) and is clipped between 0 and MAXDELAY seconds (See MAXDLY instruction above).

<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).

PPDELAY2(<input>,<DELAY>,<FEEDLF>,<FEEDMF> <FEEDHF>,<FCLO>,<FCHI>)



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 three filters of 6dB/oct.

The LP, MID and HP components are given the gain <FEEDLF>, <FEEDMF> and <FEEDHF> respectively and then fed back as feedback. (only the LEFT value, since this signal is mono).

<FCLO> and <FCHI> are mono: only the L component is used.

<DELAY>, <FEEDLF>, <FEEDMF>, <FEEDHF>, <FCLO> and <FCHI> 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 MAXDELAY seconds (See MAXDLY instruction above).

<FEEDLF>, <FEEDMF> 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).

PPDELAY2D(<input>,<DELAY>, <FEEDLF>, <FEEDMF>, <FEEDHF>, <FCLO>, <FCHI>, <DRIVE>)



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 three filters of 6dB/oct.

The LP, MID and HP components are given the gain <FEEDLF>, <FEEDMF> and <FEEDHF> respectively and then fed back as feedback. (only the LEFT value, since this signal is mono).

<FCLO> and <FCHI> are mono: only the L component is used. The feedback is then saturated with a TANH saturation with the given DRIVE.

<DELAY>, <FEEDLF>, <FEEDMF>, <FEEDHF>, <FCLO>, <FCHI> and <DRIVE> 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 MAXDELAY seconds (See MAXDLY instruction above).

<FEEDLF>, <FEEDMF> 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).

Recap and examples: Implementing Delay and Echo Functions

The Crescendo engine provides a versatile family of delay functions designed to create time-based effects such as simple echoes, chorusing, flanging, and complex reverbs. These functions are

typically utilized in the **POST** section for global effects processing but can also be implemented within individual **LAYER** sections for note-specific sound design.

1. Technical Overview: Interpolating vs. Non-Interpolating

Crescendo offers two variants for every delay function:

- **Standard Delay (`DELAY`, `DELAYF`, etc.):** Utilizes linear interpolation. These are required for time-varying effects (like a chorus where delay time is modulated by an LFO) to prevent audible pitch artifacts.
- **Non-Interpolating Delay (`NIDELAY`, `NIDELAYF`, etc.):** Faster and more CPU-efficient. These are ideal for fixed delays or reverb simulations where delay times do not change rapidly.

Important Memory Requirement: Before using long delays, you must use the `MAXDLY` instruction in the **COMMON** section to pre-allocate the necessary memory buffer. By default, Crescendo supports approximately 40 seconds of delay.

2. Basic Delay Functions

Example A: Simple Echo (`DELAY`) `DELAY(<input>, <DELAY>)` This function simply delays the input signal by a specified time in seconds.

```
// Delay the layer output by 0.5 seconds
Echo = DELAY(OUT, 0.5)
OUT = OUT + 0.3 * Echo
```

Example B: Delay with Feedback (`DELAYF`) `DELAYF(<input>, <DELAY>, <FEEDBACK>)` This function routes a portion of the delayed signal back into the input, creating a repeating echo that fades over time.

```
// 1/4 bar delay with 70% feedback
// Note: negative values for time are interpreted as rhythmic BARS
OUT = DELAYF(OUT, -0.25, 0.7)
```

3. Advanced Filtered and Multi-Tap Delays

Example C: Filtered Feedback Delay (`DELAYFF`) `DELAYFF(<input>, <DELAY>, <FEEDBACK>, <FC>)` Applying a filter to the feedback loop creates "warmer" or "darker" echoes by removing high or low frequencies with each repetition.

- **Positive `<FC>`:** Low-Pass Filter (removes highs).
- **Negative `<FC>`:** High-Pass Filter (removes lows).

```
// Echo with a 2000Hz Low-Pass filter on the feedback loop
OUT = DELAYFF(OUT, 0.3, 0.8, 2000)
```

Example D: Frequency-Split Feedback (`DELAYFF3` / `DELAYFF4`) `DELAYFF4(<input>, <DELAY>, <FEEDLF>, <FEEDHF>, <FC>)` This advanced variant allows independent feedback control for low-frequency and high-frequency components of the signal.

```
// High-frequency echoes fade quickly (0.2), low-frequency echoes last longer (0.9)
// Crossover frequency at 1000Hz
```

```
OUT = DELAYFF4(OUT, 0.4, 0.9, 0.2, 1000)
```

4. Stereo Imaging Delays

Example E: Ping Pong Delay (PPDELAY) PPDELAY(<input>, <DELAY>, <FEEDLF>, <FEEDHF>, <FC>) This function bounces the echo between the left and right speakers. It automatically converts the input to mono before processing the stereo taps.

```
POST
// Rhythmic ping-pong delay using VST Variable #4 (Index 604) for time
// Variables 601, 602, and 603 control feedback and frequency crossover
OUT = PPDELAY(OUT, -MCC(604), MCC(601), MCC(602), MCC(603))
```

Example F: Stereo Cross-Feedback (DELAYFF4X) DELAYFF4X(<input>, <DELAY>, <FEEDLF>, <FEEDHF>, <FC>) This function exchanges the left and right channels within the feedback loop to enhance the stereo width of the effect.

5. Technical Constraints Summary

| Parameter | Units / Range | Note |
|------------|----------------|---|
| <input> | Audio Signal | Can be mono or stereo. |
| <DELAY> | Seconds / BARS | Positive = Seconds; Negative = BARS (tempo-synced). |
| <FEEDBACK> | -1.0 to +1.0 | Feedback amount; values near 1.0 approach infinite decay. |
| <FC> | Hertz | Cutoff frequency for the internal 6dB/oct filter. |
| MAXDLY | 0.01 to 1M sec | Pre-allocates delay buffers in RAM. |

Crescendo Dynamics & Restoration Suite

The Crescendo Dynamics Suite is more than a collection of audio processors; it is a high-precision toolkit designed for the surgical restoration and enhancement of audio signals.

At the heart of this suite lies the **NLM (Non-Local Means)** module, a breakthrough in audio de-noising logic.

Most noise reduction systems rely on local spectral subtraction (FFT) or simple gates, which often introduce "musical noise" or strip away the natural harmonics of the recording. Our approach is different.

By adapting **Non-Local statistical estimation**—a technology primarily used in **2D and 3D medical imaging (MRI/CT scans)** for high-resolution tissue analysis—we treat audio as a 1D high-fidelity image.

The core innovation is the **Hyperbolic Kernel**. While traditional NLM implementations use Gaussian kernels (Buades et al.), Crescendo utilizes a **1/d response (Hyperbolic)**. This provides a more "organic" and fluid response, avoiding the "dead air" feeling of digital gates and maintaining the natural "air" and spatial reverb of the original recording.

1. Restoration: NLM & NLMH Functions

The NLM functions identify and suppress incoherent noise (hiss, hum, and static) by comparing the current audio snippet with its own history.

Technical Syntax

```
result = NLM(input, strength, block_size, backward)
```

```
result = NLMH(input, strength, block_size, window)
```

- **NLM (Zero Latency):** Best for live monitoring and real-time interactive restoration. It uses only past and current samples to determine weights.
- **NLMH (Look-Ahead):** Best for offline rendering and maximum quality. It scans both past and future samples, providing a more symmetric and stable mean. If you can live with a slight delay, you can use it also for live performance: the latency is the sum of block size and window in samples. With the typical values used in practical filters, we are talking of a few hundred samples.

Parameters & Tuning

- **Strength (H) [useful range 0.5 – 10.0]:** Controls aggressiveness.
 - 0.5 – 1.5: Maximum transparency, preserves vocal "shimmer."
 - 2.0 – 4.0: The "sweet spot" for vintage masters.
- **Block Size [useful range 100 – 500]:** The "fingerprint" length. Use 200+ for stable results; 400+ for "Monster" precision in offline mode. Over 400 there is diminishing returns.
- **Window / Backward [useful range 5 – 50]:** Defines the search range. Larger windows allow noise to "average out" toward zero more effectively, but there is risk of getting signal not correlated. Best results, also for offline processing, is in the 5 – 20 range. Over 20 there is diminishing returns.

2. Noise Gate Functions: SNG and SNG2

Selective attenuation based on amplitude to silence background noise during quiet passages.

- **SNG:** Uses a single cutoff frequency (`ABS_FC`) to smooth the detected signal envelope.
- **SNG2:** Independent control over `attack` and `decay` times, ideal for percussive instruments.

3. Compression and Expansion: COMP and COMPNG

Modifies dynamic range by scaling signals above or below a threshold.

- **COMP:** Standard peak compression. `amount > 1.0` creates a compressor; `< 1.0` creates an expander.
- **COMPNG:** Focuses on the low portion of the signal, acting as a "soft" noise gate through downward expansion.

4. Advanced Dynamics: COMP2

A bilateral processor for simultaneous high-level compression and low-level gating.

- **Logic:** Signals below `thres1` are gated/expanded via `slope`; signals above `thres2` are compressed/expanded via `amount`.

NLM(<input>, <strength>, <block_size>, <backward>)

OR

NLMH(<input>, <strength>, <block_size>, <window>)

The **NLM** and **NLMH** functions are sophisticated de-noising algorithms based on non-local statistical estimation. Unlike standard noise gates or spectral subtractors (FFT), these modules preserve the harmonic integrity and "warmth" of the original signal by comparing the current audio snippet with its own history to identify and suppress incoherent noise (hiss, hum, and static).

This algorithm is a 1D adaptation of **Non-Local Means (NLM)**, a technique heavily utilized in medical image processing (MRI/CT).

As a researcher in the medical field, I have implemented these high-precision logic systems for 2D and 3D medical images; here, they are optimized for high-fidelity audio restoration, treated as an 1D image.

The Hyperbolic Breakthrough:

Unlike traditional NLM which uses Gaussian kernels (Buades et al.), Crescendo uses a **Hyperbolic Kernel (1/d response)**. This provides a more "organic" and fluid response, avoiding the "dead air" feeling of digital gates and maintaining the natural air of the recording.

1. Core Functions

```
result = NLM(input, H, Block_Size, Backward)
```

- **Mode:** Zero Latency.
- **Best For:** Live monitoring and real-time interactive restoration.
- **Logic:** Uses only past and current samples to determine weights.
- **Parameters:**
 - **Strength (H) [useful range 0.5 – 10.0]:** Controls aggressiveness.
 - *0 – 0.5:* Just slight denoise, but often sufficient for low noise signal.
 - *0.5 – 1.5:* Maximum transparency, preserves vocal "shimmer."
 - *2.0 – 4.0:* The "sweet spot" for vintage masters.
 - *> 4.0:* Use only for extremely damaged recordings. May result in a darker tone.

- **Block Size [useful range 40 – 1000]:** The "fingerprint" length. Use 200+ for stable results; 400+ for "Monster" precision in offline mode. Over 200 there is diminishing returns. Maximum is 2048.
- **Backward [useful range 10 – 100]:** Defines the search range. Larger windows allow noise to "average out" toward zero more effectively, but there is risk of getting signal not correlated. Best results, also for offline processing, is in the 10 – 40 range. Over 40 there is diminishing returns. Maximum is $32767 - \text{Block_Size}$.

```
result = NLMH(input, H, Block_Size, Window)
```

- **Mode:** Look-Ahead (Bidirectional).
- **Best For:** Offline rendering and maximum quality restoration.
- **Logic:** Scans both past and future samples, providing a more symmetric and stable mean.
- **Parameters:**
 - **Strength (H) [useful range 0.5 – 10.0]:** Controls aggressiveness.
 - 0 – 0.5: Just slight denoise, but often sufficient for low noise signal.
 - 0.5 – 1.5: Maximum transparency, preserves vocal "shimmer."
 - 2.0 – 4.0: The "sweet spot" for vintage masters.
 - > 4.0: Use only for extremely damaged recordings. May result in a darker tone.
 - **Block Size [useful range 20 – 500]:** The "fingerprint" length. Use 100+ for stable results; 200+ for "Monster" precision in offline mode. Over 200 there is diminishing returns. Maximum is 2048.
 - **Window [useful range 5 – 50]:** Defines the total search range (look-back and look-ahead). Larger windows allow noise to "average out" toward zero more effectively, but there is risk of getting signal not correlated. Best results, also for offline processing, is in the 5 – 20 range. Over 20 there is diminishing returns. Maximum is $16382 - 2 * \text{Block_Size}$.

2. Post-Processing & EQ

To compensate for "dull" or aged masters, use the **BIQUADEQDB** function after the NLM stage:

- **Filter Type:** High Shelf.
- **Order:** **Order 1** is highly recommended. It provides a natural, musical slope (6 dB/oct) that "re-opens" the sound without phase harshness.
- **Frequency:** Start around **1000 Hz – 5000 Hz** to bring forward the "presence" and "air" of the recording.
- **Gain:** Because NLM removes the noise floor, you can push the gain significantly (even **+12 to +24 dB**) to reveal hidden details like the "click" of nails on guitar strings.

3. Practical Tips for Restoring Vintage Audio

The "Church" Effect: If the recording sounds like it was in a spacious room (reverb), keep Strength low (<1.0). High Strength settings tend to treat reverb tails as noise and will "shrink" the acoustic space.

- **Multi-Stage Filtering:** For better results, try two stages of NLM with very low Strength (0.5 each) instead of one single heavy pass.
- **Headroom:** When applying high Gain in the post-filter (+24 dB), ensure you reduce the input gain to avoid **digital clipping**.

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 $2 * \text{<thres>}$, 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 $2 * \text{<thres>}$ 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 $2 * \text{<thres>}$.

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 $2 * \text{<thres>}$, 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 $2 * \text{<thres>}$ 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 $\langle A \rangle$ is below $\langle \text{thres} \rangle$, this factor is 1. If $\langle A \rangle$ is above or equal $\langle \text{thres} \rangle$, this factor is $(\langle \text{thres} \rangle + (\langle A \rangle - \langle \text{thres} \rangle) / \langle \text{amount} \rangle) / \langle A \rangle$.

The $\langle \text{input} \rangle$ signal is multiplied by this factor (for each channel) and the result is the output of the function.

If $\langle \text{amount} \rangle > 1$ then the function acts as a compressor.

If $\langle \text{amount} \rangle < 1$ acts as an expander.

No check is made for $\langle \text{amount} \rangle$ to be > 0 .

COMP2($\langle \text{input} \rangle, \langle \text{thres1} \rangle, \langle \text{slope} \rangle, \langle \text{thres2} \rangle, \langle \text{amount} \rangle, \langle \text{atk} \rangle, \langle \text{dcy} \rangle$)

Stereo bilateral compression (or expansion) of input signal, with $\langle \text{thres1} \rangle$ and $\langle \text{thres2} \rangle$ 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 $\langle A \rangle$.

The initial state is "attack", so the ramp time is $\langle \text{atk} \rangle$ seconds. This state remains until $\langle A \rangle$ is above the absolute value of the input signal.

At this point the state is switched to "decay" and the ramp time is $\langle \text{dcy} \rangle$ seconds ($\langle \text{atk} \rangle$ and $\langle \text{dcy} \rangle$ are stereo and continuously automatable).

The state remains "decay" until $\langle A \rangle$ goes below the absolute value of the input signal.

For each channel a multiplicative factor is calculated:

- if $\langle A \rangle$ is below $\langle \text{thres1} \rangle$, this factor is $\max(0, (\langle \text{thres1} \rangle + (\langle A \rangle - \langle \text{thres1} \rangle) * \langle \text{slope} \rangle) / \langle A \rangle)$.
- if $\langle A \rangle$ is between $\langle \text{thres1} \rangle$ and $\langle \text{thres2} \rangle$, extremes included, this factor is 1.
- If $\langle A \rangle$ is above $\langle \text{thres2} \rangle$, this factor is $(\langle \text{thres2} \rangle + (\langle A \rangle - \langle \text{thres2} \rangle) / \langle \text{amount} \rangle) / \langle A \rangle$.

The factor is limited to 1000 (+60Db) and care is taken to avoid NaNs.

The $\langle \text{input} \rangle$ signal is multiplied by this factor (for each channel) and the result is the output of the function.

If $\langle \text{amount} \rangle > 1$ then the function acts as a compressor on high signal levels.

If $\langle \text{amount} \rangle < 1$ acts as an expander.

No check is made for $\langle \text{amount} \rangle$ to be > 0 .

If $\langle \text{slope} \rangle > 1$ then the function acts as a compressor on low signal levels (soft/hard noise gate).

If $\langle \text{slope} \rangle < 1$ acts as an expander.

No check is made for $\langle \text{slope} \rangle$ to be > 0 .

$\langle \text{thres1} \rangle$ should be greater than 0 and $\langle \text{thres2} \rangle$ greater than $\langle \text{thres1} \rangle$ but no check is made.

COMPNG($\langle \text{input} \rangle, \langle \text{thres} \rangle, \langle \text{slope} \rangle, \langle \text{atk} \rangle, \langle \text{dcy} \rangle$)

Stereo compression (or expansion) of low portion of input signal, with $\langle \text{thres} \rangle$ 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 $\langle A \rangle$.

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 $\max(0, (\text{<thres>} + (\text{<A>} - \text{<thres>}) * \text{<slope>}) / \text{<A>})$.
- 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 expander.

No check is made for <slope> to be > 0.

Recap and Examples: Dynamics and Signal Level Processing

Crescendo provides a suite of stereo dynamics functions designed to control the amplitude envelope of audio signals. These tools are commonly utilized in the **POST** section for global signal conditioning, such as noise reduction or dynamic range compression, but they are equally effective within individual **LAYER** sections for note-specific processing.

1. Noise Gate Functions: SNG and SNG2

These functions selectively attenuate signals based on their amplitude, effectively silencing background noise during quiet passages.

Example: Basic Signal Cleanup (SNG) The SNG function uses a single cutoff frequency (ABS FC) to smooth the detected signal envelope before applying the gate.

```
POST
// Gate the main input (INZ)
// Threshold: 0.02
// Envelope smoothing filter: 10 Hz
OUT = SNG(INZ, 0.02, 10)
```

If the smoothed absolute signal amplitude is below 0.02, the output is silenced.

Example: Percussive Gating (SNG2) The SNG2 function allows for independent control over the attack and decay times of the gate, making it ideal for drums or instruments with sharp transients.

```
LAYER
// Apply a gate to an oscillator
// Threshold: 0.05
// Attack: 0.005s (fast)
// Decay: 0.2s (smooth fade)
O = GAIN * OSCG("s1f0000")
OUT = SNG2(O, 0.05, 0.005, 0.2)
```

2. Compression and Expansion Functions: COMP and COMPNG

These functions modify the dynamic range by scaling signals that fall above or below a specific threshold.

Example: Standard Peak Compression (COMP) Setting the `amount` parameter to a value greater than 1.0 creates a compressor, while a value less than 1.0 creates an expander.

```
POST
// 4:1 Compressor on the summed output
// Threshold: 0.1
// Amount: 4.0
// Attack: 0.01s, Decay: 0.1s
OUT = COMP(OUT, 0.1, 4.0, 0.01, 0.1)
```

Example: Low-Level Expansion (COMPNG) The `COMPNG` function focuses exclusively on the low portion of the signal, typically used to act as a "soft" noise gate.

```
POST
// Smoothly expand signals below the 0.05 threshold
// Slope: 2.0 (downward expansion)
OUT = COMPNG(INZ, 0.05, 2.0, 0.01, 0.2)
```

3. Advanced Dynamics: COMP2

The `COMP2` function is a bilateral processor that allows for simultaneous high-level compression and low-level gating/expansion within a single instruction.

Technical Syntax: `COMP2(<input>, <thres1>, <slope>, <thres2>, <amount>, <atk>, <dcy>)`

- **<thres1> and <slope>:** Define the behavior for signals at the bottom of the range (noise gating).
- **<thres2> and <amount>:** Define the behavior for signals at the top of the range (compression).

Example: Full Dynamic Range Control In this scenario, signals below 0.01 are gated, while signals above 0.5 are compressed, leaving the middle range untouched.

```
POST
// Gating signals below 0.01 with a 10:1 slope
// Compressing signals above 0.5 with a 4:1 ratio
OUT = COMP2(OUT, 0.01, 10, 0.5, 4, 0.01, 0.1)
```

4. High-Fidelity Vintage Restoration with NLM filter

In this scenario, we first remove the tape hiss with NLM and then re-open the brilliance with a post-filter.

```
POST
// 1. Remove hiss using NLM (Zero Latency)
// Strength: 1.2 (Transparent), Block: 250, Backward: 20
OUT = NLM(OUT, 1.2, 250, 20)

// 2. Re-open the high-end with a 1st order Biquad
```

```
// Because NLM removed the noise floor, we can push +18dB safely
OUT = BIQUADEQDB(OUT, 4500, 1, 18)

// 3. Final safety compression
OUT = COMP(OUT, 0.8, 2.0, 0.01, 0.1)
```

Reverb Engine: Technical Overview

Crescendo's reverberation system is built upon an optimized **Schroeder-style topology**, utilizing a precision-engineered network of **All-Pass filters** (for initial diffusion) followed by a **Reverb Tank** of parallel **Comb filters** (for decaying tails).

The core of the engine relies on the mathematical properties of prime numbers to ensure a natural, inharmonic resonance profile, free from the metallic "ringing" typical of digital delays. Unlike standard VSTs that use fixed values, Crescendo allows you to "tune" the geometry and density of your virtual space.

Signal Flow and Topology

- **Standard Chain:** In all high-fidelity modes, the signal flows through a **Diffusion Network** first, which smears transients to create a smooth onset, and then enters the **Late Reflection Tank** for the final decay.
- **Dual-Stage Mixing:** REVERB4 features independent gain controls for **Early Reflections** (All-Pass output) and **Diffuse Tail** (Comb output). This allows for surgical precision in balancing "clarity" against "wash."

Key Professional Parameters

1. Density (G-Factor)

- **Definition:** The feedback gain for the All-Pass Filters.
- **Sonic Effect:** Determines the density and "thickness" of the sound. This is the heart of the "diffusion" control found in classic hardware.
 - **Low values (< 0.5):** Clear, sparse, and granular.
 - **Golden Ratio (~0.618):** The mathematical "sweet spot" for perfect phase distribution and transparency.
 - **High values (> 0.8):** Extremely dense, eventually becoming metallic and highly resonant.
- **Pro Tip:** Start at **0.618** (the Golden Ratio) for a natural feel. Lower it if you want to hear the individual "grains" of the reverb for an ASMR or lo-fi effect.

2. Room Size (C-Prime / Comb Base)

- **Definition:** Sets the starting index in the internal Prime Number table for the Comb Filter bank.
- **Sonic Effect:** Directly controls the physical scale of the room. Lower values (e.g., Prime #40) simulate small, resonant boxes; higher values (e.g., Prime #200+) create vast, cathedral-like spaces.
- **Pro Tip:** Use lower values for percussive sounds to avoid "laggy" transients, and higher values for ambient pads to achieve a deep sense of immersion.

3. Complexity (C-Step)

- **Definition:** The step (or interval) between the prime numbers chosen for each consecutive Comb Filter. Supports fractional values for fine-tuning.
- **Sonic Effect:** Governs the modal density and "color" of the decay.
 - A step of **1.0** (consecutive primes) can result in a metallic "ringing" due to close resonant frequencies.
 - Higher steps (**3.0 to 7.0**) spread the resonances, creating a smoother, more natural "cloud" of sound.
- **Pro Tip:** If the reverb sounds too "tuned" or pitch-like, increase the **Complexity** to break up harmonic patterns.

4. Initial Reflections (A-Prime / All-Pass Base)

- **Definition:** Sets the starting index in the Prime Number table for the All-Pass Filter bank.
- **Sonic Effect:** Influences the initial density of the reflections. These filters "smear" the incoming signal before it enters the main decay loop. Smaller primes create a tight, grainy attack; larger primes produce a softer, more blurred onset.
- **Pro Tip:** For a "shimmering" effect, keep **A-Prime** significantly smaller than **C-Prime**.

5. Early Diffusion (A-Step)

- **Definition:** The step between prime numbers used for the All-Pass chain.
- **Sonic Effect:** Controls the texture of the transient response. A higher **A-Step** increases the "phase-scrambling" effect, making the initial hit of the reverb feel more diffuse and less "clicky."
- **Pro Tip:** Keep this value between **1.0 and 3.0**. Excessive A-Step values can make the onset of the reverb feel disconnected from the original sound.

6. Stereo Width

- **Definition:** A configurable offset between Left and Right channels.
- **Logic:** Uses **Late-Binding Float Indexing** so the two channels never share the same delay lengths, resulting in a significantly wider stereo image.
- **Pro Tip:** For optimal phase coherence, set **Stereo Width** to about half the value of your **A-Step** and **C-Step**.

7. Diffusion Stages (All-Pass Count)

- **Definition:** Sets the number of cascading All-Pass filters within the Diffusion Network.
- **Sonic Effect:** Determines how many times the signal is "shuffled" and phase-scrambled before entering the main tank.
 - **Lower values (2–4):** Preserve more of the original transient's punch but can result in a "grainy" or "jittery" onset.
 - **Higher values (6–8):** Create a perfectly smooth, liquid-like transition from the dry sound to the reverb tail.
- **Pro Tip:** Use fewer stages for percussive sounds (like drums) to maintain "snap," and maximum stages for pads and vocals to achieve that signature "ghostly" smoothness.

8. Modal Richness (Comb Count)

- **Definition:** Sets the number of parallel delay lines (Comb Filters) inside the Reverb Tank.
- **Sonic Effect:** Directly governs the structural density of the reverb tail.
 - **Lower values:** May exhibit periodic "flutter" or simpler harmonic patterns.
 - **Higher values:** Increase the number of resonant "modes," resulting in a thick, velvety decay that feels like a real physical space.
 - The maximum value is 32 comb filters.
- **Pro Tip:** This is your primary "Quality" control. If the reverb tail sounds too "thin" or "electronic," increase the **Modal Richness** to fill the spectral gaps.

A Note on Performance

CPU vs. Quality: Both **Diffusion Stages** and **Modal Richness** increase the computational load linearly. For mobile or high-polyphony Soundfont use, a **4/8** configuration (4 Diffusion Stages, 8 Modal Richness) is the professional standard for efficiency. For studio-grade "infinite" pads, don't hesitate to push these values higher to unlock the full mathematical potential of the Crescendo Prime-Number engine.

Dual-Stage Output Mixing (Early vs. Diffuse)

The reverb engine features an independent dual-stage mixer at the output:

- **Early Reflections (ER) Level:** Controls the output of the Diffusion Network (All-Pass). Adjust this to increase the initial "body" and presence, simulating the first bounces off nearby surfaces.
NOTE: The **Early Level** in Crescendo isn't just a volume knob; it's a 'Presence Reinforcer'. While 1.0 provides a mathematically transparent response, pushing it towards 2.0 and beyond adds a professional sheen and structural density to the sound, typical of high-end studio chambers.
- **Diffuse Level (Tail):** Controls the output of the Reverb Tank (Comb filters). This dictates the intensity of the long, lush decay.

The "Clarity" Secret: For a transparent, modern sound, keep the **Early Level** prominent while tucking the **Diffuse Level** slightly back. For ambient textures and pads, push the **Diffuse Level** to emphasize the spectral cloud.

Feedback & Spectral Control (Dual-Band Decay)

Crescendo utilizes a split-band feedback architecture to simulate how different environments absorb sound energy.

- **LROOMSIZE (Bass Decay):** Controls the persistence of low-end energy. Values near 1.0 create "infinite" bass sustain (cathedrals/canyons), while lower values tighten the low end to prevent "muddiness."
- **HROOMSIZE (Treble Decay):** Controls the persistence of high-end air and shimmer. This is the key to **ASMR textures**. High values create a crystalline, glassy atmosphere.
- **FC (Crossover Frequency):** The threshold frequency (in Hz) that splits the signal between the Low-Pass and High-Pass feedback loops.

Advanced Topology: Late-Binding Float Interpolation

All indexing parameters are handled as high-precision floating-point values. The final index selection from the Prime Number table is rounded only at the moment of filter initialization.

- **Unique Prime Distribution:** Fractional **Complexity** and **Stereo Width** ensure L and R channels never share delay lengths, eliminating "narrow stereo" overlap.
- **Fractional Spacing:** Non-integer steps create an "irregular grid" of resonances, preventing harmonic build-ups and resulting in a smoother, more organic decay.
- **Stereo Cross-Feedback:** Combined with internal channel swapping, every sound grain eventually traverses every unique prime delay in both channels, doubling the perceived reflection density without increasing CPU overhead.

Technical Note: All internal buffers are fixed to a maximum prime of **8191 samples** (approx. 170ms at 48kHz). For longer pre-delays, a dedicated discrete delay line must be placed before the reverb network.

Advanced Space Design

Unlike traditional reverb algorithms, Crescendo operates on a principle of **Geometric Scale** rather than a simple "Decay Time" knob. This requires a small shift in how you approach sound design:

The Decay Dual-Logic

The duration of the reverb tail is the result of a synergy between two domains:

- **Spectral Conservation (Feedback):** This simulates the "material" of the walls. High values represent reflective surfaces like marble; low values represent absorbent materials like wood.
- **Geometric Scale (Complexity & Room Size):** These parameters define the physical path the sound travels. By increasing **Complexity**, you are effectively moving the virtual walls further apart.

Note: Because the path is longer, the sound takes more time to return, extending the decay naturally. For a lush, modern "Hall" sound, try favoring higher **Complexity** with moderate **Feedback** to achieve a smooth, non-metallic tail.

Presence and Diffusion

With the signal chain employed (**Diffusion** → **Tank**), the **Early Level** acts as a "Presence" reinforcer.

- Boosting the **Early Level** (even up to 2.0 and beyond) adds structural body and "expensive" air directly to the dry signal.
- The **Tail Level** then handles the deep spatial immersion. If the reverb feels "muddy," try lowering the **Tail** and pushing the **Early** reflections: you will retain the instrument's clarity while surrounding it with a sophisticated atmospheric cloud.

Streamlined Versions

The faster, reduced versions of the algorithm share this same high-fidelity DNA but with optimized fixed settings. The value for the missing parameters are specified in the function description.

If not specified otherwise, the output includes the early reflections plus 0.7 times the full tail processing (the Comb filters output). Use REVERB4 to configure also this mixing.

REVERB0(<input>,<ROOMSIZE>)

Fast reverb with 8 parallel DELAYs with feedback (equivalent to DELAYFF4X) and 4 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 of the comb filters is the 120th prime number, about 15ms at 48KHz of sample rate. The other 7 stages take the next prime numbers with C-Step = 2.

The minimum delay of the all pass filters is the 40th prime number. The other 3 stages take the next prime numbers with A-Step = 2.

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 has feedback coefficient 0.8. 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 5000 Hz.

The offset, in primes, applied to right delay on the DELAYFF4X filters is 1, so the the right DELAY starts from the 121st prime.

The <ROOMSIZE> parameter is mono and continuously automatable.

REVERB(<input>,<LPROOM>,<HPROOM>,<FC>,<STEREO>)

Fast reverb with 8 parallel DELAYs with feedback (equivalent to DELAYFF4X) and 4 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 of the comb filters is the 120th prime number, about 15ms at 48KHz of sample rate. The other 7 stages take the next prime numbers with C-Step = 2.

The minimum delay of the all pass filters is the 40th prime number. The other 3 stages take the next prime numbers with A-Step = 2.

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 right delay on the DELAYFF4X filters and the All Pass filters. The value is added to the right delay. E.g. if <STEREO> is 2, the right DELAY starts from the 122nd prime.

The parameters are mono and continuously automatable.

The final indexes are automatically capped between 0 and 1027 (the range of prime indexes).

REVERB1(<input>,<LPROOM>,<HPROOM>,<FC>,<STEREO>,<DELAY_OFFSET>)

Reverb with 8 parallel DELAYs with feedback (equivalent to DELAYFF4X) and 4 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 of the comb filters is the 120th + <DELAY_OFFSET> prime number, about 15ms at 48KHz of sample rate. The other 7 stages take the next prime numbers with C-Step = 2.

The minimum delay of the all pass filters is the 40th prime number. The other 3 stages take the next prime numbers with A-Step = 2.

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 right delay on the DELAYFF4X filters and the All Pass filters. The value is added to the right delay.

The parameters are mono and continuously automatable.

The final indexes are automatically capped between 0 and 1027 (the range of prime indexes).

REVERB2(<input>,<LPROOM>,<HPROOM>,<FC>,<STEREO>,<DELAY_OFFSET>,<G_ALL_PASS>)

Reverb with 8 parallel DELAYs with feedback (equivalent to DELAYFF4X) and 4 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 of the comb filters is the 120^{th} + <DELAY_OFFSET> prime number, about 15ms at 48KHz of sample rate. The other 7 stages take the next prime numbers with C-Step = 2.

The minimum delay of the all pass filters is the 40^{th} prime number. The other 3 stages take the next prime numbers with A-Step = 2.

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 right delay on the DELAYFF4X filters and the All Pass filters. The value is added to the right delay.

The parameters are mono and continuously automatable.

The final indexes are automatically capped between 0 and 1027 (the range of prime indexes).

REVERB3(<input>,<LPROOM>,<HPROOM>,<FC>,<STEREO_WIDTH>,<ROOM_SIZE>,<MODAL_RICHNESS>,<INITIAL_REFLECTIONS>,<DIFFUSION_STAGES>,<DENSITY>)

Reverb with <MODAL_RICHNESS> parallel DELAYs with feedback (equivalent to DELAYFF4X) and <DIFFUSION_STAGES> 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 of the comb filters is the <ROOM_SIZE>th prime number. The other stages take the next prime numbers with C-Step = 2.

The minimum delay of the all pass filters is the <INITIAL_REFLECTIONS>th prime number. The other stages take the next prime numbers with A-Step = 2.

The feedback coefficient of the all pass filters is <DENSITY> (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_WIDTH> is the offset, in primes, applied to right delay on the DELAYFF4X filters and the All Pass filters. The value is added to the right delay.

The parameters are mono and continuously automatable.

The final indexes are automatically capped between 0 and 1027 (the range of prime indexes).

<MODAL_RICHNESS> is capped to 32, <DIFFUSION_STAGES> is capped to 8.

REVERB4(<input>,<LPROOM>,<HPROOM>,<FC>,<STEREO_WIDTH>,<ROOM_SIZE>,<COMPLEXITY>,<MODAL_RICHNESS>,<TAIL_LEVEL>,<INITIAL_REFLECTIONS>,<DIFFUSION>,<DIFFUSION_STAGES>,<DENSITY>,<EARLY_LEVEL>)

Reverb with <MODAL_RICHNESS> parallel DELAYs with feedback (equivalent to DELAYFF4X) and <DIFFUSION_STAGES> 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 of the comb filters is the $\langle \text{ROOM_SIZE} \rangle^{\text{th}}$ prime number. The other stages take the next prime numbers with $\text{C-Step} = \langle \text{COMPLEXITY} \rangle$.

The minimum delay of the all pass filters is the $\langle \text{INITIAL_REFLECTIONS} \rangle^{\text{th}}$ prime number. The other stages take the next prime numbers with $\text{A-Step} = \langle \text{DIFFUSION} \rangle$.

The feedback coefficient of the all pass filters is $\langle \text{DENSITY} \rangle$ (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.

$\langle \text{LPROOM} \rangle$ and $\langle \text{HPROOM} \rangle$ 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.

$\langle \text{FC} \rangle$ is the cutoff frequency of the 6dB/oct filter used to separate the frequencies.

$\langle \text{STEREO_WIDTH} \rangle$ is the offset, in primes, applied to right delay on the DELAYFF4X filters and the All Pass filters. The value is added to the right delay.

The parameters are mono and continuously automatable.

The final indexes are automatically capped between 0 and 1027 (the range of prime indexes).

The final output of the Reverb is $\langle \text{EARLY_LEVEL} \rangle$ multiplied the output of the All_Pass stage, plus $\langle \text{TAIL_LEVEL} \rangle$ multiplied the sum of all the Comb Filter outputs.

$\langle \text{MODAL_RICHNESS} \rangle$ is capped to 32, $\langle \text{DIFFUSION_STAGES} \rangle$ is capped to 8.

REVERB recap

Crescendo provides a suite of reverberation instructions that utilize parallel delays with feedback and series all-pass filters to simulate acoustic spaces. To ensure high-quality diffusion and minimize unwanted resonances, the engine uses prime numbers for its internal delay lengths. All reverb parameters are continuously automatable and can be driven by VST variables or MIDI CCs.

REVERB: Standard Stereo Reverb

The standard `REVERB` instruction is designed for general-purpose spatialization. It utilizes 8 parallel delays and 4 series all-pass filters to create a dense reverberation tail.

- **Syntax:** `REVERB(<input>, <LPROOM>, <HPROOM>, <FC>, <STEREO>)`.
- **Example Usage:** `OUT = OUT + 0.3 * REVERB(OUT, 0.85, 0.5, 5000, 2)`.

In this manual example, the reverb is applied to the combined output of all layers. The $\langle \text{LPROOM} \rangle$ value of 0.85 provides a long decay for low frequencies, while the $\langle \text{HPROOM} \rangle$ of 0.5 results in faster damping for high frequencies. The crossover frequency $\langle \text{FC} \rangle$ is set to 5000 Hz, and the $\langle \text{STEREO} \rangle$ offset of 2 shifts the delay primes to widen the stereo image.

REVERB0: Fast Optimized Reverb

For projects where CPU resources are constrained, `REVERB0` offers a streamlined alternative with fewer user-definable parameters. It uses fixed internal settings for the crossover frequency and stereo offset to maximize processing speed.

- **Syntax:** `REVERB0(<input>, <ROOMSIZE>).`
- **Example Usage:** `WET_SIGNAL = REVERB0(DRY_SIGNAL, 0.9).`

The `<ROOMSIZE>` parameter acts as the feedback coefficient for the low-frequency portion of the signal, with higher values approaching infinite sustain.

REVERB1: Extended Delay Offset

`REVERB1` builds upon the standard model by adding a parameter to offset the starting prime number used for the delays. This allows developers to simulate significantly larger or smaller rooms by shifting the base delay times.

- **Syntax:** `REVERB1(<input>, <LPROOM>, <HPROOM>, <FC>, <STEREO>, <DELAY_OFFSET>).`
- **Example Usage:** `LARGE_HALL = REVERB1(INZ, 0.9, 0.8, 4000, 5, 50).`

A `<DELAY_OFFSET>` of 50 shifts the internal prime indices upward, resulting in longer initial delays characteristic of a large hall.

REVERB2: Custom Diffusion Gain

The `REVERB2` instruction introduces control over the gain of the all-pass filters used for diffusion. This is particularly useful for adjusting the "thickness" or density of the reverb's initial reflections.

- **Syntax:** `REVERB2(<input>, <LPROOM>, <HPROOM>, <FC>, <STEREO>, <DELAY_OFFSET>, <G_ALL_PASS>).`
- **Example Usage:** `OUT = REVERB2(OUT, 0.7, 0.6, 6000, 1, 0, 0.618).`

Setting `<G_ALL_PASS>` to 0.618 (the inverse of the golden ratio) ensures a true all-pass response, while other values can be used to vary the diffusion characteristics.

REVERB3: Advanced Configuration

`REVERB3` is the second most flexible implementation, allowing the developer to specify the exact number of parallel delays and series all-pass filters. It provides granular control over the starting prime indices for both the delay and all-pass stages.

- **Syntax:** `REVERB3(<input>, <LPROOM>, <HPROOM>, <FC>, <STEREO>, <FIRST_DELAY>, <NUM_DLY_FF>, <FIRST_ALL_PASS>, <NUM_ALL_PASS>, <G_ALL_PASS>).`
- **Example Usage:** `OUT = REVERB3(OUT, 0.8, 0.4, 5000, 2, 111, 16, 40, 8, 0.5).`

In this complex configuration, the engine is instructed to use 16 parallel feedback delays starting from the 111th prime and 8 all-pass stages starting from the 40th prime.

REVERB4: The Ultra-High Fidelity Engine

REVERB4 is the flagship spatial processor of Crescendo. It provides surgical access to the internal prime-number matrices, allowing for advanced acoustic modeling that goes far beyond standard VST algorithms.

Syntax: REVERB4(<input>, <LPROOM>, <HROOM>, <FC>, <STEREO_WIDTH>, <ROOM_SIZE>, <COMPLEXITY>, <MODAL_RICHNESS>, <TAIL_LEVEL>, <INITIAL_REFLECTIONS>, <DIFFUSION>, <DIFFUSION_STAGES>, <DENSITY>, <EARLY_LEVEL>)

Example Usage (High-End Studio Hall):

```
WET = REVERB4(IN, 0.95, 0.93, 5000, 1.0, 120, 5.0, 12, 0.7, 40, 2.0, 6, 0.618, 1.5)
```

Key Parameter Breakdown:

- **The Diffusion Stage:** Controls the "blur" of the attack. By setting <DIFFUSION_STAGES> to 6 and <DENSITY> to 0.618 (Golden Ratio), you achieve a perfectly transparent onset.
- **The Modal Tank:** Defined by <MODAL_RICHNESS> (number of parallel filters). A value of 12-16 is ideal for most instruments, while 32 provides an "infinite" cinematic density.
- **Geometric Scaling:** Unlike a simple decay knob, the duration is a synergy between <LPROOM/HROOM> (energy conservation) and <COMPLEXITY> (physical distance between reflections).
- **Presence Boost:** Use <EARLY_LEVEL> to reinforce the instrument's body and "air" without washing it away in the tail.

Pro Tip for Developers: In the simplified versions (**REVERB0** through **REVERB2**), the engine automatically injects the **0.618 Golden Ratio** and a **1.0 Presence Boost**. This ensures that even a single-parameter reverb instruction carries the "expensive" high-end sheen of the full REVERB4 engine.

Implementation Note: The "Presence" Secret

If your mix feels "muddy" or "distant," do not simply lower the reverb volume. Instead, switch to **REVERB4** and:

1. **Lower the <TAIL_LEVEL>** to 0.5.
2. **Increase the <EARLY_LEVEL>** to 1.8 or 2.0.
3. **Increase <COMPLEXITY>** to 7.0.

This configuration creates a "High-Definition" space where the instrument stays front-and-center, enveloped in a sophisticated, clear atmospheric cloud rather than a thick digital fog.

Implementation in the POST Section

For global effects, reverb instructions should typically be placed in the `POST` section to process the summed output of all active layers. Developers often implement a Dry/Wet control using a VST variable to blend the reverberated signal with the original audio.

- **Dry/Wet Mix Example:**
VSTVAR 0, 50, "Dry/Wet", "%", 0, 100, 1
POST
WET = REVERB0(OUT, 0.85)

```
MIX = (VAR0 / 100)
OUT = (OUT * (1 - MIX)) + (WET * MIX).
```

This setup allows the user to rotate a knob on the Crescendo GUI to transition from a completely dry signal to a fully reverberated one.

The Simple Reverb example file

Here it is a simplified version of the SimpleReverb.txt instrument file:

```
INTERFACE 800,230
VSTVARS 6
VSTVAR 0, .94, "LPROOMSIZE", "", .9, .9999, 0
VSTVAR 1, .92, "HPROOMSIZE", "", .9, .9999, 0
VSTVAR 2, 5000, "F Thres", "Hz", 10, 20000, 0
VSTVAR 3, 50, "Dry/Wet", "%", 0, 100, 1
VSTVAR 4, 0, "Pre-Delay", "ms", 0, 100, 1
VSTVAR 5, 1, "RVB Wide", "", 0, 5, 1
POST
IF MCC(603)<.1 EXIT
OUT=.01*((100-
MCC(603))*OUT+MCC(603)*WIDE(DELAY(REVERB(OUT,MCC(600),MCC(601),MCC(602),-
1),MCC(604)*.001),MCC(605)))
```

Technical Analysis: The "Simple Reverb"

The `Simple Reverb` file is a high-efficiency spatial processor designed to run in the `POST` layer. By nesting multiple DSP operators into a single functional pipeline, it achieves professional-grade results (Reverb + Pre-Delay + Stereo Widening) with a fixed CPU overhead (approx. **2.3%** on HEDT platforms).

Functional Pipeline Architecture

The module uses a nested execution logic: `OUT = WIDE(DELAY(REVERB(OUT, ...), ...), ...)`

- **REVERB Stage:** Based on a feedback-delay network (FDN) similar to the *Freeverb* architecture (8 parallel comb filters and 4 serial all-pass filters). It is non-interpolating, which allows it to process high-density reflections with negligible CPU cost.
- **DELAY Stage (Pre-Delay):** Positioned after the reverb but before the mix/widening. This controls the temporal separation between the dry signal and the onset of reflections, essential for maintaining clarity in the original sound.
- **WIDE Stage (Stereo Imaging):** A post-reverb width processor that expands the stereo field of the wet signal.

Dual-Band Decay Control

Unlike basic reverbs with a single "Room Size" parameter, this implementation features frequency-dependent feedback:

- **Low-Frequency Room Size (mcc 600):** Controls the decay time of the lower spectrum.
- **High-Frequency Room Size (mcc 601):** Controls the decay of the higher spectrum.

- **F-Threshold (mcc 602):** Defines the crossover point. This allows the user to simulate acoustically treated rooms (short high-decay, long low-decay) or "bright" reflective spaces.

CPU Optimization & Performance

- **Short-Circuit Logic:** The `IF MCC(603) < .1 EXIT` instruction ensures that if the Dry/Wet mix is at zero, the entire DSP chain is bypassed. This reduces the CPU load from **2.8%** to a mere **0.5%** (host baseline).
- **Cache Alignment:** By default, the reverb's internal delay lines are sized to fit within the CPU's **L1 and L2 caches**. This prevents "Memory Stalling," allowing the reverb to maintain the same performance even when the host is under heavy load from other instruments.
- **In-Place Processing:** By nesting the functions, the engine avoids redundant memory copying between buffers. The audio data stays "hot" in the CPU registers as it passes from the Reverb to the Delay and finally to the Widener.

Usage Tips

- **For External Audio:** Because this sits in the `POST` layer, you can use *Crescendo* as a standalone high-end reverb unit for external instruments (guitars, vocals) without sending any MIDI notes.
- **Stereo Width:** Setting `RVB Wide` to **0** creates a perfectly monophonic reverb, which is useful for "vintage" production styles or for centering a lead instrument while still providing depth.

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.

Pitch shifters: Recap

Crescendo provides real-time pitch-shifting capabilities through the `PSHIFT` and `PSHIFT2` functions, allowing for the modification of audio pitch without altering the playback tempo. These functions are typically implemented within assignments in the `POST` or `LAYER` sections to process signals either globally or per-note.

PSHIFT: Standard Pitch Shifter

Example Usage: `OUT = PSHIFT(OUT, 12)` *In this manual example, the final output of the plugin is shifted up by exactly one octave (12 semitones).*

PSHIFT2: Advanced Configurable Pitch Shifter

Example Usage with Automation:

```
VSTVAR 0, 0, "Detune", "semis", -12, 12, 1
```

```
POST
```

```
OUT = PSHIFT2(OUT, VAR0, 30, 20, 15)
```

In this example, a VST variable (knob) allows the user to adjust the pitch between -12 and +12 semitones. The algorithm is configured with a 30ms latency, a 20ms backward search, and a 15ms range for overlap calculation.

Chorus, Flanger and Phaser

CHORUS (N, IN, DMIN, DDELTA, FLFO, MIX)

The **CHORUS** function is designed to add thickness, width, and "shimmer" to any audio source. It achieves this by duplicating the input signal into multiple delayed voices (N), each modulated by an independent Low-Frequency Oscillator (LFO). This mimics the effect of multiple musicians playing the same part with slight variations in timing and pitch.

Unlike basic mono-to-stereo choruses, Crescendo utilizes a fixed **Quadrature Phase Offset** (90°) between the Left and Right LFO banks.

- **The Result:** When the Left channel voices are at their maximum pitch shift (fastest delay change), the Right channel voices are at their "turnaround" point (zero pitch shift).
- **The Benefit:** This eliminates the "volume pumping" effect common in lesser choruses and creates a rock-solid, wide stereo image that feels "glued" to the speakers, making it the perfect companion for **SuperSaw** leads and lush **SoundFont** pads.

Parameters

Parameter **Function** **Range / Technical Note**

N Voice Count **1 to 8 voices**. Higher values create a denser, "cloud-like" texture.

IN Audio Input Supports stereo signals.

DMIN Base Delay Usually set between **10ms and 30ms** for natural thickening.

DDELTA Mod Depth The "intensity" of the pitch wobble.

FLFO LFO Speed Usually **0.1Hz to 1.5Hz**. Uses sinusoidal wave.

MIX Dry/Wet 0.0 to 1.0. For maximum thickness, 0.5 (50/50) is recommended.

Expert Sound Design Tips

1. The "12-String" Effect

To simulate a 12-string guitar or a multi-oscillator synth using a single-oscillator source, we use a low number of voices and a very slow modulation to mimic the natural, subtle tuning discrepancies of double strings.

- **N:** 2 (Simulates the primary and the octave/unison string).
- **DMIN:** 0.02s (20ms) - Creates enough separation to "de-quantize" the digital feel.
- **DDELTA:** 0.005s - Keeps the pitch variance subtle and musical.
- **FLFO:** 0.2Hz - A very slow rate to avoid a "wobbly" pitch sensation.
- **MIX:** 0.5 (Equal blend of dry and wet).

```
// Example 1: 12-String Effect
WAVE = OSCG(a sampled or synth slot similar to a string)
OUT = CHORUS(2, WAVE, 0.02, 0.005, 0.2, 0.5)
```

2. SuperSaw Optimization

When processing a SuperSaw that already has internal detune, the goal is to add a lush stereo halo without washing out the transients.

- **N: 8** - Maximize the "blurring" effect of the sidebands for a massive wall of sound.
- **DMIN: 0.012s** - A tighter delay keeps the sound focused.
- **DDELTA: 0.003s** - Low delta to prevent the already detuned Saw from sounding "out of tune."
- **FLFO: 0.6Hz** - Provides a gentle movement.
- **MIX: 0.35** - Maintains the "bite" of the original transients while surrounding them with a lush stereo atmosphere.

```
// Example 2: SuperSaw Stereo Halo
// Standard lush detuned sound
// Detune: 0.05 (stereo spread), Mix: 0.5 (amplitude of side harmonics)
SAW = GAIN * SUPERSAW0(0.05, 0.5)
OUT = CHORUS(8, SAW, 0.012, 0.003, 0.6, 0.35)
```

3. High-Speed "Rotary" Simulation

For an organ or an "electric" piano vibe, we simulate the physical rotation of a speaker (Doppler effect) by using high-speed modulation and a larger delay depth.

- **N: 3** - Provides enough density for the rotation without becoming a chaotic mess.
- **DMIN: 0.008s** - Short delay to emphasize the frequency modulation over the echo.
- **DDELTA: 0.012s** - A higher value creates the swirling, rotating sensation.
- **FLFO: 6.5Hz** - Mimics the "Fast" setting of a Leslie speaker cabinet.
- **MIX: 0.7** - High mix to make the rotating effect dominant.
- The 90° phase offset will create a swirling, rotating sensation similar to a Leslie speaker cabinet.

```
// Example 3: Rotary Speaker Simulation
// Using an almost Ssquare wave (Type 0, sinusoidal with exponent 0.1) for an
organ-like tone
ORGAN = OSCG("ygf0000", 0, 0.1)
OUT = CHORUS(3, ORGAN, 0.008, 0.012, 6.5, 0.7)
```

Quick Parameter Reference

- **DMIN (10ms - 25ms):** The "sweet spot" for chorus. Below 10ms, it starts sounding like a Flanger; above 25ms, it starts sounding like a Slapback Delay.
- **DDELTA (0.002 - 0.015):** Controls the depth. Use lower values for "Ensemble" effects and higher values for "Vibrato" or "Rotary" effects.
- **FLFO (0.1Hz - 8.0Hz):** Use low rates for thickness and high rates for mechanical movement.

FLANGER (IN, DMIN, DDELTA, FLFO, FEEDBACK, MIX)

The **FLANGER** is a resonant comb-filter effect designed to create the iconic "jet-plane" sweeping sound. While technically related to the Chorus, the Flanger operates with significantly shorter delay times (D_{min}) and introduces a **Feedback** loop. This loop recirculates the output back into the input, creating sharp, tunable resonant peaks and notches in the frequency spectrum.

The Industry Standard: Triangle Modulation

By default, the Flanger utilizes a **Triangle Waveform**. In the world of high-end DSP, the Triangle wave is preferred for flanging because:

- **Linear Pitch Shift:** The constant slope of a triangle wave ensures the pitch shift remains perfectly stable throughout the sweep.
- **Predictable Motion:** Unlike a Sine wave, which slows down at the peaks, the Triangle wave ensures the "comb-filter" moves through the frequency spectrum at a constant speed, providing a more aggressive and "mechanical" sweep.

Parameters

Parameter Description / Technical Note

IN Stereo Audio Input.

DMIN **Base Delay (0.1ms - 5ms).** Controls the "height" of the resonance.

DDELTA **Sweep Range.** Determines how far the "jet" travels.

FLFO **LFO Frequency.**

FEEDBACK **Regeneration Level (-1.0 to 1.0).**

MIX **Dry/Wet Balance.** 0.5 (50%) provides the deepest notches.

All parameters are stereo and automatable.

Phase Physics in Crescendo:

- **Negative Feedback:** By setting feedback to a negative value (e.g., -0.9), the phase of the recirculating signal is inverted. This causes cancellation at frequencies where positive feedback would normally create a peak, resulting in a unique, "inside-out" harmonic structure.
- **Manual Control Latency:** Because Crescendo's `FLANGER` operator updates its parameters at high resolution, manual automation of `DMIN` via `MCC` is extremely smooth. This allows the flanger to act as a **Physical Modeling** resonator, where the physical length of the "virtual pipe" is controlled by the performer.

Expert Sound Design Tips

1. The "Negative Feedback" Bite

This setup uses a high negative feedback value to create a hollow, "colder" timbre by shifting the comb filter notches to even harmonics.

```
// Recommended VSTVAR for Feedback: -1.0 to 1.0
// Negative feedback creates that "hollow" metallic character.

POST
// Check if Dry/Wet (MCC 609) is active to save CPU
IF MCC(609) < .001 EXIT

// Parameters: DMIN=2ms, DDELTA=1ms, LFO=0.4Hz, FEEDBACK=-0.85, WET=MCC(609)
OUT = FLANGER(OUT, 0.002, 0.001, 0.4, -0.85, MCC(609))
```

2. Zero-Crossing Simulation (Tape Flanging)

To simulate the classic 60s "through-zero" effect, we set the delay floor as low as possible. By keeping feedback at 0, we maximize the pure phase cancellation when the delayed signal aligns with the dry signal.

```
C
// Parameters:
// DMIN=0.0001 (0.1ms - the absolute minimum for phase cancellation)
// DDELTA=0.0004 (Small range for intense "whoosh")
// FLFO=0.15 (Slow sweep for that dramatic tape feel)

POST
IF MCC(609) < .001 EXIT

// Feedback is 0 for the cleanest "disappearing" phase cancellation
OUT = FLANGER(OUT, 0.0001, 0.0004, 0.15, 0, MCC(609))
```

3. Manual "Wah" Flanging

In this configuration, we disable the internal LFO ($FLFO = 0$) and use a MIDI controller (like a Mod Wheel or Expression Pedal) to manually sweep the delay time. High positive feedback adds the "resonant" peak typical of a Wah-Wah pedal.

```
C
// Assume MCC(605) is your manual controller (0.0 to 1.0)
// FLFO = 0 disables the internal modulation.

LAYER
// Scaling MCC(605) to a range of 0.1ms to 10.1ms
// Feedback 0.92 provides the high resonance needed for the "Wah" effect
OUT = FLANGER(OUT, (MCC(605) * 0.01) + 0.0001, 0, 0, 0.92, 0.5)
```

CHOFLA (N, IN, DMIN, DDELTA, FLFO, FEEDBACK, MIX, WAVE_TYPE)

CHOFLA is a high-density hybrid operator that bridges the gap between a lush multi-voice Chorus and a resonant Flanger. By allowing the user to specify a `WAVE_TYPE` per channel, it creates complex phase relationships that can be automated in real-time, resulting in "living" textures that evolve with the music.

Core Features

- **Multi-Voice Density (N):** Supports up to **8 voices** per channel. Unlike a standard Flanger (usually 1 voice), CHOFLA can create a "wall of sound" where multiple resonant peaks sweep across the frequency spectrum simultaneously.
- **Stereo Independent LFOs:** Each channel (L/R) has its own LFO frequency (`FLFO`) and waveform type (`WAVE_TYPE`).
- **Resonant Feedback:** Includes a dedicated feedback loop for each channel, allowing for "metallic" chorus sounds or "shimmering" flangers.

You must use the JOIN function to exploit the stereo nature of this function.

The Stereo WAVE_TYPE "Trick"

The standout feature of CHOFLA is the **Stereo-Independent, Automatable Waveform Type**.

Because `WAVE_TYPE_L` and `WAVE_TYPE_R` are processed independently every sample, you can achieve psychoacoustic effects that standard modulation effects cannot:

1. **Waveform Morphing:** By automating the `WAVE_TYPE` via MIDI CC or a Modulation Matrix, you can transition a sound from a smooth Sine-based swirl (Type 0) to a jagged, aggressive Triangle rip (Type 3).
2. **Asymmetric Modulation:**
 - **Left Channel:** Set to **Sine (0)** for a classic, gentle pitch wobble.
 - **Right Channel:** Set to **Triangle (3)** for a sharp, linear sweep.
 - **Result:** The listener's brain struggles to "track" the modulation, resulting in an incredibly wide and "3D" stereo image where the left side feels "liquid" and the right side feels "mechanical."

Parameter Table

| Parameter | Function | Technical Note |
|-----------------|--------------------|--|
| N | Voice Count (1-8) | Fixed at trigger. Higher values increase CPU load. |
| IN | Stereo audio Input | |
| DMIN | Base Delay | Shorter values = Flanger; Longer values = Chorus. |
| DDELTA | Modulation Depth | The range of the LFO sweep in samples. |
| FLFO | LFO Frequency | |
| FEEDBACK | Regen Level | Can be positive or negative (-1.0 to 1.0). |

| Parameter | Function | Technical Note |
|------------------|-----------|---|
| MIX | Dry/Wet | 0.0 (Dry) to 1.0 (Wet). |
| WAVE_TYPE | LFO Shape | Stereo & Automatable. Maps to <code>SAMPLE</code> shapes 0-12. |

Expert Tip: The "Phase-Shifted SuperSaw"

To make a SuperSaw sound "massive" but still defined, use **N=4**, a low **FEEDBACK** (0.2), and set the **WAVE_TYPE** to **Type 9 (Smoothed Saw)**. The internal 90° phase offset between Left and Right, combined with the "jumping" nature of the saw wave, will create a rhythmic, percussive energy within the pad.

CHOFLA2 (N, IN, DMIN, DDELTA, FLFO, FEEDBACK, MIX, WAVE_TYPE, WAVE_P)

CHOFLA2 is the advanced version of the hybrid engine. While **CHOFLA** uses standard fixed waveforms, **CHOFLA2** introduces the **WAVE_PARAM**, allowing you to morph the geometry of the LFO in real-time. This is essential for creating "non-linear" sweeps, where the modulation speeds up or slows down within a single cycle.

The Power of Geometric LFOs

By modulating the **WAVE_PARAM**, you can change the "tension" of the sweep:

- Variable Slope Triangle (Type 2):**
 - Param = 0.5:** Perfect Triangle (linear up/down).
 - Param = +-1:** rising or falling saw.
- PWM Square (Type 1):**
 - Changes the duty cycle. Great for rhythmic "gate-flanging" where the effect stays "on" for 20% of the cycle and "off" for 80%.
- Warped Sine (Type 0):**
 - Adds "power" to the sine, pushing it towards a square-like shape. This makes the pitch-shift hang longer at the extremes, creating a more dramatic "vibrato-shelf."

Parameter Table

| Index | Parameter | Description |
|-------|-----------|--------------------|
| 0 | N | Voice Count (1-8). |

| Index | Parameter | Description |
|-------|-----------|-----------------------------|
| 1 | IN | Audio Input. |
| 2 | DMIN | Base Delay. |
| 3 | DDELTA | Sweep Depth. |
| 4 | FLFO | Frequency (Pre-multiplied). |
| 5 | FEEDBACK | Regen Level (Stereo). |
| 6 | MIX | Dry/Wet balance. |
| 7 | WAVE_TYPE | Base Shape (0-5). |
| 8 | WAVE_P | Geometry Control |

CHOFLA3 (N, IN, DMIN, DDELTA, FLFO_L, FLFO_R, FEEDBACK, MIX, WAVE_TYPE, WAVE_P)

CHOFLA3 is the flagship evolution of the modulation engine. While maintaining the high-density 8-voice architecture of the previous operators, it introduces **independent LFO frequencies** for the Left and Right channels. This decoupling of the stereo field creates an organic, non-repeating movement that is essential for cinematic pads, wide SuperSaw leads, and complex sound design. This can be done also with CHOFLA2 just by using JOIN on the FLFO, but CHOFLA3 is a little bit faster.

The Innovation: Asynchronous Modulation

In a standard Chorus or Flanger, the Left and Right channels are locked to the same frequency, typically separated by a fixed phase offset (like the 90° implemented in CHORUS).

CHOFLA3 breaks this symmetry by allowing `FLFO_L` and `FLFO_R` to run at different speeds:

- **The "Orbital" Effect:** By setting slightly different rates (e.g., 0.41 Hz on the Left and 0.43 Hz on the Right), the phase relationship between the two channels constantly drifts. The stereo image "rolls" and "breathes," providing a sense of depth that never feels static or "looped."

- **Mono Compatibility:** Asynchronous modulation significantly reduces the risk of phase cancellation when the signal is summed to mono, as the notches in the frequency spectrum are almost never in the same place for both channels simultaneously.

Variable Geometry Control

Like its predecessor (CHOFLA2), this operator utilizes the **WAVE_PARAM** to reshape the LFO waveform in real-time. Whether using a warped Sine or a variable-slope Triangle, the geometry remains stereo-independent, allowing for a "Liquid" texture on one side and a "Mechanical" pulse on the other.

Parameter Table (CHOFLA3)

| Index | Parameter | Description |
|-------|-----------|---|
| 0 | N | Voice Count (1-8). Fixed at trigger. |
| 1 | IN | Stereo Audio Input. |
| 2 | DMIN | Base Delay Time. |
| 3 | DDELTA | Modulation Depth (Sweep Range). |
| 4 | FLFO_L | LFO Speed for the Left channel. |
| 5 | FLFO_R | LFO Speed for the Right channel. |
| 6 | FEEDBACK | Independent Stereo Regeneration Level. |
| 7 | MIX | Dry/Wet Balance. |
| 8 | WAVE_TYPE | LFO Shape (Sine, Tri, Saw, etc.). |
| 9 | WAVE_P | Geometric Control |

Expert Sound Design Tips

1. The "Dual-Rate" Shimmer

For a "Boutique" chorus sound, set **FLFO_L** to a slow, drifting rate (0.2 Hz) and **FLFO_R** slightly faster (0.6 Hz). This creates a shimmering effect where one side of the soundstage provides the "body" while the other provides the "motion."

2. Rhythmic Flanging

With high **FEEDBACK** and **N=1**, use two different but mathematically related LFO speeds (e.g., 1.0 Hz and 1.5 Hz, a 2:3 ratio). This creates a polyrhythmic flanging effect where the resonant peaks cross each other at musical intervals.

PHASER (N, IN, FLFO, DEPHASE, CENTER, DEPTH, FEEDBACK, MIX, LFO_WAVE)

The **PHASER** creates a lush, swirling frequency response by shifting the phase through a chain of All-Pass filters. It supports up to 32 stages and full stereo independence via `JOIN()`.

- **Structure:** A 4-stage Phaser (typical vintage sound) uses 4 slots per channel. A 12-stage "Mega-Phaser" uses 12.
- **The Math:**

$$y[n] = a \cdot x[n] + x[n-1] - a \cdot y[n-1]$$

Where a is modulated by a periodic function to move the phase notches.

Stereo Power & JOIN()

Since every parameter is stereo, you can use the `JOIN(left, right)` function to differentiate any aspect of the effect between the two channels.

- **Asynchronous Sweeps:** Use `JOIN()` on **FLFO** or **DEPHASE**.
- **Spectral Skewing:** Use `JOIN()` on **CENTER** to set different frequency focal points for L and R.
- **Width Control:** Use `JOIN()` on **DEPTH** or **FEEDBACK** to create asymmetrical textures.

Technical Notes

- **Phase Logic:** The LFO position is calculated at every sample, allowing for high-speed Phase Modulation (PM) via the **DEPHASE** parameter.
- **Exponential Sweep:** The modulation travels in octaves. A **DEPTH** of 1.0 sweeps one octave above and below the **CENTER** frequency.
- **N Parameter:** The number of stages is sampled at trigger and forced to an even value (max 32).

Parameters

| Index | Parameter | Description |
|-------|-----------|--|
| 0 | N | Number of Stages (2 to 32, even). Higher values = deeper "vocal" sound.
<i>Sampled at trigger.</i> |
| 1 | IN | Stereo Audio Input. |
| 2 | FLFO | LFO Frequency (Hz). |
| 3 | DEPHASE | LFO Phase Offset (0.0 to 1.0). Real-time phase modulation for "FM-style" growls. |
| 4 | CENTER | Center Frequency (Hz). Base point of the sweep. |
| 5 | DEPTH | Sweep Range (+/- Octaves). Width of the movement. |
| 6 | FEEDBACK | Feedback Amount (-1.0 to 1.0). Sharpens peaks. Negative values create hollow tones. |
| 7 | MIX | Dry/Wet Balance. 0.5 is maximum notch depth. |
| 8 | LFOWAVE | LFO Waveform Code. See <code>SAMPLE</code> . |

Tips and examples

Tip: Set FLFO to 0.0 and modulate CENTER with an ENV() operator to create an auto-wah or manual vocal filter.

```
// Stationary Phaser as a comb-like resonator
// N=32, FLFO=0, CENTER=Variable, DEPTH=0, RES=0.98
FREQ = JOIN(220, 222) // Slight detune for stereo thickness
OUT = PHASER(32, IN, 0, 0, FREQ, 0, 0.98, 0.5, 0)
```

Infinite Modulation Possibilities Unlike standard effects, every parameter in Crescendo is fully automatable at audio rate. You can modulate the Phaser's center frequency or the Flanger's delay time using LFOs, secondary oscillators, or even external sample data. This opens the door to complex FM-style filtering and cross-synthesis textures, all while maintaining hardware-level stability through our optimized signal path.

Spatial Image Functions

The Spatial Image Functions in Crescendo provide the surgical tools necessary to manipulate the stereo field. While many VSTs treat stereo as a fixed "left+right" pair, Crescendo allows you to deconstruct, re-route, and reshape the stereo image at any point in the signal chain—whether you are processing individual oscillator layers or final global outputs.

These functions are essential for creating wide, cinematic textures, fixing phase issues, or developing complex "dual-mono" processing chains where the left and right channels are treated with entirely different logic.

1. Extraction and Mono Conversion

These fundamental tools allow you to isolate specific parts of the stereo signal or collapse them for compatibility.

- **MONO(<input>)**: Averages the left and right channels into a single mono signal. This is the primary tool for ensuring mono compatibility or preparing a signal for effects that require a single-channel input.
- **LEFT(<input>)** / **RIGHT(<input>)**: These functions "strip" the signal, returning only the specified channel. If the input is already mono, the value is returned unchanged.

2. Reconstructing the Field: JOIN

The **JOIN(<op1>, <op2>)** function is one of the most powerful architectural tools in the engine. It takes two independent mono expressions and "glues" them together into a new stereo pair.

- **Logic**: The first operand becomes the **Left** channel; the second becomes the **Right**.
- **Use Case**: This allows for "True Stereo" processing. For example, you can **JOIN** two different oscillators, or two different filters, to create a sound that is fundamentally different in each ear.

3. Stereo Width and Image Manipulation

Crescendo offers dynamic control over how "wide" or "centered" a sound feels.

- **WIDE(<input>, <w>)**: A continuous stereo-image processor.
 - **W = 1**: Original signal (no change).
 - **W = 0**: Collapses the signal to pure mono.
 - **W = -1**: Hard-swaps the left and right channels.
 - *Intermediate values (e.g., 0.5) allow for a gradual narrowing of the field, perfect for automating a sound that "spreads out" over time.*
- **PAN(<input>, <pan>)**: A standard percentage-based panner.
 - **Range**: -100 (Hard Left) to +100 (Hard Right).
 - **Behavior**: As you pan toward the extremes, stereo signals are gradually summed to mono to ensure no signal energy is lost when pushed to a single speaker.

4. Discrete Gain Control: `GAINS`

For absolute precision, `GAINS (<input>, <L>, <R>)` provides independent volume control for each side of the stereo image. Unlike a panner, which shifts the balance, `GAINS` allows you to boost or attenuate the Left and Right channels using separate formulas or VST variables.

Developer Tip:

When using effects like `CHOFLA` or `PHASER`, try using `JOIN()` with different LFO speeds for each operand. By combining asynchronous modulation with these Spatial Image functions, you can create a "living" stereo image that feels wider and deeper than standard "linked" effects.

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.

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].

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.

Other Instructions and functions

The HOLD construct:

The syntax is **HOLD <var> = <complex expression>**.

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>**

Example Usage: Capturing Trigger States

A common use for HOLD is to capture the position of a MIDI CC or a random value at the start of a note so that the parameter remains consistent throughout the note's duration, even if the user moves the physical controller later.

Capture Filter Cutoff at Note Start:

LAYER

```
HOLD START_CUTOFF = MCC(74)
OUT = FILT(0, OSCG("s1f"), START_CUTOFF, 0.5)
```

In this example, the filter cutoff for the note is "locked" to the position of MIDI CC 74 at the moment the key was struck. Subsequent movements of CC 74 will not affect the currently ringing note.

Capturing Randomness:

```
LAYER
HOLD PITCH_DRIFT = RND(10)
OUT = OSCG("s1c", PITCH_DRIFT)
```

*This uses the **HOLD** construct to generate a random pitch offset of up to 10 cents once per note. Without the **HOLD** keyword, the **RND** function would generate a different value every sample, resulting in white noise rather than a stable pitch shift.*

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.

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
```

This construct 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...

Another example: Unified Per-Voice Filter and Reverb Send

In this scenario, we have a multi-layered instrument (e.g., a synth with an "Attack" layer and a "Sustain" layer). We want both layers to be processed by a final per-voice low-pass filter and then sent to a global reverb bus (SENDS1) before they leave the layer stage.

```

// --- COMMON SECTION ---
VSTVARS 2
VSTVAR 0, 1000, "Cutoff", "Hz", 20, 20000, 0
VSTVAR 1, 0.2, "Verb Send", "", 0, 1, 1

// Define global variables based on VST knobs
GLOBAL_FC = VAR0
REVERB_AMT = VAR1

// These instructions will be automatically appended to every LAYER block.
// Note: Variable checking happens during the import into the layer.
LAST OUT = FILT(0, OUT, GLOBAL_FC, 0.5)
LAST SENDS1 = OUT * REVERB_AMT

// --- LAYER SECTIONS ---

LAYER
// Layer 1: Noise Transient
OUT = OSCG("yg", 4, 0) // White noise
// The engine automatically adds the filter and send instructions here.

LAYER
// Layer 2: Main Sine Body
OUT = OSCG("sg") // Simple sine wave
// The engine automatically adds the filter and send instructions here.

```

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 <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 <op1> <comp> <op2> GOTO <label>

OR

IF <op1> <comp> <op2> EXIT

OR

GOTO <label>

OR

EXIT

Crescendo provides a set of low-level instructions to control the execution flow of your scripts on a **sample-by-sample** basis. Unlike `EXECIF` (see below), which evaluates conditions only at the moment a note is triggered, these instructions allow for dynamic branching and looping based on real-time values like LFOs, Envelopes, or VST Variables.

1. Syntax and Variations

| Instruction | Behavior |
|---|--|
| <code>IF <op1> <comp> <op2> GOTO <label></code> | Jumps to the specified label if the condition is true. |
| <code>IF <op1> <comp> <op2> EXIT</code> | Terminates the current section (LAYER or POST) if the condition is true. |
| <code>GOTO <label></code> | An unconditional jump to a label. |
| <code>EXIT</code> | An unconditional termination of the current section. |

Comparison Operators (<comp>): =, >, <, >=, <=, <>, !=

2. Technical Rules and Logic

- **Left-Channel Bias:** All comparisons are performed using the **Left channel** of the operands. If you are comparing stereo expressions, wrap them in `MONO ()` to ensure consistent results.
- **The "Assembly" Approach:** The language does not support `AND`, `OR`, `NOT`, or parentheses. Complex logic must be constructed using sequential tests and jumps, mirroring assembly language.
- **Label Requirements:** A `LABEL` must be attached to an instruction. If you need to skip to the very end of a layer, you must use `EXIT` rather because even a dummy label at the bottom of the script requires an assignment or instruction.
- **Notes:** To ensure stability, the engine disables certain optimizations for instructions associated with labels or those appearing before an `IF` block:
 - The instructions after a `LABEL` is never deleted by the dead code elimination, even if the result is not used. This is to have an instruction to jump to. This can be used for the `PREV` trick (see below).
 - If `<label>` does not exist, the `(IF ...)GOTO` instruction is ignored.
 - The `(IF ...)GOTO` (if valid) is always executed. This means that the eventual expressions used in the comparison and the jump microoperation itself are always executed.

- To avoid uninitialized variables, the dead code elimination is performed as if the instructions are all unconditional, so an instruction never executed because of an unconditional jump, may cause an instruction whose result is discarded to in any case be executed (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.
- If the comparison is between constants, the comparison is resolved at compile time and the GOTO or EXIT instruction is emitted based on the result of the comparison. E.g. IF 1 = 1 GOTO is transformed in an unconditional jump and IF 1 > 1 GOTO is deleted from the instruction stream. Use DEBUG level 2 or greater to see the details of the instruction parsing.
- **CAUTION: Infinite Loops are NOT Detected:** Crescendo does not have a "safety timeout" for loops. If you create a GOTO path that loops back without an escape condition, the audio processing thread will hang. Depending on your DAW, this will result in a total freeze, a crash, or silent output. Always ensure your loop counter or condition has a guaranteed exit.
- **WARNING: Memory-Based Instructions (Delays, Filters):** Skipping or re-executing instructions with internal memory (like DELAY, REVERB, or FILTER) will not work as expected.
 - **Skipping** stops the internal state from advancing (the "buffer" freezes).
 - **Re-executing** (looping) advances the state multiple times per sample.
 - Use flow control primarily for **memoryless** math or switching between different synthesis blocks.

3. Implementation Examples

Example 1: CPU Optimization (The "Fail-Fast" POST Section)

Bypass an expensive effect entirely if the Dry/Wet mix is near zero.

```
POST
// If the Mix knob (VAR0) is less than 0.1%, stop processing the POST section
IF VAR0 < 0.001 EXIT

// The CPU-heavy effect only runs if VAR0 is high enough
OUT = PPDELAY(OUT, 0.25, 0.8, 0.5, 5000)
```

Example 2: Branching Logic

In the Crescendo engine, there is no native ELSE or END keyword. To achieve a high-level "If-Then-Else" structure, you must use **Inverted Logic**: you test for the condition that would make you **skip** the first block.

Mechanism: The "Skip and Stop" Logic

The implementation relies on two key maneuvers:

1. **The Inverted Jump:** Use an IF...GOTO to check the **opposite** of your condition. If the opposite is true, you jump straight to the ELSE label.

2. **The Hard Stop:** If the jump wasn't taken, the "Then" block executes. You must then use `EXIT` (to stop the section) or a second `GOTO` (to jump to the end of the logic) to prevent the engine from "falling through" into the `ELSE` code.

Implementation Example: Global Effect Toggle

In this example, we use a VST Knob (`VAR0`) to decide whether the output of the instrument passes through a **Crushing Distortion** or a **Smooth Reverb**.

High-Level Concept:

IF (Distortion Mode is ON) **THEN** Apply Distortion **ELSE** Apply Reverb **END**

Crescendo Implementation:

```
POST
// Condition: VST VAR 0 acts as our switch.
// If VAR0 >= 0.5, we want Distortion. If VAR0 < 0.5, we want Reverb.

// 1. Test the INVERSE: If we are NOT in distortion mode, jump to ELSE
IF VAR0 < 0.5 GOTO ELSE_REVERB

// --- THEN BLOCK: Distortion ---
// This only runs if VAR0 >= 0.5
OUT = CHEBY(IN0, 3, 0.5)

// 2. THE EXIT: Stop the section so we don't accidentally process the Reverb
EXIT

LABEL ELSE_REVERB
// --- ELSE BLOCK: Reverb ---
// This only runs if the GOTO jump was triggered
OUT = REVERB(IN0, 0.8, 0.5, 5000, 1)
```

Why `EXIT` is the "High-Performance" Choice

When used in the `POST` section, the `EXIT` instruction is highly efficient. In the example above:

- If **Distortion** is active, the engine never even looks at the `REVERB` instruction.
- The CPU cycles required for the Reverb (which can be quite high) are completely saved for that sample.
- This allows you to build complex multi-FX scripts that only consume CPU for the specific modules currently "switched on."

Important: `EXIT` VS. `GOTO END`

- **Use `EXIT`:** If your If-Then-Else block is the **final** thing in your section. It stops everything immediately.
- **Use `GOTO <label_at_bottom>`:** If you have more code *after* the If-Then-Else block that needs to run regardless of which path was taken (e.g., a final master volume adjustment).

Example of Continuing After the Block:

```

POST
IF VAR0 < 0.5 GOTO ELSE_PATH
    OUT = op1
    GOTO FINALIZE // Skip the else, but don't exit the whole section
LABEL ELSE_PATH
    OUT = op2
LABEL FINALIZE
OUT = OUT * 0.9 // This runs for both paths

```

Example 3: Implementing Compound Statements

Because the language does not support high-level boolean operators like AND, OR, or NOT, nor does it support nested parentheses, developers must structure compound logic as a series of sequential tests and jumps. This manual approach mirrors assembly language, where each comparison is evaluated individually to determine the flow of execution.

Implementing an "AND" Compound Statement

To execute a block of code only when multiple conditions are true (e.g., IF VAR1 > 0.5 AND VAR2 < 0.8), you must use a "fail-fast" assembly structure. In this configuration, each IF statement checks for the *inverse* of the desired condition and jumps to a label that skips the logic if that inverse is met.

Example:

```

// We want logic to run only if VAR0 > 0.5 AND VAR1 < 100
IF VAR0 <= 0.5 GOTO END_LOGIC
IF VAR1 >= 100 GOTO END_LOGIC

// Target logic (only reached if both conditions above were bypassed)
OUT = OUT * 0.5

LABEL END_LOGIC

```

In this scenario, if the first comparison fails, the engine immediately jumps to END_LOGIC, saving the CPU cycles required to evaluate the second comparison.

Implementing an "OR" Compound Statement

To execute a block of code if *any* of several conditions are met (e.g., IF VAR1 > 0.5 OR VAR2 < 0.1), the structure uses multiple jumps directed at the same success label. If none of the conditions trigger a jump, the code must then jump to an exit point to skip the target logic.

Example:

```

// We want logic to run if VAR0 > 0.5 OR VAR1 < 0.1
IF VAR0 > 0.5 GOTO RUN_LOGIC
IF VAR1 < 0.1 GOTO RUN_LOGIC

// If neither GOTO was triggered, exit this logic block
GOTO SKIP_ALL

LABEL RUN_LOGIC
// Target logic for the OR condition
OUT = OSCG("sg")

LABEL SKIP_ALL

```


Using `GOTO <label>` or `EXIT` acts as a deterministic jump to ensure the logic block is not entered by default when conditions are not met.

Example 4: The "Instant Comment" (Debugging & A/B Testing)

When you are designing a sound, you often want to temporarily disable a large block of complex code—like an expensive reverb or a series of filters—without manually adding `//` to dozens of lines. Placing an unconditional `EXIT` above the block effectively "comments out" everything below it.

```
LAYER
// --- ACTIVE CORE ---
OUT = GAIN * OSCG("slf0000") // Basic Sine wave

EXIT // <--- Everything below this point is bypassed by the engine

// --- EXPERIMENTAL BLOCK ---
// I'm still tweaking these, so I'll keep them "offline" for now.
OUT = DISTORT(OUT, 15.0, 0.5)
OUT = REVERB(OUT, 0.85, 0.4, 8000, 1)
OUT = FILTER(OUT, 440, 0.9, 0)
```

Why do this? It is much faster to toggle a single `EXIT` instruction than to comment/uncomment a 50-line processing chain.

Example 5: Implementing a Simple Loop

`IF ... GOTO` can be used to create loops for memoryless mathematical operations.

```
LAYER
FOO = 0
ITERATION_COUNT = 0

LABEL LOOP_START
FOO = FOO + 0.1
ITERATION_COUNT = ITERATION_COUNT + 1

// Loop until we have performed the addition 10 times
IF ITERATION_COUNT < 10 GOTO LOOP_START

OUT = FOO * OSCG("sg")
```

This allows for iterative calculations within a single sample's processing cycle.

4. Summary of Best Practices

- **Uninitialized Variables:** Avoid skipping the first assignment of a variable; ensure all variables have a "base" value before any `GOTO` jumps.
- **Looping Math:** Use `IF ... GOTO` for iterative mathematical operations (like calculating a complex series) where the logic is memoryless.
- **Layer Merging:** You can save resources by merging multiple triggered layers into a single layer that uses `IF` conditions to decide which sound to play, which is highly beneficial for polyphony-constrained instruments. But if the conditions are trigger time condition often is better to use `EXECIF` (see below).

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.

Note: the values used in the checks for key and velocities are the original values received in the NOTE ON message, before the MIDI POST process steps.

Example 1: Conditionally Adding a Layer via VST Knob

```
LAYER OUT = OSCG("sg") // Base Sine Wave
EXECIF 600, 0.5, 1.0
OUT = OUT + 0.5 * OSCG("yg", 2, 0.5) // Add Sawtooth only if VST Var #0 is ON
```

In this scenario, if VST Variable #0 (Index 600) is at 0.5 or higher when the note starts, the second oscillator is added to the output. If the condition is not met, the assignment is skipped entirely for that specific note instance.

Example 2: High-Velocity Accent Layer

```
LAYER OUT = OSCG("slf", 200) // Standard Piano Sample
EXECIF 0, 127, 100, 127 OUT = OUT + OSCG("slf", 201) // Add Accent Sample only
for Velocities 100-127
```

By merging these into a single LAYER with EXECIF, the developer ensures both sounds are triggered simultaneously without exceeding polyphony limits that might occur with separate layer definitions.

Example 3: Low-Key Muting via Sustain Pedal

```
LAYER
OUT = OSCG("slf")
EXECIF 0, 48, 0, 127, 64, 64, 127 OUT = 0 // Mute notes below C2 if the Sustain
Pedal (CC 64) is pressed
```

Example 4: Contextual Humanization: EXECIF is a supported prefix within the MIDIPOST program to apply transformations to note data before they reach the sound engine.

```
MIDIPOST "EXECIF 0, 59, 0, 127 DTRIG = 0.25" // Add a 0.25 bar delay only to
notes below Middle C
MIDIPOST "EXECIF 0, 127, 0, 20 ONVEL = ONVEL * 2.0" // Boost velocity only for
very soft notes
```

Other Functions

VAR(<num>)

Used to access the VST VAR values.

<num> is mono and must be a numeric constant. Values outside 0 – 127 and 600 – 727 make VAR return 0. Both ranges are supported to avoid cut and paste errors.

There is a slight smooth to avoid abrupt changes (not on integer or beats VSTVARs). See the SMOOTH instructions for details.

Example: FOO = VAR(42)

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. Values outside 0 – 1007 make MCC return 0.

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). See the SMOOTH instructions for details.

The default MIDI channel is used to assess the MIDI CC value.
See above table for details on number assignment.

Example: `FOO = MCC(7) / 128`

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). See the SMOOTH instructions for details.

The default MIDI channel is used to assess the MIDI CC value.
See above table for details on number assignment.

Example: `FOO = MCC(VAR(0))`

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.

Example: `FOO = MCC(VAR(0), VAR(1))`

RPN(<mode>, <num>)

Used to access last RPN or NPRN message values. No smooth is performed.

<mode> is the type of message to link and the processing mode.

- 0: the RPN MSB is accessed.
- 1: the RPN LSB is accessed.
- 2: the 14 bit full RPN is accessed.
- 3: the 14 bit full RPN, scaled to [0, 1[is accessed.
- 4: the 14 bit full RPN, scaled to [-1, 1[is accessed.
- 5: the NRPN MSB is accessed.
- 6: the NRPN LSB is accessed.
- 7: the 14 bit full NRPN is accessed.
- 8: the 14 bit full NRPN, scaled to [0, 1[is accessed.
- 9: the 14 bit full NRPN, scaled to [-1, 1[is accessed.

<num> specifies the RPN/NRPN message number to fetch, is mono and can be automatable.

The default MIDI channel is used to assess the RPN/NRPN message value.

Example: accessing the master tuning

```
MASTER = RPN(2, 6)
```

RPN2(<mode>, <num>, <channel>)

Similar to RPN, but <channel> (still mono, but automatable) selects the MIDI channel used to assess the RPN/NRPN message value.

Example: accessing the master tuning of the given channel

```
MASTER = RPN2(2, 6, VAR(0))
```

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.

Examples:

```
CUTOFF = SHOLD(VAR(0), 0)           // Like the HOLD construct
CUTOFF2 = SHOLD(VAR(0), 1)          // At trigger and every 1 BAR
VOLUMEPE = SHOLD(MCC3(7,0), -1)     // Sampling is paused only at release stage
```

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 |

Example 1:

```
CURVE 42, ...  
OUT = CURVE(42, OUT) // Custom distortion
```

Example 2:

```
OUT = CURVE(-3, OUT) // x * ABS(x) distortion
```

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>)).`

The big difference with MOD2 of the SoundFont specification is that there, it is limited to MIDI CC values and a set of predefined algorithm, while in Crescendo MOD2, like all the other functions, can be used with any expression, including oscillators, envelopes, other MOD2 functions, etc. and supports custom CURVES.

Example 1: Classic Velocity-to-Gain Modulation

In this scenario, MOD2 is used to scale an oscillator's volume based on MIDI note velocity using a non-linear "concave" curve, which is a standard technique for emulating the expressive response of acoustic instruments,.

```
LAYER
VOL_CONTROL = MOD2(ONVEL, 127, -3, -2, 1, -2)
OUT = GAIN * VOL_CONTROL * OSCG("s1f0000")
```

In this manual example, the input velocity (`op1`) is transformed by a concave curve (-3) and multiplied by a constant value of 127 (`op2`) which is treated linearly (-2),. The `amount` is set to 1, and the final result is output through a linear transfer function (-2) to drive the oscillator amplitude,.

Example 2: Nonlinear Ring Modulation

Unlike standard ring modulators that perform simple multiplication, MOD2 can create complex sidebands by applying non-linearities to the carrier and modulator before and after they interact,.

```
LAYER
MODULATOR = OSCG("s1v0000", 100)
CARRIER = OSCG("s1f0000")
OUT = GAIN * MOD2(MODULATOR, CARRIER, -1, -10, 0.8, -3)
```

Here, the absolute value of the modulator is multiplied by a bipolar linear version of the carrier,.
The result is scaled by 0.8 and then subjected to a concave transfer function (-3) to add harmonic grit to the output,.

Example 3: Dynamic Envelope Scaling via VST Parameter

This example demonstrates using a VST variable to dynamically scale the intensity of an envelope through a "bipolar convex" transformation, allowing for highly customized performance controls,.

```
VSTVAR 0, 0.5, "Env Depth", "", 0, 1, 1
LAYER
MY_ENV = ENV0(0.1, 0.2, 0.5, 0.5)
SCALED_ENV = MOD2(MY_ENV, VAR0, -2, -12, 1, -2)
OUT = GAIN * SCALED_ENV * OSCG("s1f0000")
```

The envelope (`op1`) remains linear (-2), while the VST knob (`op2`) is mapped to a bipolar convex curve (-12), providing more sensitivity at the extremes of the knob's range,.

Example 4: Custom Lookup Table (LUT) Modulation

By defining a custom CURVE in the COMMON section, MOD2 can perform specialized transformations that are not possible with predefined algorithms.

```
//COMMON
CURVE 50, 0, 0, 0.5, 0.2, 0.8, 1.0, 1.0, 0.5
LAYER
OUT = MOD2(INZ, MCC(1), -2, -2, 1, 50)
```

This example uses the main input (`INZ`) and the Modulation Wheel (`MCC(1)`) as inputs,. The final product is passed through the custom transfer function defined in slot 50, which provides a non-monotonic "spike" response as the combined signal level changes,.

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.

```

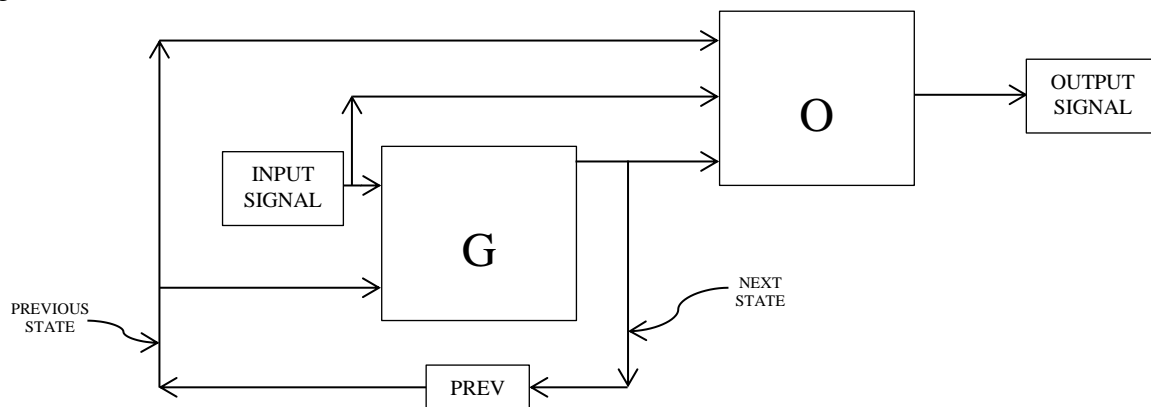
OUT = Y

```

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)

```

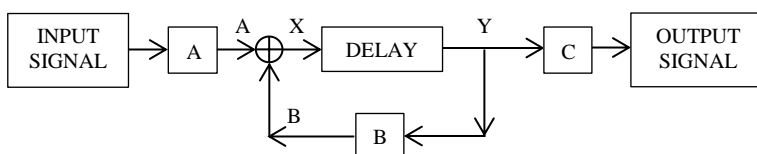
The example given is with a single state variable, a single input and a single output. More complex systems with more state variables require more variables, each for each state variable.

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

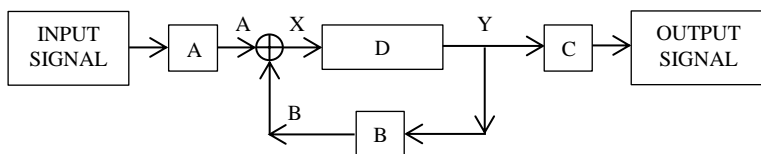
```

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: $Y = \text{SIGN}(X) * \text{ABS}(X) ^ 2$

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: $Y = \text{COPYSIGN}(X^2, X)$

POWABS(<X>,<Y>)

Calculates $\text{SIGN}(\langle X \rangle) * \text{ABS}(\langle X \rangle)^\langle Y \rangle$. Applied to both channels.

Can be used for nonlinear symmetric distortions. E.g. even order distortion: $Y = \text{POWABS}(X, 2)$

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.

LOWBYTE(<op1>)/ HIGHBYTE(<op1>)

Let <op1> be a number between 0 and 1.

LOWBYTE will calculate the low order byte to be put in a MIDI CC: an integer between 0 and 127.

HIBYTE will calculate the high order byte to be put in a MIDI CC: an integer between 0 and 127.

Example: you have a value between 0 and 1 and you want put this in the MIDI CCs 7 and 39. Put HIBYTE(<val>) into the MIDI CC 7 and LOWBYTE(<val>) into the MIDI CC 39.

Mono, uses only the left channel of <op1>.

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.

SINH(<op1>) / COSH(<op1>) / TANH(<op1>)

Hyperbolic trigonometric function of <op1>. Applied to both channels.

ASINH(<op1>) / ACOSH(<op1>) / ATANH(<op1>)

Inverse hyperbolic trigonometric function of <op1>. Applied to both channels.

CHEBY (IN, ORDER, MIX)

The **CHEBY** operator is a harmonic waveshaper based on **Chebyshev polynomials of the first kind**. Unlike standard saturation (**SAT**) or clipping, which generate a broad spectrum of odd and even harmonics, **CHEBY** allows for the targeted excitation of specific harmonic overtones.

Mathematical Principle

The operator uses the recursive definition:

- $T_0(x) = 1$
- $T_1(x) = x$
- $T_{n+1}(x) = 2x T_n(x) - T_{n-1}(x)$

When the input is a normalized sine wave, the output is a pure cosine wave of the N^{th} harmonic. On complex signals (like strings or pads), it acts as a **surgical exciter** that enhances the perceived "edge" or "presence" of the sound without traditional equalization.

Parameters

| Index | Parameter | Description |
|-------|-----------|-------------|
|-------|-----------|-------------|

Index Parameter Description

| | | |
|---|-------|--|
| 0 | IN | Stereo Audio Input. (Signal should be normalized between -1.0 and 1.0). |
| 1 | ORDER | Polynomial Order (1 to 10). Higher values target higher harmonics. |
| 2 | MIX | Dry/Wet Balance. |

Usage Tips

- **Harmonic Reinforcement:** Use Order 2 to add "body" and warmth (octave enrichment) to thin basslines.
- **Metallic Texture:** Use Orders 5 through 8 on your **EKS (Karplus-Strong)** scripts. This simulates the high-frequency reflections of a real metallic wire or a bridge rattle.
- **The "Clean" Distortion:** Because it doesn't "square off" the wave immediately, it sounds much cleaner than `TANH` even at high gain settings.

Stereo Implementation

CHEBY processes both channels of the stereo field independently. Because the internal logic is phase-coherent, it preserves the stereo image of the input signal while enriching the spectral content.

Usage & Performance

- **Harmonic Excitement:** Use a low `mix` (0.1 - 0.3) on pads or leads to add "sheen" and "air" without changing the core timbre.
- **Sub-Harmonic Reinforcement:** Apply **Order 2** to a bassline to ensure it cuts through on smaller speakers by reinforcing the first octave.

Technical Note for Developers

The CHEBY operator utilizes an internal **Clamping logic** (limiting the signal between -1.0 and 1.0) before polynomial calculation. This ensures that even with high input levels or heavy Phase Modulation (FM), the harmonic generation remains stable and musical without digital "wrap-around" artifacts.

Example 1:

```
// Enhance a simple sine lead with a "nasal" 3rd harmonic
LEAD = SIN(FREQ)
OUT = CHEBY(LEAD, 3, 0.5)
```

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. $RESULT = 20 * LOG10 (<op1>)$. Applied to both channels.

DB2LIN(<op1>)

<op1> in dB converted to linear value, i.e. $RESULT = 10 ^ { (<op1> / 20)}$. 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.

FAQs and Tutorials

This section contains some FAQs and tutorials. This FAQ section is designed to address common operational challenges and implementation queries encountered when developing instruments or effects within the Crescendo environment.

The tutorials are full blown examples, explained in detail, ready to be integrated or even used directly as is in your workflow.

System and Installation

Why are there three different DLL files in the package? While all three files (`Crescendo.dll`, `Crescendo-effect.dll`, and `Crescendo-MIDI.dll`) contain identical internal code and functionality, they differ in how they identify themselves to the host DAW via the `isSynth` method. This ensures compatibility with DAWs that strictly enforce instrument/effect classifications or have fixed I/O counts. For example, Audacity only supports the `-effect.dll` version, while Ableton Live requires the standard `.dll` for instrument tracks.

Why are some audio formats not loading into sample slots? Crescendo natively supports uncompressed WAV, AIFF, and AIFF-C files. To support other formats (such as MP3 or video files), you must have `FFMPEG.EXE` located in one of the predefined search paths within your `<My Documents>\Crescendo` directory. If the executable is missing, the plugin will fail to load unsupported formats and log an error.

How do I fix the "Ableton Live 64 parameter limit" when importing SoundFonts? Ableton Live 9.7.5 restricts VST plugins to a maximum of 64 VST variables. When importing a SoundFont that generates the `00000_ALL_INSTRUMENTS_16.txt` file, the variable count may reach 69, causing some parameters to become unreachable in the DAW interface. You should either use the version of the file with fewer variables or manually reduce the `VSTVARS` declaration.

Programming and Logic

What is the difference between `IF...GOTO` and `EXECIF`? `IF...GOTO` is a sample-accurate construct evaluated every single audio sample, allowing for real-time reactions to modulating signals like LFOs or audio inputs. Conversely, `EXECIF` is a trigger-time construct evaluated only once at the moment a MIDI note is struck. `EXECIF` is significantly more CPU-efficient for conditional layering or performance-based switching.

Why do my envelopes or oscillators appear "stuck" when using `HOLD`? The `HOLD` construct calculates the value of a variable only once (at trigger time in a `LAYER` or at start-up in `POST`). Because these variables do not recalculate sample-by-sample, time-dependent functions like envelopes and oscillators cannot progress through their cycles and will remain frozen at their initial values.

How do I implement `AND` or `OR` logic if the engine doesn't support those operators? Crescendo requires a "fail-fast" assembly-style structure for compound logic. To implement an `AND` condition, use sequential `IF` statements that jump to an "end" label if the *inverse* of your condition is met. For `OR` logic, use multiple `IF` statements that all jump to the same "success" label.

How does variable inheritance work between sections? Variables defined in the `COMMON` section are considered global and are "replicated" into the `POST` and `LAYER` sections whenever they are referenced. This means the mathematical formula is shared, but the resulting value may differ based on per-layer or per-channel data, such as `ONVEL` or `MCC` values.

Why isn't my assignment to a MIDI CC sending data to the DAW? Standard assignments (e.g., `MCC (7) = 0.5`) only modify the plugin's local internal storage. To transmit values to the MIDI Output for the DAW or external hardware to see, you must utilize the `MIDIOUT` instruction.

What is the difference between `SEQUENCER` and `SEQUENCER!`? `SEQUENCER!` (Immediate) instructions are executed exactly once during the instrument's compilation stage to set the initial state of the TAPE. `SEQUENCER` instructions are queued into a "Program" that executes at run-time whenever the TAPE cursor reaches the end.

Can I use envelopes to control global effects in the `POST` section? No. The `POST` section does not process `NOTE ON` or `NOTE OFF` messages and therefore cannot utilize envelope functions like `ENV`, `ENV0`, or `ENV1`, which require note-trigger events to function.

List all the randomization options in Crescendo

To create truly "living" instruments, you must move beyond static samples. Crescendo provides a hierarchy of randomization that allows you to choose exactly *when* and *how* the engine rolls the dice.

The Three Layers of Randomization

| Method | Variable/Command | Frequency | Best Used For... |
|------------------|-------------------------------------|-----------------------|--|
| Latched Identity | MCC 139 (RANDOM) | Once per Note-On | Per-note character, stable tuning offsets, or consistent release times. |
| Volatile Chaos | MCC 157 (NOISE) | Every time it is read | Stochastic triggers, unpredictable "rolling" logic, or varied Note-Off behavior. |
| Signal Engine | <code>rnd (max) / rndn (max)</code> | Sample-accurate | Audio-rate noise, FM modulation, or granular glitches. |

Understanding `EXECIF` (Trigger-Time Logic)

The `EXECIF` instruction is a performance-optimized gatekeeper. It is evaluated **only when a MIDI event occurs** (Note-On or Note-Off), making it nearly invisible to the CPU.

Syntax Overview

1. **CC-Based:** EXECIF <mcc>, <min>, <max> <var> = <expression>
2. **Key/Velocity-Based:** EXECIF <minkey>, <maxkey>, <minvel>, <maxvel> <var> = <expression>
3. **Combined:** EXECIF <minkey>, <maxkey>, <minvel>, <maxvel>, <mcc>, <min>, <max> <var> = <expression>

Advanced Practical Examples

Example 1: The "Lucky Note" (Latched Probability)

Use **MCC 139** to decide if a note has a specific feature for its entire life. Because it is latched, the result is consistent at both Note-On and Note-Off.

```
// 20% chance this note will trigger an additional sub-oscillator
// This choice is made at Note-On and remains stable.
LAYER
EXECIF 139, 0.8, 1.0 OUT = OUT + OSCG("sub_sine", 0.5)
```

Example 2: Asymmetrical Note-Off (Volatile Probability)

Use **MCC 157** in **MIDIPOST** to create unpredictable behaviors. Since 157 changes every time it is read, a note might trigger an effect at Note-On but fail the check at Note-Off.

```
// 50% chance to apply a delay, evaluated independently at Note-On and Note-Off
MIDIPOST "EXECIF 157, 0.5, 1.0 DTRIG = 0.5"
```

Example 3: Audio-Rate Glitch (Sample-Time IF)

If you want the randomness to change **while the note is held**, you must use the standard **IF** statement inside a **LAYER** or **POST** block.

```
// Creates high-speed digital "crackling" by dropping the volume
// randomly 44,100/48,000 times per second.
LAYER
IF MCC 157 > 0.98 THEN OUT = 0
```

Example 4: High-Velocity Accent Layer

Coalescing multiple samples into one layer saves polyphony.

```
// Let's suppose that PianoMain contains an oscillator with a piano waveform
LAYER
OUT = PianoMain
// Add Accent Sample only for Velocities 100-127
// Let's suppose tha PianoAccent contains an oscillator with the required sound
EXECIF 0, 127, 100, 127 OUT = OUT + PianoAccent
```

Example 5: Contextual Muting (Sustain Pedal)

```
// Mute notes below C2 (48) only if the Sustain Pedal (CC 64) is pressed
EXECIF 0, 48, 0, 127, 64, 64, 127 OUT = 0
```

Summary: `EXECIF` VS. `IF`

- Use `EXECIF` to set the "DNA" of the note. It is the most efficient way to handle velocity layers, round-robins (via MCC 139), and conditional MIDI routing. It is ignored in the global `POST` step because `POST` lacks a note-specific trigger context.
- Use `IF` only when the logic must respond to real-time changes or create audio-rate modulations.

Developer Note: If an `EXECIF` condition is not met and that instruction was the only one responsible for assigning a value to a variable, that variable will remain `UNDEFINED`. Always ensure a "base" value is assigned before the `EXECIF` if you need a fallback!

Audio Processing, Signal Flow and MIDI

How do I use the first audio input for sidechaining? In Crescendo, `INPUT #1` is accessed via the `INZ` keyword. While it often serves as the primary audio input for effects, it can be used for sidechaining in a MIDI track if the DAW allows it. By default, `INZ` is summed to the `OUT` variable in the `POST` section; to use it purely for sidechaining, you must subtract it from `OUT` at the start of your `POST` logic if you want to apply an effect to `LAYERs`. If you are building a pure effect, in the `POST` step `OUT` at start will contain the first input (`INZ`), so you can just write `OUT = F(... OUT ...)`.

Why does my pitch shifter sound "laggy"? The `PSHIFT` function operates with a fixed block size and introduces a latency of 20ms. For more granular control, `PSHIFT2` allows you to define the block size via the `<Size>` parameter, though this will still introduce latency equal to the specified millisecond value.

What is the "Last Assignment Wins" rule? Within any processing section, the engine executes instructions in the order they are written. For output keywords like `OUTnn` or `SENDS`, only the final assignment made during a single sample's cycle determines the signal sent to the physical output or the next stage.

Why is my input audio (`INZ`) automatically summed to the output? By default, the `OUT` variable in the `POST` section is initialized as the sum of all active layers plus the signal on `INPUT #1` (`INZ`). This is done so that an empty instrument file acts as a simple audio repeater. If you only want the sound of your layers, you must manually subtract `INZ` from `OUT` at the beginning of your `POST` section.

How do I access standard MIDI CCs with 14-bit precision? Standard MIDI CCs (0-127) only provide 7-bit precision. To achieve higher resolution, you must manually compose the value from the MSB and LSB pairs (e.g., CC 7 and CC 39 for volume) using the formula `Volume = (MCC(7) + MCC(39) / 128) / 128`. For VST parameters, the engine supports full 32-bit floating-point precision directly.

Can I apply effects to specific notes in the `POST` section? No. The `POST` section operates on the summed audio of all layers and does not process `NOTE ON` or `NOTE OFF` messages. Note-specific processing, such as per-voice envelopes or filters, must be implemented within the `LAYER` sections.

How can the PREV instruction be used to create custom digital filters? The `PREV` instruction is a specialized functional tool within the Crescendo engine designed to recover the value of a variable at the preceding audio sample. This capability is fundamental for developers who wish to implement any closed-loop system, including both linear and non-linear architectures. By providing access to past states, `PREV` allows for the manual construction of custom digital filters and complex physical modeling algorithms.

The primary use of `PREV` in filter design is the construction of Infinite Impulse Response (IIR) and Finite Impulse Response (FIR) filters. Because `PREV` only accesses the single most recent sample ($n-1$), developers must use a chain of intermediate variables to retrieve samples from further back in time (such as $n-2$ or $n-M$).

To implement a standard filter, the assignments must follow a strict order to ensure the data shifts correctly through the "delay line" of variables:

1. **Shift existing states:** Move the value of $x[n-1]$ to $x[n-2]$, then $x[n]$ to $x[n-1]$.
2. **Define the input:** Capture the current input signal.
3. **Calculate the output:** Use a mathematical expression that combines the current input with the stored previous inputs (for FIR) and previous outputs (for IIR).

For example, a multi-tap IIR filter structure would be programmed as follows:

```
XN2 = PREV(XN1)
XN1 = PREV(INPUT)
INPUT = <current_audio_signal>
YN2 = PREV(YN1)
YN1 = PREV(Y)
Y = <expression_involving_INPUT_XNs_and_YNs>.
```

What else can I create with PREV instruction? `PREV` can be used to implement a classic state machine where the current output is a function of the current input and the previous internal state. This is particularly useful for simulating analog circuits or physical systems where the "next state" of the system is derived from its "previous state".

A specific and advanced application of this logic is the implementation of the **Extended Karplus-Strong (EKS) algorithm**. In this model, `PREV` is used to feed the output of a delay line back into the system, often involving non-linear functions or additional filters within the feedback loop to simulate the vibration of a string or a digital waveguide. See below for an example implementing the EKS algorithm.

Performance and Optimization

How can I reduce CPU usage for instruments with high polyphony? There are several strategies for optimization:

- Use the `NI` (Non-Interpolating) variants for fixed delays or reverbs.
- Utilize "Fast" oscillator variants (`OSCG` format characters `O`, `m`, `M`) which calculate only one phase and amplitude for both channels.
- Apply the `HOLD` construct for parameters that do not need to change during the duration of a note.
- Use the `SILTH` instruction to more aggressively free inactive voice slots.

Why should I use the avx versions of the plugin? The `AVX`, `AVX2`, `AVX512`, `AVX10.1`, and `AVX10.2` versions are compiled to utilize advanced CPU instruction sets that handle SIMD (Single Instruction, Multiple Data) operations. Using the version that matches your hardware allows the processor to handle audio calculations more efficiently, leading to significant performance gains.

How can I stop a layer from consuming CPU after the sound has faded? The `SILTH` (Silence Threshold) instruction allows you to define the point at which a layer is considered inactive and its slot is freed. If a layer has no envelopes or remains above this threshold due to feedback or loops, it will never terminate.

Is there a way to apply filters without real-time CPU cost? Yes. You can use the `FILTER`, `FILTERSEMI`, `EQ1DB`, or `EQ3DB` instructions as prefixes to a `SAMPLE` or `SAMPLES` declaration. This processes the audio data directly in memory during initialization, removing the need for real-time filter calculations during playback.

When should I use the `FEND` keyword? `FEND` is used in the `COMMON` section to ensure specific instructions are appended to the very end of the compiled instrument file. This is useful for forcing settings like `DEBUG 0` or a specific `QUALITY` level that should not be overridden by other included files or logic.

Synthesis and Tuning

Can I have different temperaments for different MIDI channels? No. Crescendo utilizes a single shared temperament system for all 16 MIDI channels within a single plugin instance. To use multiple simultaneous tuning systems, you must load multiple instances of the Crescendo VST.

How do I create a custom oscillator waveform? You can construct custom waveforms by using the `SAMPLE` instruction with synth numeric codes to define up to 32 harmonics with specific types, amplitudes, and phases. Alternatively, you can use the `WAVETABLE` function to treat any audio sample as a periodic waveform.

What happens if I skip a filter with an `IF GOTO` statement? Skipping instructions with internal memory, such as `FILT` or `DELAY`, will cause them to function incorrectly. Because their internal states do not advance while they are skipped, the audio output will exhibit artifacts or "frozen" states when the logic returns to them.

Practical Recipes & Creative Solutions

This section bridges the gap between basic concepts and advanced instrument design. You can think of these as "Modular Recipes": many of the following techniques can be combined—for example, pairing a **MIDI Effect** with a **Layered Instrument** and a **POST-processing** chain to create a professional-grade synthesizer.

Note on Merging: When combining different tutorials into a single script, ensure you maintain the structural order of the sections: **COMMON** first, then **POST**, and finally **LAYERS**.

How do I... Create a Humanized MIDI Performance?

Digital MIDI sequences often feel too robotic because notes start and end with perfect mathematical precision. This "recipe" shows you how to build a **Humanizer** that adds subtle, organic variations to the velocity and timing of incoming MIDI notes.

The Humanizer Logic

- **Velocity Shifting:** Instead of every note hitting with the exact same force, we add a random "spread" to the velocity.
- **Timing Offsets:** We introduce tiny, unpredictable shifts to the start time and duration of each note, mimicking the natural imperfections of a human player.
- **Toggle Switch:** We use a VSTVAR to allow the user to turn the effect ON or OFF instantly from the DAW interface.

The Code

```
// --- COMMON SECTION ---
// Number of VST parameters used
VSTVARS 1

// Parameter 0: Enable or Disable the effect
// 0 = OFF, 1 = ON
VSTVAR 0, 0, "ENABLE", "", 0, 1, 4, "OFF", "ON"

// --- MIDI PROCESSING ---

// 1. Velocity Humanization
// We add a spread of 10 units (out of 127) to the velocity.
// The effect is active only when VST VAR 0 is ON (set to 1).
VELOCITY 600, 0, 127, 0, 127, 0, 127, 1, 10, 0, 1, 1, 1

// 2. Timing Humanization
// We add a 1/16th bar spread to both the START time and the DURATION.
// This prevents the "grid" feel and makes the instrument breathe.
TIMING 600, 0, 127, 0, .0625, .0625, 0
```

Why this works:

- **VELOCITY:** By spreading the velocity by 10, a note played at 100 will actually trigger anywhere between 95 and 105. This is enough to change the timbre of many VST instruments subtly.
- **TIMING:** The value .0625 corresponds to a 1/16th note. Even a small fraction of this value can make a "stiff" quantize feel like a real performance.
- **MODULARITY:** Remember that this is a **MIDI Effect**. You can place this code at the top of your script (in the `COMMON` section) and it will process the notes before they ever reach your `LAYER` instruments.

Developer's Note:

This version uses constants for timing and velocity spreads. In more advanced tutorials, we will see how to make these spreads variable using VST sliders for real-time control.

How do I... Automate MIDI Humanization (Direct vs. Scripted)?

In Crescendo, you can automate MIDI parameters to create "living" performances. You can link VST parameters directly to the Timing and Velocity engines. This recipe shows you the modern, streamlined method and the classic scripted method for deeper control.

Method A: The Direct Mapping

This is the fastest way to work. By simply referencing the index of a VST Parameter (e.g., \$600), the engine automatically scales the humanization spread based on the slider position.

```
// --- COMMON SECTION ---
VSTVARS 2
VSTVAR 0, .1, "VEL.SPREAD", "", 0, 1, 1
VSTVAR 1, .1, "TIME.SPREAD", "", 0, 1, 1

// Modern Direct Syntax:
// Instead of a constant, we use <Index> to link the parameter.
VELOCITY 600, 0, 127, 0, 127, 0, 127, 1, $600, 0, 1, 1, 1
TIMING 600, 0, 127, 0, $601, $601, 0
```

Method B: The Scripted Mapping (MIDIPOST)

Use this method if you need to perform mathematical operations (like ABS or custom scaling) before applying the humanization.

```
// --- COMMON SECTION ---
VSTVARS 2
VSTVAR 0, .1, "VEL.SPREAD", "", 0, 1, 1
VSTVAR 1, .1, "TIME.SPREAD", "", 0, 1, 1

// Reset and Script
MIDIPOST 0
// We use RND(VARx) for custom randomization logic
MIDIPOST "ONVEL = ONVEL + RND(VAR0)"
MIDIPOST "DTRIG = ABS(RND(VAR1))"
MIDIPOST "DDUR = DDUR + RND(VAR2)"
```

Comparison: Which one should I use?

| Feature | Direct Method (\$nnn) | Scripted Method (MIDIPOST) |
|-------------|--------------------------|---|
| Ease of Use | Ultra-fast, single line. | Requires string scripting. |
| Performance | Native and optimized. | Very fast, but more "logic" to parse. |
| Flexibility | Standard spread. | Can use any formula (e.g., VAR0 * 0.5). |

Developer's Note:

The Direct Method (\$nnn) is generally preferred for standard humanization tasks. It keeps your COMMON section clean and is less prone to syntax errors. Use MIDIPOST only when you need 'intelligent' MIDI processing that simple spreads cannot achieve.

How do I... Achieve Real-Time "Elastic" Glide?

Following the logic of high-end professional samplers (like the one found in Ableton Live), Crescendo allows you to modulate the **Glide Time** continuously. This means the speed of the pitch transition is not "locked" when you press the key; it can be automated and changed *during* the slide itself.

The "Continuous" Advantage

Most synthesizers only check the glide knob at the start of a note (**Trigger-Sampled**). Crescendo's **Continuous Automation** (& syntax) allows for:

- **Dynamic Expression:** Speed up the glide as it approaches the target note for a more "urgent" feel.
- **Performance Control:** Link the glide time to a Mod-Wheel or Macro and change the "slur" intensity in real-time.
- **Physics Simulation:** Use formulas to create non-linear pitch curves that standard ADSR-based glides cannot achieve.

The Implementation

To enable continuous control, use the & prefix followed by the VST Variable index in the POLY instruction.

```
// --- COMMON SECTION ---
VSTVARS 1
// VAR 0: Your Glide Speed knob
VSTVAR 0, 0.2, "GLIDE.BASE", "", 0, 1, 1

// POLY Syntax: <polyphony>, <discard>, <mode>, <glidetime>
// Using &600 (VST VAR 0) makes the glide time update EVERY SAMPLE.
POLY 1, 0, 3, &600
```

Advanced: Formula-Driven Glide

If you want to create a glide that isn't just a straight line, you can "calculate" the glide time in the POST section and feed it to a hidden variable.

```
// --- POST SECTION ---
// Here we modulate VAR127 with a formula (e.g., adding jitter or an LFO)
// This value is then used continuously by the engine.
VAR127 = VAR0 * (1.0 + RND(.01))

// In COMMON, you would have used:
// POLY 1, 0, 3, &727 (Linking to VAR127)
```


Pro Tip: Avoiding "Clicky" Transitions

When using very fast Glide Times (near 0) in **RETRIG** or **GLIDE** modes, the pitch change can be so abrupt it sounds like a pop. Using continuous automation allows you to "smooth out" the beginning of a slide by starting with a slightly higher glide value and decreasing it rapidly.

Developer's Note:

This behavior was specifically modeled to match the flexibility of Ableton's sampler engine. It ensures that if you automate your 'Glide' knob in the DAW, the sound reacts instantly, even if a note-slide is already halfway through its journey.

How I do... Achieve "Decaying" or Envelope-Driven Glide?

While global glide is typically controlled by a fixed knob, Crescendo allows you to drive the **Glide Time** using an **Envelope** or any custom expression (even a sample!). This technique creates a "Natural Physics" effect where the glide starts slow and "sluggish" but accelerates as the note transition progresses—mimicking the way a physical instrument (like a trombone or a cello) might behave as it settles into a pitch.

The "Elastic" Advantage

By using an envelope to modulate glide speed within the `LAYER`, you can achieve:

- **The "Magnetic" Snap:** Start with a high glide time (slow movement) and use a decaying envelope to drop the glide time to near-zero. The pitch will start moving slowly and then "snap" suddenly into the target frequency.
- **Rhythmic Gliding:** Use a looped envelope to make the glide speed oscillate, creating a "stuttering" pitch slide that stays in sync with your project.
- **Automatic Smoothing:** Ensure that quick note-on/note-off transitions are always fluid by tying the glide response to the life-cycle of the note itself.

The Implementation

To achieve this, we define a "Modulation Envelope" in the `LAYER` and assign it to a local variable. We then point the `POLY` instruction to that variable using the **& (Continuous)** syntax.

```
// --- COMMON SECTION ---
// Define a base glide range (0.0 to 1.0 seconds)
VSTVAR 0, 0.5, "Max Glide", "s", 0, 2, 1

// POLY Syntax: <polyphony>, <discard>, <mode>, <glidettime>
// We link to VAR100 (&700) which we will calculate in the LAYER
POLY 8, 0, 3, &700

// --- LAYER SECTION ---
LAYER
// 1. Define the Glide Envelope
// This envelope starts at 1.0 (Max) and decays to 0.0 over 0.5 seconds
GlideEnv = ENV0(0.001, -0.5, 0.0, 0.1)

// 2. Map the envelope to our Glide Variable (VAR 100)
```

```
// The VAR100 setting MUST occur before the OSCG function call.
VAR100 = VAR0 * GlideEnv

// 3. Simple Sawtooth for testing the slide
OUT = GAIN * OSCG("y1f0000")
```

Understanding Variable Sharing and Processing Order

It is important to understand that while `VAR100` is technically a global variable shared across the entire engine, Crescendo processes every active layer and every voice in a strict sequence for each sample. See the `POLY` instruction help for details.

Advanced: "Inertia" Glide

You can use `ENVCURVE` to create even more complex "physics." For example, a "Double-Swell" glide where the pitch slide starts fast, slows down in the middle of the transition, and then finishes fast again.

```
// --- COMMON SECTION ---
// Define a "S-Curve" for glide speed in Slot 20
CURVE 20, 0,0, 1,1, 2,0.2, 3,0

// --- LAYER SECTION ---
LAYER
// Use the custom curve to drive the Glide Time
G_Mod = ENVCURVE(0, 20, 0.001, 0.3, 0.001, 0.3, 0.001, 0.1)
VAR100 = VAR0 * G_Mod
OUT = GAIN * OSCG("y1f0000")
```

Developer's Note:

Using an internal variable (like `VAR100`) to bridge an Envelope to the `POLY` instruction is a unique Crescendo workflow. Because the engine evaluates the `'&'` reference at the exact moment each voice is processed, we turn a global parameter into a per-voice modulation destination.

How I do... Create a "Bouncing" Variable Glide

For this second example, we will explore a "**Bouncing Glide**" effect. Instead of a simple decay, we will use the `ENVCURVE` function to create a glide speed that fluctuates—starting fast, slowing down to a crawl in the middle of the transition to "savor" the pitch shift, and then accelerating again to lock into the target note.

In certain synthesizer patches, a linear glide can feel "robotic." By using a custom curve to control the glide time, we can create a non-linear movement that feels like a physical object with inertia.

The "Inertia" Implementation

We will define a custom curve where the value (glide time) represents the "sluggishness" of the pitch change. A high value on the curve means the pitch moves slowly; a low value means it moves quickly.

```
// --- COMMON SECTION ---
// Define a "U-Shaped" glide speed in Slot 30
// x=0: Start fast (Glide time 0.1s)
```

```

// x=0.5: Slow down in the middle (Glide time 1.2s)
// x=1.0: End fast to lock the note (Glide time 0.05s)
// We only need the Attack portion (0 to 1) for this effect
CURVE 30, 0,0.1, 0.5,1.2, 1,0.05, 2,0.05, 3,0

// Link the engine's global Glide Time to VAR100 (&700)
POLY 8, 0, 3, &700

// --- LAYER SECTION ---
LAYER
// 1. Define the Custom Glide Curve
// We use a 0.8s attack time to span the duration of the pitch slide
G_Curve = ENVCURVE(0, 30, 0.001, 0.8, 0.001, 0.1, 0.001, 0.1)

// 2. Assign to the shared Glide Variable
// IMPORTANT: This must be calculated before the OSCG call.
// Although VAR100 is shared, the engine processes each voice
// sequentially. The value set here is used immediately by the
// current voice's pitch calculator for this specific sample.
VAR100 = G_Curve

// 3. Sound Generation
OUT = GAIN * OSCG("y1f0000")

```

How do I... Remap MIDI CCs and Generate 14-bit Output?

This recipe solves a common studio problem: hardware incompatibility. You might have a Foot Pedal (CC 4) but need it to act as a Sustain Pedal (CC 64), or you may want a VST slider to control an external synth using high-resolution 14-bit MIDI.

Using `MIDIOUT` in the `POST` section allows you to intercept and transform the MIDI flux with sample-accurate precision before it leaves the engine.

The Routing Logic

- **Continuous Evaluation:** Even though MIDI is a relatively slow protocol, Crescendo evaluates the `MIDIOUT` expression at **every audio sample**. This is crucial if your expression includes filters, delays, or functions with "memory" that require a continuous clock to function correctly.
- **Delta-Based Emission:** To save bandwidth, a MIDI message is generated **only if the calculated value has changed** since the last check.
- **Refresh Rate (Throttling):** The third parameter (e.g., `.01`) acts as a safeguard. It ensures that even if the value is changing constantly, the MIDI bus isn't flooded faster than the specified interval (e.g., every 10ms).

The Code

```

// --- COMMON SECTION ---
VSTVARS 1
TOOLTIP "Master Volume.", 0
VSTVAR 0, 1, "VOLUME", "", 0, 1, 1

// --- POST SECTION ---

// 1. CC Remapping (Foot Pedal to Sustain)
// The expression "MCC4" is evaluated at every sample.

```

```
// A message is sent to CC 64 only if MCC4 changes,
// and no more than once every 10ms (.01).
MIDIOUT 64, 0, .01, MCC4

// 2. High-Resolution 14-bit MIDI Volume
// We split VAR0 into two bytes to send to CC 7 (MSB) and CC 39 (LSB).
// This provides 16,384 steps of resolution for ultra-smooth sweeps.

// Most Significant Byte (Coarse)
MIDIOUT 7, 0, .01, HIBYTE(VAR0)

// Least Significant Byte (Fine)
MIDIOUT 39, 0, .01, LOWBYTE(VAR0)
```

Why this is essential:

- **Bandwidth Intelligence:** The engine is smart enough to remain silent if the value is static, but it stays "alert" at a sample-accurate level.
- **Advanced Expressions:** Because the value is evaluated at every sample, you could use a `FILT` function on a MIDI CC to "smooth out" a jittery hardware controller before re-sending it.
- **14-bit Precision:** Using `HIBYTE` and `LOWBYTE` eliminates "zipper noise," providing the professional response required for high-end orchestral libraries or external analog hardware.

Developer's Note:

Think of `MIDIOUT` as having a dual-speed engine. The **math** runs at the audio sample rate (e.g., 44.1kHz or 48kHz), but the **MIDI output** is throttled by the refresh rate parameter. This gives you the best of both worlds: the precision of high-quality DSP with the stability of the MIDI protocol.

How do I... Build a Programmed MIDI Step Sequencer?

This recipe demonstrates how to create an internal MIDI sequence that loops automatically. By using the **TAPE** system, you can pre-load a series of notes and have the engine play them back with sample-accurate timing, synced to your DAW's transport.

IMPORTANT: The example below is a **trivial demonstration** designed to show the basic syntax. The Crescendo Sequencer engine supports **dozens of advanced instructions** (including conditional jumps, variables, and real-time transformations). For the full technical specification of the sequencer language, please refer to the **Sequencer Section** in the main manual above.

The Sequencer Logic

- **The TAPE:** Think of this as a virtual strip of player-piano paper. You "write" notes onto it using time offsets (in bars).
- **Initialization (SEQUENCER1):** These commands run only once to set up the sequence. Here, we define a basic arpeggio (C, E, G, B).
- **The Program (SEQUENCER):** These commands run continuously. In this example, we use a simple `RESET` logic: as soon as the sequence reaches the end of the programmed notes, it jumps back to the beginning.

The Code

```
// --- COMMON SECTION ---
VSTVARS 1
// A dummy variable to hold a place in the VST interface
TOOLTIP "Sequencer Control.", 0
VSTVAR 0, 0, "DUMMY", "", 0, 4, 2

// 1. Activate the Sequencer
// "ACTIVATE 0, 10000": Enables the engine with a limit of 10,000 instructions.
// 0, 15: Applies the sequencer to all MIDI channels (0 through 15).
SEQUENCER "ACTIVATE 0, 10000", 0, 15

// 2. Program the "TAPE" (The Note Pattern)
// MOVE 0,0: Start at the beginning of the bar.
// ADD <time>, <note>, <duration>:
// We add an Arpeggio: C3 (60), E3 (64), G3 (67), B3 (71).
SEQUENCER "MOVE 0,0; ADD 0,60,1/32; ADD 1/4,64,1/32; ADD 2/4,67,1/32; ADD
3/4,71,1/32;", 0, 0

// 3. The Runtime Program
// RESET: Tells the sequencer to loop back to time 0 once the notes are
finished.
SEQUENCER "RESET;"
```

Why this is powerful:

- **Perfect Sync:** Because the sequencer uses bar-based timing (e.g., 1/4 for a quarter note), it will always stay perfectly in sync with your DAW's BPM.
- **Complex Rhythms:** You can create intricate polyrhythms by adding notes at unusual time offsets (like 1/7 or 3/11) that would be difficult to draw manually in a DAW.
- **Beyond the Basics:** This trivial example just loops a pattern, but with the full instruction set, you can build **generative algorithms**, **auto-accompaniment systems**, or **algorithmic composers** that react to your playing in real-time.

Developer's Note:

The `SEQUENCER` (Initialization) and `SEQUENCER` (Runtime) commands use a dedicated mini-language within Crescendo. Note the use of the semicolon `;` to separate instructions within the string. While this tutorial is a simple loop, the engine can handle thousands of instructions for complex musical structures.

How do I... Build a Punchy FM Synth Bass?

FM synthesis is famous for its "living" textures. In this recipe, we create a classic FM bass where the "brightness" of the sound decays over time, mimicking the way a real string or skin loses high-frequency energy after being struck.

The FM Architecture

- **The Modulator:** This oscillator isn't heard directly. Its output is sent into the phase of the Carrier. The more "Modulation" we apply, the more harmonically rich (brighter) the sound becomes.

- **The Carrier:** This is the oscillator you actually hear. Its frequency is "warped" by the Modulator.
- **Dynamic Envelopes:** We use two different envelopes. One controls the **Volume** (Amplitude), and the other controls the **Timbre** (Modulation depth), allowing the sound to start bright and snappy and end as a pure, deep sub-tone.

The Code

```
INTERFACE 1000,100
VSTVARS 8

// Parameter Setup
VSTVAR 0, 1, "VOLUME", "", 0, 1, 1
VSTVAR 1, 2, "MODULATION", "", 0, 5, 1
VSTVAR 2, 20, "MODDCY", "", 3, 300, 0 // Modulation Decay
VSTVAR 3, 90, "AMPDCY", "", 3, 300, 0 // Amplitude Decay
VSTVAR 4, .5, "SUSTAIN1", "", 0, 1, 1 // Mid-sustain level
VSTVAR 5, .02, "SUSTAIN2", "", 0, 1, 1 // Final sustain level
VSTVAR 6, .4975, "FREQ.MUL", "", .48, .52, 1 // Subtle detune for character
VSTVAR 7, 1, "RELEASE", "", 0, 3, 1

LAYER
// 1. The Modulator
// skk000S: Sine wave, simple ADSR (S)
// We use MCC(601) for Max Modulation and MCC(602) for the Decay time.
Modulator = OSCG("skk000S", MCC(601), MCC(606), .1, MCC(602), 0, MCC(607))

// 2. The Carrier (The Output)
// skf00PD: Sine wave, Phase Modulation (P), Double Decay Envelope (D)
// This layer takes the "Modulator" signal as its frequency input.
OUT = OSCG("skf00PD", MCC(600), Modulator, .1, .1, MCC(604), MCC(603), MCC(605),
MCC(607))
```

Why this sounds "Pro":

- **The "P" Flag:** In the OSCG string `skf00PD`, the **P** stands for Phase Modulation. This is the secret to clean FM synthesis. It prevents the pitch from drifting while creating those rich harmonics.
- **Double Decay (D):** By using the **D** envelope flag, we create a more complex volume shape: an initial "thump" (Decay 1) followed by a settled body (Decay 2).
- **Frequency Ratios:** Changing `FREQ.MUL` (VAR 6) drastically alters the character. Ratios like 0.5, 1.0, or 2.0 create musical harmonics, while slightly "off" values (like .4975) create the gritty, detuned grit found in modern electronic bass.

Developer's Note:

The `OSCG` function uses a specific string format to configure its behavior. For example, `skf00PD` tells the engine: **s** (Sine), **k** (Key tracking), **f** (Frequency input), **P** (Phase modulation), and **D** (Double decay envelope). Mastering these 'flags' is the key to unlocking Crescendo's synthesis engine.

How do I... Understand a Sample-Based Instrument?

While Crescendo is a powerful synthesizer, it is also a sophisticated **multisample engine**. The most efficient way to learn how to build a high-quality sampled instrument is to use the built-in importer to convert a **SoundFont (.sf2)** file and examine the resulting code.

Below is an anatomy of a "Carillon" instrument converted from a SoundFont.

The Anatomy of a Sampled Instrument

- **The SAMPLE Definition:** This links a unique ID (e.g., 8403) to a physical `.wav` file and defines its root pitch, loop points, and playback behavior.
- **Global Controls:** Notice the `POST` section. It contains a shared **Reverb** and a **Dry/Wet** mixer that affects all layers simultaneously.
- **Layer Splitting:** The instrument uses `ONNOTEON` to divide the keyboard. Different ranges (0–89, 90–96, 97–127) trigger different logic, though they might share the same sample ID.
- **Complex Modulation:** Sampled instruments often need pitch modulation (LFO) and gain adjustments based on MIDI CCs (like Sustain or Sostenuto pedals).

The Code (Analysis)

```
// 1. Register the Sample
// ID: 8403, Path: samples\08403_TUBLR84.wav
// The numbers following define root key, loop start, loop end, etc.
SAMPLE
8403,"samples\08403_TUBLR84.wav",0,84.000000,0,0,0,1244,1e10,294,1166,5,5,5

// 2. Global FX (Reverb Mixer)
POST
IF MCC(601) < .1 EXIT // Skip if Dry/Wet is near 0
OUT = .01 * ((100 - MCC(601)) * OUT + REVERB0(OUT, MCC(600)) * MCC(601))

// 3. Instrument Layers
LAYER
ONNOTEON 0, 89, 0, 127 // Lowest note range
// OSCG Flag "S" = Sample Playback mode
// This uses Sample 8403 and applies an LFO (MODLFO) to the pitch
MODLFO = OSCG("y1v000F", 2, .5, 5.20138, 0, 0.2670163, 0, 0, 0, 1, 1e6)
OUT = OSCG("Sdc000S", 8403, 7.4 + -0.1 * MODLFO, -5.00 + 1 * MODLFO, 0.0009766,
19.9963760, 0.0000000, 4.9990940)
OUT = PAN(OUT, -100.0)
```

Why this structure is effective:

- **sdc000s Flag:** The first letter `s` in the `OSCG` string tells the engine to act as a **Sampler** using the data defined in the `SAMPLE` instruction.
- **Global MIDI Handling:** Instructions like `PEDAL 64` (Sustain) and `SOSTENUTO 66` are defined in the `COMMON` section so they automatically manage the note-off logic for all layers.
- **Optimization:** The `IF MCC(601) < .1 EXIT` in the `POST` section is a clever trick to save CPU—if the reverb isn't being used, the engine doesn't even calculate it.

Designer's Note:

If you want to create your own sampled instruments, don't start from scratch! Take a simple `.sf2` file, convert it, and use the generated code as a template. You can then swap the `.wav` paths for your own high-quality recordings.

How do I... Build a Sidechain Ring Modulator?

Sidechaining isn't just for the "pumping" effect in EDM; it is a gateway to advanced audio manipulation. A **Ring Modulator** mathematically multiplies two signals: the main signal (Carrier) and an external signal (Modulator). The result is the creation of "sidebands"—new frequencies that give the sound a metallic, robotic, or bell-like character.

The Sidechain Logic

- **External Input (IN0):** Crescendo can receive audio from another channel in your DAW. In this script, `IN0` represents the first channel of the sidechain input.
- **Sample-Accurate Multiplication:** By multiplying the current signal (`OUT`) by the external input (`IN0`), the amplitude of the first is instantaneously scaled by the value of the second at the audio sample rate.
- **Post-Processing:** Since this is an effect that acts upon an existing audio signal, the code resides in the `POST` section.

The Code

```
// --- COMMON SECTION ---
INTERFACE 800,100
VSTVARS 1

// Parameter 0: Master Volume to control the effect output
VSTVAR 0, 1, "VOLUME", "", 0, 1, 1

// --- POST SECTION ---
// The POST section is evaluated for every audio sample.

// Ring Modulation Formula:
// OUT (current signal) is multiplied by IN0 (Sidechain Input).
// We also apply the Volume variable (MCC 600) to the result.

OUT = MCC(600) * OUT * IN0
```

Why this works:

- **Precision:** The expression is evaluated at the audio sample rate. If you send a 440Hz sine wave into the sidechain, you get a perfect 440Hz modulation, not just a slow volume automation.
- **Versatility:** If you send a clean tone into the sidechain, you get a classic Ring Modulator. If you send a rhythmic signal (like a kick drum), you get a syncopated gating or "tremolo" effect.
- **INZ vs OUT:** At the very beginning of the `POST` section, `OUT` and `INZ` (the raw input entering the plugin) are identical. Using `OUT` in this first instruction is a fast and efficient way to process the incoming signal.

Developer's Note:

To use this effect, ensure your DAW's routing is set up correctly: you must send an audio signal to Crescendo's 'Sidechain Input.' Without a signal in `IN0`, the result of the multiplication will be zero, and you will hear nothing!

How do I... Create a Synchronized Ping-Pong Delay?

A Ping-Pong delay isn't just a simple echo; it bounces the delayed signal between the left and right speakers. This recipe features **DAW-sync timing** (bars) and **dual-band feedback**, allowing you to control the decay of low and high frequencies independently for a more professional, "expensive" sound.

The Delay Logic

- **Dynamic Feedback:** We use two feedback coefficients (`LFEEDBACK` and `HFEEDBACK`) separated by a frequency threshold (`F_Thres`). This allows you to have deep, long-lasting bass echoes while keeping high-frequency "shimmer" under control.
- **Bar-Based Sync:** Instead of milliseconds, the delay time is defined in **BARS**. This ensures the echoes always land perfectly on the beat, even if you change your project's BPM.
- **CPU Optimization:** We use an `IF` statement to exit the `POST` section entirely if the effect is dialed to zero, saving precious processing power.

The Code

```
// --- COMMON SECTION ---
INTERFACE 800,100
VSTVARS 5

VSTVAR 0, 50, "Dry/Wet", "%", 0, 100, 1
VSTVAR 1, .8, "LFEEDBACK", "", 0, .9999, 1 // Low freq feedback
VSTVAR 2, .5, "HFEEDBACK", "", 0, .9999, 1 // High freq feedback
VSTVAR 3, 5000, "F_Thres", "Hz", 10, 20000, 0
VSTVAR 4, .25, "DELAY", "BARS", .03125, 5, 3

// --- POST SECTION ---

// 1. Efficiency Check
// If Dry/Wet is below 0.1%, we stop processing immediately to save CPU.
IF MCC(600) < .1 EXIT

// 2. The Ping-Pong Logic
// PPDELAY Syntax: (input, delay_time, low_feedback, high_feedback, threshold)
// Note the negative sign on MCC(604):
// A negative value tells the engine the delay is in BARS, not seconds.

OUT = .01 * ((100 - MCC(600)) * OUT + PPDELAY(OUT, -MCC(604), MCC(601),
MCC(602), MCC(603)) * MCC(600))
```

Why this works:

- **The "Negative Time" Trick:** In the `PPDELAY` function, using `-MCC(604)` is the secret to BPM synchronization. If the value were positive, the engine would interpret it as seconds.
- **Stereo Movement:** The `PPDELAY` function automatically handles the internal phase and routing to ensure the first tap starts on one side and subsequent echoes bounce to the other.

- **Dual-Band Feedback:** By setting `LFEEDBACK` higher than `HFEEDBACK`, you create an "analog-style" delay where the high-end rolls off over time, making the echoes feel warmer and less cluttered in the mix.

Developer's Note:

The `PPDELAY` is a highly optimized internal primitive. Notice how the final `OUT` line is a standard **Linear Interpolation** formula: it balances the clean (Dry) signal and the processed (Wet) signal based on the percentage from VST Variable 0.

How do I... Build a Professional MPE-Ready Instrument?

This recipe demonstrates how to handle the "Master" and "Member" channel logic required for MPE. We will sum global expressions (from the Master channel) with per-note expressions (from Member channels) to create a truly expressive synthesizer.

The MPE Architecture

- **The Master Channel (0):** Controls global parameters like the Sustain Pedal or global Pitch Bend that affects all currently playing notes.
- **The Member Channels (1-15):** Each new note is assigned its own channel, allowing for independent Aftertouch and "Timbre" (CC 74).
- **Sample-and-Hold Expression:** We use `SHOLD(..., -1)` to ensure that expression data is sampled continuously until the note enters the release stage, preventing "ghost" modulations after a key is lifted.

The Code

```
// --- COMMON SECTION ---
VSTVARS 3

// 1. Global Pedal Setup
// Master Channel (0) controls the VST variables for all notes.
VSTVAR 0, 0, "HOLD",      "", 0, 1, 4, "OFF", "ON"
VSTVAR 1, 0, "SOSTENUTO", "", 0, 1, 4, "OFF", "ON"

MIDICC 64, 0, 0    // Link MIDI CC 64 on Channel 0 to VST VAR 0
MIDICC 66, 0, 1    // Link MIDI CC 66 on Channel 0 to VST VAR 1
PEDAL 600          // Use VST VAR 0 (600) as the Sustain engine
SOSTENUTO 601       // Use VST VAR 1 (601) as the Sostenuto engine

// 2. Expression Summing Logic
// We combine Master Channel data (MCC3) with per-note data (MCC2/SHOLD).

// VOLUME: Master Volume * Member Volume
MASTER = MCC3(7,0) * SHOLD(MCC2(7), -1)

// PITCH BEND: 2 semitones on Master + 48 semitones on Member
BEND = 2 * MCC3(135,0) + 48 * SHOLD(PBEND, -1)

// AFTERTOUCH: Master pressure + Per-note pressure
AFT = MCC3(133,0) + SHOLD(AFTERTOUCH, -1)

// TIMBRE (CC 74): Global brightness + Per-note brightness
```

```

TIMBRE = MCC3(74,0) + SHOLD(MCC2(74), -1)

// --- LAYER SECTION ---
LAYER
// Apply the summed Pitch Bend to the base frequency
F = Cents(FREQ, BEND)

// Combine Master Volume and Aftertouch for the final Gain
G = MASTER + AFT

// Generate a Sawtooth wave (y) with Keytracking (k), Velocity (v), and Loop (L)
O = OSCG("ykvL000", 2, 0.9, G, F)

// 12dB/oct LP Filter where Timbre (0 to 2) modulates the Cutoff (0 to 4 * F)
OUT = FILT(0, O, 2 * F * TIMBRE, 0.7)

```

Why this is "Pro" Logic:

- **SHOLD(..., -1):** This is the secret for MPE stability. It tells the engine: "Listen to the expression data while the note is held, but the moment the note is released, keep the last known value." This prevents the sound from jumping or glitching during the release tail.
- **Cents(FREQ, BEND):** This helper function calculates the exact frequency shift based on the BEND variable we created. It handles the logarithmic math of pitch so you don't have to.
- **Dual-Channel Summing:** By summing MCC3 (Master) and SHOLD (Member), your instrument responds correctly to both global DAW automation and individual finger movements on MPE controllers like the Seaboard or LinnStrument.

Developer's Note:

MPE can be CPU-intensive because every note is effectively its own mini-synth with its own logic. Using `COMMON` to define the expressions once, then referencing them in the `LAYER`, is the most efficient way to keep your instrument performing smoothly even with high polyphony.

How do I... Simulate a Plucked String (EKS Algorithm)?

The **Extended Karplus-Strong (EKS)** algorithm is the gold standard for simulating instruments like guitars, harps, or harpsichords. Instead of using a repeating waveform, it excites a "virtual string" (a high-speed delay loop) with a short burst of energy.

The Physics of the Script

- **The Exciter (The Pluck):** We use two layers to simulate the strike. A tonal pulse (**SINC**) provides the pitch definition, while a noise burst (**RND**) simulates the mechanical "scratch" of a plectrum or finger.
- **The Resonator (The String):** The `DELAYF` function creates the feedback loop. The string's length is mathematically tied to the note's frequency.
- **The Damper (The Material):** Real strings lose high frequencies faster than low ones. The `FILT` function inside the delay loop simulates this natural decay, making the sound "warm" and realistic over time.

The Code

```

INTERFACE 800,100
DEBUG 1
VSTVARS 2

VSTVAR 0, 1, "VOLUME", "", 0, 1, 1
VSTVAR 1, 1, "SLOT", "", 0, 1, 2

// Period calculation: sets the "length" of the virtual string
// This is the core mathematical relationship for physical modeling.
A0 = .5 / FREQ

// --- LAYER 1: The Tonal Pluck ---
// Uses a SINC pulse for a clear, defined fundamental pitch.
LAYER
ONMCCT 601, 0, 0
OUT = MCC(600) * GAIN * DELAYF(FILT(-1, SINC(FREQ), FREQ, 0), A0, -1 + A0) *
ENV0(.01, .1, 1, 10)

// --- LAYER 2: The Noise Pluck ---
// Uses filtered White Noise (RND) to simulate the "scratch" of a plectrum.
LAYER
ONMCCT 601, 1, 1
OUT = MCC(600) * GAIN * DELAYF(FILT(-1, RND(1), FREQ, 0) * ENV0(.1, .1, 0, .1),
A0, -1 + A0) * ENV0(.01, .1, 1, 10)

```

Key Parameters for Sound Design

1. **The "Pick" Intensity:** In Layer 2, the `ENV0` attached to `RND(1)` controls the duration of the noise burst. A very short burst (e.g., `.05`) sounds like a sharp plastic pick; a longer burst sounds like a soft thumb pluck.
2. **String Material:** The `FILT` function inside the loop acts as the string's physical dampener. Lowering the filter frequency makes the string sound like nylon or gut; raising it makes it sound like bright steel.
3. **Dynamic Expression:** Because `GAIN` is pre-mapped to MIDI Velocity, the "pluck" will naturally sound harder or softer depending on how you play.

Designer's Note: Mathematical Tuning

The formula $A0 = 0.5 / \text{FREQ}$ is critical. It ensures that the delay buffer length matches the period of the desired note. Because `DELAYF` uses **fractional interpolation**, the string remains perfectly in tune even at very high frequencies where standard integer delays would fail and produce 'out-of-tune' notes.

How do I ... Play in the Dorian Scale

This is a perfect example of how the Crescendo engine acts as a "musical assistant." If you've ever wanted to capture that haunting, medieval sound of *Scarborough Fair* or *Greensleeves* without memorizing complex scales, here is how you build a **Dorian Performance Environment**.

The **Dorian Mode** is a minor-type scale, but it has a "secret ingredient": a **Major 6th**. This gives it a bright, hopeful lift compared to a standard, sad minor scale. In the key of **C**, that "secret note" is **A natural**.

This is a perfect example of how the Crescendo engine acts as a "musical assistant." If you've ever wanted to capture that haunting, medieval sound of *Scarborough Fair* or *Greensleeves* without memorizing complex scales, here is how you build a **Dorian Performance Environment**.

Step 1: The "Safety Net" (Scale Mapping)

The first thing we do is tell the engine to ignore any notes that aren't Dorian. If you hit a "wrong" note (like E natural or B natural), the engine will instantly slide it to the nearest "right" note (**Eb** or **Bb**).

The Code:

```
// C Dorian Map: C, D, Eb, F, G, A, Bb
// Forces every accidental to stay within the mode
MAP -1, 0, 127, 0, 0, 2, 3, 3, 5, 5, 7, 7, 9, 10, 10
```

Step 2: The "Instant Folk" Accompaniment

Traditional Dorian tunes often use a **Drone**—a constant, ringing harmony. We'll program the left side of your keyboard to play a "Power Chord" (Root and 5th) automatically whenever you press one key.

The Code:

```
// Generate a 3-note drone for the left hand (Notes 0-59)
CHORD -1, 0, 59, 0,1, 7,0.8, 12,0.7
```

Step 3: Humanizing the "Pluck"

To move away from a "computer" sound, we add a tiny bit of random timing. This simulates a real musician whose fingers don't hit the keys at the exact same millisecond every time.

The Code:

```
// Add a 10ms window of randomness to note onsets and 5ms to note duration
TIMING -1, 0, 127, 0, 0.01, 0.005
```

Step 4: The Final Performance Script

Combine these in your **COMMON** section. Now, any sound you trigger in the **LAYER** section will be "Dorian-ready."

```
// --- COMMON SECTION ---
// 1. Setup the Harmony
CHORD -1, 0, 59, 0,1, 7,0.8, 12,0.7

// 2. Lock the Scale
MAP -1, 0, 127, 0, 0, 2, 3, 3, 5, 5, 7, 7, 9, 10, 10

// 3. Add the Human Touch
TIMING -1, 0, 127, 0, 0.01, 0.005
```

```
// 4. Ensure no "Double Notes"  
// (Assumes CHOKE is enabled in engine settings)
```

How to Play it:

1. **Left Hand:** Hold down a single **C** in the lower octaves. You will hear a rich, ringing drone.
2. **Right Hand:** Play your melody. Focus on the **A natural** (the white key) and the **Eb** (the black key).
3. **The Magic:** Even if you "mess up" and hit an E natural, the `MAP` ensures it comes out as a perfect Eb. You literally cannot play a wrong note!

Developer's Tip

If you want to play *Scarborough Fair* in a different key (like D Dorian, the most common key for that song), we can simply adjust the `PITCH` instruction in the next step of the pipeline.

How do I... Create Instant Stereo Width (Haas Effect)?

The Haas Effect (or Precedence Effect) is a trick used to create massive stereo width. By delaying one channel (usually the Right) by a very small amount—typically **10ms to 25ms**—the brain perceives the sound as coming from a much wider space, extending far beyond the physical position of the speakers.

In Crescendo, we don't need complex routing or dual-tracking. We use **Stereo Parameters** to apply the delay to only one side of the signal in a single, elegant line of code.

The "Single-Line" Haas Logic

- **The JOIN Function:** We use `JOIN(Left_Value, Right_Value)` to provide different parameters to the Left and Right channels simultaneously.
- **The Delay:** By setting the Left delay to 0 and the Right delay to `HAAS_TIME`, we offset the stereo image instantly.
- **The Sweet Spot:** A delay of **0.015s (15ms)** is the gold standard for achieving maximum width without the listener hearing a distinct, distracting echo.

The Code

```
// --- POST SECTION ---  
  
// 1. Define the Haas Delay (15 milliseconds)  
HAAS_TIME = 0.015  
  
// 2. The Haas Command  
// We apply 0 delay to the Left and HAAS_TIME to the Right.  
// The JOIN function directs these values to their respective channels.  
OUT = DELAY(OUT, JOIN(0, HAAS_TIME))
```

Why this is "The Crescendo Way":

- **Efficiency:** There is no need to manually separate `LEFT_CH` and `RIGHT_CH`. The engine handles the stereo interleaving automatically within the `JOIN` function.
- **Phase Stability:** Because the Left channel is untouched (0 delay), the transient punch of your sound remains centered and sharp, while the "perceived body" of the sound expands outwards.
- **Dynamic Width:** You can easily link this to a knob. For example: `OUT = DELAY(OUT, JOIN(0, VAR7 * 0.03))` allows the user to dial in anything from 0 to 30ms of width in real-time.

Designer's Note: Mono Compatibility

While the Haas effect sounds amazing in stereo, it can cause **comb filtering** (a thin, hollow sound) if the two channels are ever summed back to Mono. Always check your mix in mono to ensure that the 15ms delay doesn't cancel out important frequencies in your sound's fundamental tone.

How do I... Modulate Delay for Chorus and Vibrato?

In Crescendo, time and pitch are two sides of the same coin. When you modulate a delay time using a formula, you are performing a sample-accurate recalculation of the reading position in the audio buffer. This is functionally a form of **Frequency Modulation (FM)**.

By dynamically changing the reading speed relative to the output clock, you create perceived shifts in pitch. Slow modulations result in the lush, ensemble sounds of **Chorus**, while faster modulations lead to complex, metallic sidebands.

The Requirement for Interpolation

To achieve these effects without "zipper noise" or clicks, you must use **Interpolating Delay Functions** (`DELAY`, `DELAYF`, `DELAYFF`). These functions use linear interpolation to calculate values between samples as the reading pointer moves.

Note: Non-interpolating functions like `NIDELAY` are optimized for fixed offsets and will produce audible artifacts if modulated.

Implementation: Manual Logic vs. The `CHORUS` Instruction

Method 1: Manual "True" Chorus

This method allows you to see the underlying "physics." We use an LFO to wobble the delay time between 15ms and 25ms.

```
// --- POST SECTION ---

// 1. Create a Sine LFO oscillating at 0.8 Hz
MOD_SIGNAL = OSCG("slv0000", 0.8)

// 2. Calculate the varying delay time (Base 20ms +/- 5ms)
VARYING_DELAY = 0.02 + (0.005 * MOD_SIGNAL)

// 3. Apply the interpolating delay to the Input (INZ)
WET_SIGNAL = DELAY(INZ, VARYING_DELAY)
```

```
// 4. Manual Mix: 50% Dry and 50% Wet
OUT = (0.5 * INZ) + (0.5 * WET_SIGNAL)
```

Method 2: The High-Fidelity CHORUS Instruction

The built-in CHORUS instruction is optimized for richness. It uses multiple voices and internal phase offsets to thicken the sound automatically, including an integrated dry/wet mixer.

Signature: CHORUS(N, IN, DMIN, DDELTA, FLFO, MIX)

```
// N: Number of voices (e.g., 3)
// IN: Input signal (INZ)
// DMIN: Base delay (0.02s)
// DDELTA: Modulation depth (0.005s)
// FLFO: LFO Rate in Hz (0.8Hz)
// MIX: Dry/Wet balance (0.5 = 50% mix)

OUT = CHORUS(3, INZ, 0.02, 0.005, 0.8, 0.5)
```

Classification of Modulation Effects

By adjusting the base delay and the modulation speed, you can transform the same piece of code into different classic effects:

| Effect | Base Delay (DMIN) | Mod Depth (DDELTA) | Mix |
|-----------------|-------------------|--------------------|-----------------------------|
| Vibrato | 5ms - 20ms | 2ms - 5ms | 1.0 (100% Wet) |
| Chorus | 15ms - 35ms | 5ms - 10ms | 0.5 (Balanced) |
| Flanging | 1ms - 5ms | 1ms - 3ms | 0.5 (With Feedback*) |

Why this is "The Crescendo Way":

- **No MAXDLY Required:** Unlike other engines, Crescendo provides a massive **40-second buffer by default**, giving you total freedom to experiment with long, modulated delays without worrying about memory allocation.
- **Integrated Mixing:** The MIX parameter in the CHORUS instruction ensures phase-coherent blending between your dry input and the modulated voices.
- **Sample-Accurate Evaluation:** Since the POST section runs at the audio rate, your chorus remains smooth and artifact-free even with high-frequency modulators.

Developer's Note:

While the `CHORUS` instruction is the fastest way to get a professional sound, building it manually with `DELAY` allows you to inject custom logic—like using different dephasing between the delayed signals.

How I do... Emulate a 1970s "Muted Trumpet" Guitar Effect

In the 1970s, guitarists often used fixed-filter banks or "parked" wah-wah pedals to transform the guitar's natural sustain into a nasal, brassy "honk." This effect, famously produced by hardware like the *Colorsound Dipthomite*, makes an electric guitar sound remarkably like a muted trumpet.

To recreate this in **Crescendo**, we combine a non-linear **Moog Band-Pass filter** with a specialized **Spring Reverb** configuration using the `REVERB3` engine.

1. The Core Components

A. The Brassy "Honk" (MOOGG)

A muted trumpet has a very narrow frequency peak. We use the `MOOGG` engine with **Filter Code 8 (Band-Pass)**. Unlike standard filters, the Moog model adds non-linear saturation.

- **Fixed Frequency:** In the '70s, this was a static hardware setting. We set it between **1000Hz and 1200Hz**.
- **High Resonance:** Setting resonance near **0.9** creates the metallic "ring" of a physical trumpet mute.
- **Automated Drive:** By automating the **Drive** parameter, you simulate the "overblown" character of a horn, adding harmonics as the input intensity increases.

B. The Vintage Space (REVERB3)

Spring reverbs are mechanical, not algorithmic. They have specific decay times and a "clanky" high-end damping.

- **Reverb Time (RT60):** A classic 1970s "Long Decay" spring tank (like an Accutronics Type 3) lasts between **2.75 to 4.0 seconds**. In **Crescendo**, this is achieved with a feedback coefficient (`LPROOM`) of approximately **0.92**.
- **Damping:** Springs lose high-frequency energy very quickly. We set `HPROOM` much lower than `LPROOM` to "darken" the tail.

2. The Implementation Script

Copy the following code into your instrument file. This script uses `INZ` to process live guitar input through the `POST` section. If you want to apply the effect to a virtual instrument, use `OUT` and put this code on the `POST` section of the virtual instrument.

```
// --- COMMON SECTION ---
// VST Variables for live tweaking and DAW automation
VSTVAR 0, 1150, "Mute Freq", "Hz", 600, 1800, 1 // The fixed "Trumpet" sweet spot
VSTVAR 1, 0.92, "Resonance", "", 0.1, 0.98, 1 // Metallic "nasality"
```

```
VSTVAR 2, 1.5, "Drive", "Saturation", 1, 5, 1 // Automate this for
"overblown" tones
VSTVAR 3, 0.4, "Spring Mix", "", 0, 1, 1 // Reverb wet/dry balance

// --- POST SECTION ---
POST
// 1. Capture the dry guitar signal
Guitar = INZ

// 2. The Trumpet Filter (Code 8 = 12dB Band-Pass)
// We use a fixed freq (VAR 600) and automated drive (VAR 602)
Tone = MOOGG(8, Guitar, VAR(600), VAR(601), VAR(602))

// 3. The Spring Reverb (REVERB3)
// Params: (input, LPROOM, HPROOM, FC, STEREO, FIRST_DELAY, NUM_DLY, FIRST_AP,
NUM_AP, G_AP)
// LPROOM 0.92 = ~3.0s decay (Classic 70s Long Spring)
// HPROOM 0.45 = Quick high-end damping for mechanical spring feel
// G_AP 0.7 = Slightly metallic diffusion "clank"
Spring = REVERB3(Tone, 0.92, 0.45, 1200, 2, 40, 8, 20, 4, 0.7)

// 4. Final Mix
OUT = Tone + (VAR(603) * Spring)
```

3. Tuning Your "Trumpet"

To get the most authentic results, follow these testing steps:

- **Finding the "Sweet Spot":** Play your guitar and slowly sweep **VAR 0 (Mute Freq)**. Look for the frequency where the "nasal" quality is most prominent—usually around **1100Hz**. Once found, leave it fixed.
- **Adjusting Decay Time:** * If the reverb is too short: Increase **LPROOM** (0.92 → 0.94).
 - If the reverb is too "digital/clean": Decrease **HPROOM** (0.45 → 0.35) or increase **G_AP** (0.7 → 0.8) to add more metallic ringing.
- **Dynamics:** Automate the **Drive (VAR 2)** in your DAW. Increase the drive during solo sections to make the "trumpet" sound like it's being played with more air pressure, resulting in a richer, grittier texture.

Technical Summary

| Parameter | Value | 70s Hardware Equivalent |
|-------------|---|---------------------------------|
| Filter Type | MOOGG BP (8) Fixed Inductor Filter Bank | |
| Frequency | 1150 Hz | "Parked" Wah Position |
| LPROOM | 0.92 | ~3.0s RT60 (Accutronics Type 3) |
| HPROOM | 0.45 | Mechanical Spring Damping |

| Parameter | Value | 70s Hardware Equivalent |
|-------------------|-------|-------------------------------|
| G_ALL_PASS | 0.70 | Spring Tank Diffusion "Clank" |

How I do... Create a Phase Distortion (PD) Synth

Phase Distortion synthesis (famously used in the Casio CZ series) works by taking a standard waveform—usually a sawtooth—and "warping" its phase before it reaches the output. By distorting the phase ramp, you can transform a simple sine or sawtooth into complex, harmonically rich textures that sound like they are passing through a resonant filter.

Why use WAVETABLES for this?

As established in the `WAVETABLE` series help section, calculating phase using `FRAC (TIME * FREQ)` is prone to jitter. For a PD synth, precision is even more critical because the modulation happens at the audio rate. By using one `OSCG` to modulate the phase of another, we maintain perfect sample-accurate synchronization.

The Implementation

In this example, we use a "Modulator" oscillator to warp the "Carrier" oscillator. The Carrier then indexes a Wavetable.

```
// --- COMMON SECTION ---
VSTVARS 1
VSTVAR 0, 0.5, "PD Amount", "", 0, 1, 1 // Controls the "brilliance" or
distortion

// --- LAYER SECTION ---
LAYER
// 1. The Modulator: A Sine wave to warp the phase
// Frequency: same of the main oscillator.
// We use 's' (sine) and 'C' (automated amplitude, VST VAR 0)
Mod = OSCG("sCf0000", VAR(600))

// 2. The Carrier: A Sawtooth with modulated Phase
// flag 'p' enables Phase Modulation (the 6th argument)
// 2 = asymmetric triangle, 1 = ascending Sawtooth ramp
// We use the 'Mod' signal to distort the 'PhaseRamp'
PhaseRamp = OSCG("y1f00p", 2, 1, Mod)

// 3. Index the Wavetable (Slot 200)
// Using WAVETABLES ensures we use the -1 to 1 range of the Sawtooth perfectly.
PDSignal = WAVETABLES(200, PhaseRamp)

// 4. Final Output with Envelope
OUT = PDSignal * GAIN * ENV0(0.01, 0.3, 0.4, 0.5)
```

Technical Deep Dive

1. The 'p' Flag

In the `OSCG` function string `"y1f00p"`, the `p` at the end tells the engine to accept a **Phase Modulation** input. This input is added to the internal phase accumulator *before* the waveform is generated.

2. Why this sounds unique

When the `PD Amount` (`VAR 0`) is at 0, the `PhaseRamp` is a perfect linear sawtooth, and the wavetable plays back normally. As you increase the `PD Amount`, the `Mod` signal pushes and pulls the phase, causing the wavetable to speed up and slow down within a single cycle. This creates "phantom" resonant peaks that mimic the sound of a hard-sync synth or a sweeping filter.

3. Precision and the POST Section

Because this logic uses `OSCG` internal accumulators rather than global `TIME`, you can move this entire block to the **POST** section to create a "Phase Distortion Effect" for incoming audio:

- Replace the `Mod` oscillator with `IN0` (audio input).
- The audio input will now "warp" the playback of the wavetable in Slot 200.

Advanced Topic: Recursive Stereo Self-Modulation

These examples use a **SuperSaw** as a signal generator to demonstrate the power of recursive calculation, but the logic within the **POST** block functions as a **universal effect processor**. You can drop this code into any preset to radically transform the timbre of any source—including external audio.

The secret behind these algorithms is **Zero-Latency Feedback (Z-1)**: we use the previous output sample to modulate the effect's parameters in real-time, swapping the Left and Right channels for an ultra-wide, organic stereo field.

1. The Spectral "Chewer" (32-Stage Recursive Phaser)

This transforms the signal into a dense, "masticating" mass. The waveform itself moves the 32 phaser stages at audio-rate.

- **Effect:** The phaser's center frequency (`F_MOD`) vibrates according to the previous sample's output.
- **Result:** A liquid, evolving sound that maintains extreme resonance (0.7) without ever clipping, staying rock-solid at **-0.9 dB**.

```
INTERFACE 1200,100
VSTVARS 10
VSTVAR 0,0.5,"VOLUME","",0,1,1
VSTVAR 1,.1,"ATTACK","s",.001,10,0
VSTVAR 2,.1,"DECAY","s",.001,10,0
VSTVAR 3,1,"SUSTAIN","",0,1,1
VSTVAR 4,10,"RELEASE","s",.01,100,0
```

```

VSTVAR 5,0.001,"DETUNE","",.0001,.02,0
VSTVAR 6,0.1,"MIX","",.001,1,0
VSTVAR 7,3,"N","",0,15,2
VSTVAR 8,1,"FILTER?","",0,1,4,"No","Yes"
VSTVAR 9,.5,"Dry/Wet","",0,1,1

GAINENV &600,0,&601,0,&602,&603,&604

POST
// Cross-channel modulation: Left moves Right, and vice versa
F_MOD = 440 + (WIDE(PREV(O), -1) * 100)
O = PHASER(32, OUT, 8, 0, F_MOD, 1, 0.7, VAR9, 0)
OUT = O

LAYER
OUT = GAIN * SUPERSAW(FREQ, 0, MCC(605), MCC(606), MCC(607), MCC(608))

```

2. The "Turrican" Resonator (Recursive CHOFLA)

Inspired by the gritty, metallic textures of legendary 8-bit sound chips (like the Commodore 64's SID). Here, we modulate the **delay time** with the signal itself.

- **Effect:** The delay (D_MOD) undergoes a jitter based on the output amplitude.
- **Result:** A metallic "mosquito buzz" that follows the signal's dynamics and disappears naturally when the notes stop.

```

INTERFACE 1200,100
VSTVARS 10
VSTVAR 0,0.5,"VOLUME","",0,1,1
VSTVAR 1,.1,"ATTACK","s",.001,10,0
VSTVAR 2,.1,"DECAY","s",.001,10,0
VSTVAR 3,1,"SUSTAIN","",0,1,1
VSTVAR 4,10,"RELEASE","s",.01,100,0
VSTVAR 5,0.001,"DETUNE","",.0001,.02,0
VSTVAR 6,0.1,"MIX","",.001,1,0
VSTVAR 7,3,"N","",0,15,2
VSTVAR 8,1,"FILTER?","",0,1,4,"No","Yes"
VSTVAR 9,.5,"Dry/Wet","",0,1,1

GAINENV &600,0,&601,0,&602,&603,&604

POST
MOD_SIGNAL = WIDE(PREV(O), -1)
// Audio-rate delay time modulation
O = CHOFLA(2, OUT, 0.002, MOD_SIGNAL * 0.001, 0, 0.9, VAR9, 0)
OUT = O

LAYER
OUT = GAIN * SUPERSAW(FREQ, 0, MCC(605), MCC(606), MCC(607), MCC(608))

```

3. The Recursive "Crunch" (Custom TANH Saturator)

An "intelligent" distortion where the input gain (DRIVE) is controlled by the output itself.

- **Effect:** The distortion coefficient increases as the signal gets stronger, creating internal "Ring Modulation."

- **Result:** An aggressive, "broken" analog gear saturation that remains musical thanks to the `TANH` mathematical function.

```

INTERFACE 1200,100
VSTVARS 10
VSTVAR 0,0.5,"VOLUME","",0,1,1
VSTVAR 1,.1,"ATTACK","s",.001,10,0
VSTVAR 2,.1,"DECAY","s",.001,10,0
VSTVAR 3,1,"SUSTAIN","",0,1,1
VSTVAR 4,10,"RELEASE","s",.01,100,0
VSTVAR 5,0.001,"DETUNE","",.0001,.02,0
VSTVAR 6,0.1,"MIX","",.001,1,0
VSTVAR 7,3,"N","",0,15,2
VSTVAR 8,1,"FILTER?","",0,1,4,"No","Yes"
VSTVAR 9,3,"MOD STRENGTH","",0,10,1

GAINENV &600,0,&601,0,&602,&603,&604

POST
// Drive coefficient varies based on the previous signal (VAR9 = Intensity)
D_MOD = 1 + (WIDE(PREV(O), -1) * VAR9)
O = TANH(OUT * D_MOD)
OUT = O

LAYER
OUT = GAIN * SUPERSAW(FREQ, 0, MCC(605), MCC(606), MCC(607), MCC(608))

```

4. The Harmonic "Disintegrator" (CHEBY)

The **Chebyshev (CHEBY)** operator is a surgical harmonic tool, but when used recursively, it can quickly turn into a weapon of sonic destruction. Unlike `TANH`, which rounds off the sound, `CHEBY` adds mathematically precise layers of harmonics that can "overcrowd" the frequency spectrum.

The "Legibility" Threshold

- **The "Sweet Spot" (VAR9 < 3):** At these levels, the recursion adds a rich, metallic sheen to the SuperSaw. You can still hear the notes, but they are wrapped in a high-voltage harmonic "aura."
- **The "Chaos Zone" (VAR9 > 10):** Here, the feedback loop pushes the signal so hard that the polynomial folds the waveform hundreds of times per second. The result is "illegible"—a wall of complex, shifting digital noise that barely resembles the original source.

The Final Template

We use a **Safety Valve** to prevent the signal from diverging, even when we push the `VAR9` slider to the "unreadable" zone.

```

INTERFACE 1200,100
VSTVARS 10
VSTVAR 0,0.5,"VOLUME","",0,1,1
VSTVAR 1,.1,"ATTACK","s",.001,10,0
VSTVAR 2,.1,"DECAY","s",.001,10,0
VSTVAR 3,1,"SUSTAIN","",0,1,1
VSTVAR 4,10,"RELEASE","s",.01,100,0
VSTVAR 5,0.001,"DETUNE","",.0001,.02,0

```

```

VSTVAR 6,0.1,"MIX","",.001,1,0
VSTVAR 7,3,"N","",0,15,2
VSTVAR 8,1,"FILTER?","",0,1,4,"No","Yes"
VSTVAR 9,3,"MOD STRENGTH","",0,10,1

GAINENV &600,0,&601,0,&602,&603,&604

POST
// 1. Recursive Safety Valve
// Clamping the feedback ensures the engine stays stable
// even when the harmonics become extreme.
SAFE_MOD = TANH(WIDE(PREV(O), -1))

// 2. Harmonic Drive
// VAR9 acts as the "Disintegration" control.
// Values 0-3: Musical/Metallic.
// Values 3-10: Industrial/Noise.
DRIVE = 1 + (SAFE_MOD * VAR9)

// 3. The Cheby Operator
// We use Order 3 for the 50% Mix Phase-Null effect.
O = CHEBY(OUT * DRIVE, 3, 0.5)

// 4. Output Assignment
OUT = O

LAYER
OUT = GAIN * SUPERSAW(FREQ, 0, MCC(605), MCC(606), MCC(607), MCC(608))

```

User Guide: Managing the Chaos

- **Finding the Fundamental:** If the sound becomes too "noisy" and you lose the pitch of your melody, lower VAR9. The Chebyshev polynomial is extremely sensitive; sometimes a value of 0.5 is all you need for a massive change.
- **The "Hollow" Trick:** At MIX 0.5, notice how the center of the sound seems to "scoop out." This is the Phase Null at work. It turns your SuperSaw into a "Ghost Saw."
- **Stability Note:** If the signal hits the **10^9 protection**, it means the DRIVE is too high for the current polynomial order. Simply lower the multiplier or use the TANH safety valve shown above.

Pro Tip: Hybrid External Processing

Crescendo's architecture is "open" by design. In the POST block, the OUT variable automatically contains the **sum of all LAYERS and INPUT 1** (external audio from your DAW). This means these recursive scripts work instantly as external effects.

How to Use It:

- **As a Pure Effect:** Do not play any MIDI notes. The external signal (e.g., a vocal or drum loop) will pass through the recursive loop, using its own waveform as the modulator.
- **As a Hybrid Processor:** Play chords with the SuperSaw while sending external audio into Input 1. The external signal (like a kick drum) will "punch" and modulate the SuperSaw, while the SuperSaw adds harmonic "weight" to the external audio.

Developer Note: We use the `O = ...` and `OUT = O` structure to prevent "Dead-Code Elimination." This ensures the compiler keeps the variable alive in memory, allowing `PREV(O)` to correctly access the data for the next sample.

Recursive Effects Cheat Sheet

| Effect Name | Modulated Parameter | Sonority | Best Used For... |
|---------------------------|---------------------|--------------------------------|---|
| Spectral Chewer | Phaser Frequency | Liquid, Vocal, "Masticating" | Evolving pads, sci-fi textures, and organic movement. |
| Turrican Resonator | Delay Time (Jitter) | Metallic, 8-bit Buzz, Gritty | Industrial leads, "broken" hardware sounds, and retro-gaming vibes. |
| Recursive Crunch | TANH Drive (Gain) | Saturated, Aggressive, "Fried" | Heavy techno leads, drum processing, and adding "heat" to external audio. |

Quick Integration Guide

To add these effects to any existing preset, simply copy the code into your POST block. Use the following logic to control the "Chaos Factor":

- **The WIDE(PREV(O), -1) Factor:** This is the heart of the engine. It swaps the stereo channels in the feedback loop. Without it, the effect is "dual-mono"; with it, the effect becomes a holographic stereo experience.
- **Safety First:** Crescendo's internal **100k Safeguard** and the natural limiting of TANH and PHASER ensure that even if you push the modulation to the max, your speakers (and ears) are safe. The signal will stay consistently around **-0.9 dB**.
- **External Routing:** Remember, if you route a microphone or a drum loop into **Input 1**, it is automatically summed into OUT. This means your external audio becomes the "fuel" for the recursion.

Which one should you choose?

1. **Want it to sound like a voice or a moving filter?** Go for the **Phaser**. It's the smoothest of the three.
2. **Want that classic "Mosquito" 8-bit sting?** Use the **CHOFLA**. It creates the most distinct harmonic "artifacts."
3. **Want to destroy the sound in a musical way?** Use the **TANH** or **CHEBY**. It's the ultimate tool for industrial "grit."

The "Recursive Chaos" Master Template

Copy and paste this code into your **POST** section. By default, it uses the **Recursive Phaser**. To switch effects, simply comment out the active one and uncomment your choice.

```
INTERFACE 1200,100
VSTVARS 10
VSTVAR 0,0.5,"VOLUME","",0,1,1
VSTVAR 1,.1,"ATTACK","s",.001,10,0
```



```

VSTVAR 2,.1,"DECAY","s",.001,10,0
VSTVAR 3,1,"SUSTAIN","",0,1,1
VSTVAR 4,10,"RELEASE","s",.01,100,0
VSTVAR 5,0.001,"DETUNE","",.0001,.02,0
VSTVAR 6,0.1,"MIX","",.001,1,0
VSTVAR 7,3,"N","",0,15,2
VSTVAR 8,1,"FILTER?","",0,1,4,"No","Yes"
VSTVAR 9,.5,"MOD","",0,1,1

GAINENV &600,0,&601,0,&602,&603,&604

POST
// --- RECURSIVE MODULATION SOURCE ---
// We swap L/R channels of the previous sample (O)
// to drive the effects with a stereo-interlocked signal.
MOD_SIGNAL = WIDE(PREV(O), -1)
// --- CHOOSE YOUR BEAST (Uncomment only one) ---
// [1] THE SPECTRAL CHEWER (32-Stage Phaser)
// Best for: Liquid, vocal, and evolving textures.
F_MOD = 440 + (MOD_SIGNAL * 100)
O = PHASER(32, OUT, 8, 0, F_MOD, 1, 0.7, VAR9, 0)
// [2] THE TURRICAN RESONATOR (Recursive CHOFLA)
// Best for: Metallic 8-bit buzzing and industrial grit.
// O = CHOFLA(2, OUT, 0.002, MOD_SIGNAL * 0.001, 0, 0.9, VAR9, 0)
// [3] THE RECURSIVE CRUNCH (Custom TANH Saturator)
// Best for: Aggressive "fried" analog distortion.
// D_MOD = 1 + (MOD_SIGNAL * VAR9 * 10)
// O = TANH(OUT * D_MOD)
// [4] The Harmonic "Disintegrator" (CHEBY)
// Clamping the feedback ensures the engine stays stable
// even when the harmonics become extreme.
// SAFE_MOD = TANH(WIDE(PREV(O), -1))
// VAR9 acts as the "Disintegration" control.
// Values 0-.3: Musical/Metallic.
// Values .3-1: Industrial/Noise.
// DRIVE = 1 + (SAFE_MOD * VAR9 * 10)
// We use Order 3 for the 50% Mix Phase-Null effect.
// O = CHEBY(OUT * DRIVE, 3, 0.5)
// --- FINAL OUTPUT ---
OUT = O

```

How to Use This Template

1. **The VAR9 Slider:** In all three examples, VAR9 acts as the "Chaos Intensity."
 - In the **Phaser**, it controls the Dry/Wet mix.
 - In the **CHOFLA**, it controls the Dry/Wet mix (increasing the "buzz").
 - In the **TANH and CHEBY**, it controls the "Mod Strength" (the amount of recursive distortion).
2. **The Hybrid Input:** Remember that OUT already contains any external audio from **Input 1**. If you route a drum loop into Crescendo, it will automatically be processed alongside the internal SuperSaw.
3. **The "Z-1" Loop:** This template uses the PREV(O) function. This creates a single-sample delay loop (Z-1), which is the fastest possible feedback path in digital audio, providing that "analog-like" instantaneous response.

Developer Cheat Sheet: Troubleshooting

- **No Sound?** Make sure at least one effect line and the OUT = O line are active.

- **Too Much "Mosquito" Buzz?** In the **CHOFLA** section, try reducing the multiplier 0.001 to 0.0005.
- **Distortion Too Square?** In the **TANH** or **CHEBY** section, lower VAR9 or the base 1 in D_MOD.

Final Note: The Ethics of Noise

The techniques described in this section—**Recursive Self-Modulation**—are designed to push the boundaries of digital synthesis. By following these scripts, you are intentionally creating non-linear, chaotic systems that behave more like a "living" analog circuit than a predictable digital plugin.

What to Expect:

- **The "Mosquito":** At high settings, you will hear high-frequency aliasing and "buzzing." This is not a bug; it is the sound of the mathematical feedback loop reacting at the sample frequency. It is a signature sound of the 8-bit era.
- **Dynamic Response:** These effects are highly sensitive to your input. A soft piano melody will sound completely different through these loops than a screaming SuperSaw.
- **Safety Guaranteed:** Despite the aggressive textures and "fried" harmonics, your signal is protected. The internal architecture of Crescendo, combined with the TANH and PHASER saturators, ensures that the output remains within a safe professional range (approx. **-0.9 dB**).

Sound Designer's Responsibility:

Use the **MOD STRENGTH (VAR9)** slider wisely.

- **Lower values** add "analog soul" and subtle stereo widening.
- **Higher values** invite the "Ghost in the Machine."

If you find yourself in a territory that is too chaotic, simply pull back VAR9 or increase the DMIN / F_MOD base values to "stabilize" the beast.

"In the world of recursion, there are no mistakes—only new harmonic discoveries."

Appendix A: 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.

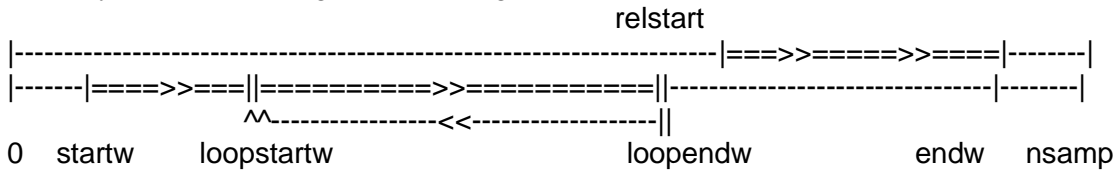
|-----|-----||====>=====<=====||=====<=====|-----|
0 endw loopendw loopstartw startw nsamp

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.



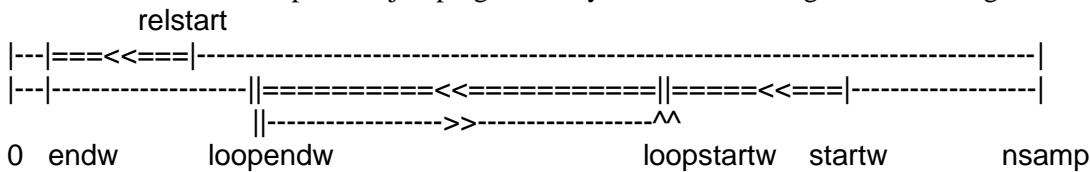
Loop with separate release, backward

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.



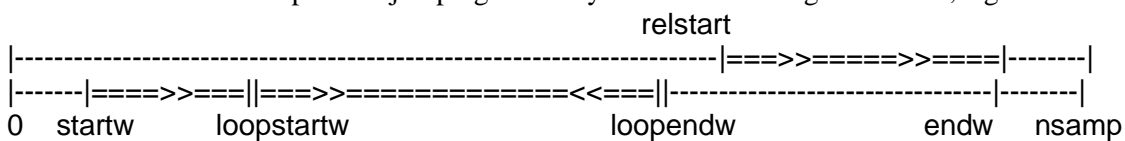
Loop with separate release, forward+backward

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.



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.

