ChatGPT

Generating Logic Pro `.pst` and `.cst` Preset Files from Scratch

Overview of Logic Pro Preset Formats

Logic Pro uses two main preset file types for saving settings: **Plug-In Settings** (`.pst` files) for individual plug-in presets, and **Channel Strip Settings** (`.cst` files) for whole channel strip configurations. These files are **not plain text** – for example, opening a `.pst` in a text editor yields gibberish [1]. Instead, they are structured binary or plist files that Logic reads internally. To generate these from scratch, one must mimic their exact format and content:

- `.pst` **(Plug-In Setting)** – A proprietary binary format containing the settings for one plug-in (e.g. an EQ or compressor preset). Each `.pst` is essentially a custom binary blob specific to the plug-in, with a fixed header and binary-encoded parameter values [2]. It is **not** an XML or JSON plist, but a binary format likely defined by the plug-in or Logic's audio engine.
- `.cst` **(Channel Strip Setting)** – In modern Logic (10.x and 11.x), a channel strip preset is saved as a Property List (plist) structure (often XML) that references one or more plug-in presets [3]. The `.cst` acts as a container listing each insert slot's plug-in and an associated preset file name. The actual plug-in settings may reside in separate preset files (often `.aupreset` files for Audio Unit plug-ins). In older Logic versions, `.cst` files were a single binary archive embedding all plug-in data, but in Logic Pro 11.0.0 the format is plist-based for stock use.

`.pst` File Format (Logic Pro Plug-In Setting)

Each `.pst` file has a specific binary structure consisting of a header followed by the encoded parameter values. **The header** is 28 bytes long in observed cases [2] and includes identifying information for the preset and plug-in. Following the header, the plug-in's parameter values are stored in a fixed sequence of binary numbers (typically floats or similar). The general structure is:

- **File Header (28 bytes):** Contains multiple 32-bit little-endian integers and a magic ASCII signature. For example, a Logic `.pst` header might break down as: 4 bytes file size, 4 bytes format version (often `0x01`), 4 bytes parameter count, an 8-byte ASCII sequence `"GAMETSPP"` (constant magic) and a 4-byte plug-in identifier, plus 4 bytes reserved or zero [4]. In one reverse-engineered example, the header bytes for an ES M synth patch were:
  `58 00 00 00` (file length 0x58 bytes)
  `01 00 00 00` (version number 1)
  `10 00 00 00` (16 parameters expected)
  `47 41 4D 45 54 53 50 50` (`"GAMETSPP"` ASCII)
  `C9 00 00 00` (plug-in ID 0xC9 for ES M)
  `00 00 00 00` (reserved zeros) [4].
  The plug-in ID is what identifies which plug-in the preset is for – Logic will refuse to load a `.pst`

into the wrong plug-in. (For example, Channel EQ uses a different ID value than ES M.) These IDs aren't documented publicly; you typically obtain them by inspecting known presets for that plug-in.

- **Parameter Data:** After the header, the file encodes each parameter's value in order. In known Logic presets, parameters are stored as binary floating-point numbers in **little-endian** format [5]. Each parameter value is usually a 32-bit float (4 bytes), though the file may align or pad data to 8-byte groups in some cases [2]. The parameters appear **in a fixed sequence** – specifically, in the same order the plug-in's UI or API defines them [6]. There are no human-readable keys or names in the `.pst`; the meaning of each value is implied by its position. For example, if the first parameter is "Gain" and the second is "Frequency", the first 4-byte float corresponds to gain, the second to frequency, etc.

Notably, the values are stored in **actual units or internal units** that the plug-in uses, not a generic 0–1 range in many cases. For instance, a frequency parameter might be stored as `250.0` (float) for 250 Hz, and an EQ gain might be stored as `6.0` for +6 dB. In a sample Channel EQ `.pst` we analyzed, we observed float values like `250.0`, `6.0`, `0.707` etc., corresponding to meaningful settings (frequency, gain, Q, etc.) in the EQ preset. This means the AI agent must supply correctly scaled values exactly as the plug-in expects them. **Enumerated parameters** (like a filter type or reverb algorithm) are typically stored as numeric codes as well – e.g. an index 0,1,2… corresponding to the selected option (Logic's `.aupreset` uses human-readable names for these, but the `.pst` will use numbers).

- **No high-level structure:** `.pst` files do not contain XML tags or key–value dictionaries. They are essentially a binary dump of the plug-in's state. As one reverse-engineering study concluded, "a 28 byte-long header… [and then] each plugin setting stored as [a little-endian float]" [2]. There is no checksum or encryption; if the bytes are in the correct format, Logic will load it. Conversely, any deviation from the expected format (wrong length, wrong ID, too few/many bytes) will cause Logic to **silently fail** to load the preset. There is minimal error feedback – an invalid `.pst` will just not change the plug-in settings.

Example `.pst` Structure (Channel EQ)

For illustration, here is a breakdown of an example Channel EQ `.pst` (hypothetical structure based on analysis):

- **Header:** 28 bytes total:
- Bytes 0–3: `0xEC 00 00 00` (236 in little-endian, meaning the file is 236 bytes long)
- Bytes 4–7: `0x01 00 00 00` (format version = 1)
- Bytes 8–11: `0x33 00 00 00` (51 parameters in this preset)
- Bytes 12–19: ASCII `"GAMETSPP"` (magic constant)
- Bytes 20–23: `0xEC 00 00 00` (plug-in ID = 236 for Channel EQ)
- Bytes 24–27: `0x00 00 00 00` (reserved padding)
- **Parameter data:** Starting at byte 28, a sequence of 51 little-endian 32-bit floats. For example, the first few parameters might be: `1.0` (`0x3F800000`), `36.4` (`0x4211999A`), `4.0` (`0x40800000`), `0.70999998` (`0x3F35C28F`), `1.0` (`0x3F800000`), `250.0` (`0x437A0000`), `6.0` (`0x40C00000`), etc. Each corresponds in order to a specific knob/setting on the Channel EQ (perhaps band enables, frequencies, gains, Q values, etc. in sequence).

To successfully **create a `.pst` file**, an AI coding agent must output bytes in this exact format. In practice, this means coding the binary write of the header (with the correct length, count, plugin ID, etc.), then converting each parameter value from the desired setting into the appropriate binary float bytes in little-endian order. Tools like struct packing (in Python or C) or manual byte assembly are used for this. Robert Heaton's reverse-engineering example shows building a custom patch by writing a header and mapping settings to bytes [7] [8].

`.cst` File Format (Logic Pro Channel Strip Setting)

A Logic channel strip setting (`.cst`) encapsulates an entire channel strip configuration – which plug-ins are loaded on the channel (instrument and effects), and each plug-in's preset. In Logic Pro 11.0.0, `.cst` files are implemented as property lists that reference the individual plug-in presets. In other words, the `.cst` itself is an **XML or binary plist** (Apple Property List) describing the setup, and each plug-in's settings may be in separate files (often `.pst` or `.aupreset` files).

Structure: A `.cst` plist typically contains a top-level dictionary with keys like `name` (the name of the channel strip preset) and `inserts` (an array of insert slot descriptions). Each entry in the `inserts` array is a dictionary with at least: - `name` – the plug-in name (as recognized by Logic's plugin list, e.g. "Channel EQ", "Compressor", "ChromaVerb"). - `preset` – the file name of the preset to load into that plug-in. This is usually the base name of a `.pst` or `.aupreset` file (without extension) that Logic will look up.

For example, part of a `.cst` file in XML format might look like this:

```
<dict>
    <key>name</key>
    <string>My Vocal Chain</string>
    <key>inserts</key>
    <array>
        <dict>
            <key>name</key>
            <string>Channel EQ</string>
            <key>preset</key>
            <string>My Vocal Chain_Channel EQ_Pre</string>
        </dict>
        <dict>
            <key>name</key>
            <string>Compressor</string>
            <key>preset</key>
            <string>My Vocal Chain_Compressor_</string>
        </dict>
        <dict>
            <key>name</key>
            <string>ChromaVerb</string>
            <key>preset</key>
            <string>My Vocal Chain_ChromaVerb_</string>
        </dict>
```

```
    </array>
</dict>
```

In the above, three inserts are defined: Channel EQ, Compressor, and ChromaVerb, each with an associated preset name. This matches the pattern seen in the provided example `.cst` [3] [9]. The actual preset files would be named `My Vocal Chain_Channel EQ_Pre.pst` (or `.aupreset`), `My Vocal Chain_Compressor_.pst`, etc., and stored in the appropriate folder where Logic expects user presets.

Preset File References: Logic resolves the `preset` file names to actual files on disk. For stock (Apple) plug-ins and Audio Units, Logic typically uses the standard AudioUnit preset directories: e.g. `~/Library/Audio/Presets/<Manufacturer>/<PluginName>/`. The preset file could be a `.pst` or an `.aupreset` (Logic's new Audio Unit preset format, which is an XML plist of parameters). In many cases Logic's own effects now save presets as `.aupreset` (XML) files. In the example above, the Channel EQ preset might actually reside as an `.aupreset` file (since Apple's Channel EQ supports the AU preset format), whereas older Logic plug-ins or certain cases might use `.pst`. The `.cst` doesn't explicitly state the extension – Logic will try known extensions in the expected location for that plug-in.

Older vs New Format: In older versions of Logic, a `.cst` was a monolithic binary file (likely an NSKeyedArchiver or similar) embedding all plug-in data. In those files you would find the plug-in names and their `.pst` data back-to-back in one file. For instance, a legacy `.cst` might contain the string "Channel EQ", followed by that plug-in's binary preset blob, then "Compressor", followed by its blob, etc., in a proprietary sequence. The provided `Seed.cst` appears to be such a binary archive – it contains ASCII names ("Channel EQ", "Compressor", "ChromaVerb") and internal `GAME...TSPP` sequences, suggesting embedded `.pst` sections for each plug-in. However, **Logic Pro 11** uses the plist approach for user-saved channel strips, which is easier to generate and parse (and aligns with the Logic **Patch** format introduced in Logic Pro X's library). We will focus on the plist-style `.cst` since that is the current format.

Generating a `.cst` (plist style): An AI agent can create a valid `.cst` by constructing a proper plist dictionary structure. In plain text (XML), it should match Apple's plist DTD. The keys and formatting must be exact. For example, the `<key>name</key>` should be the channel strip name (this can be anything the user wants). The `inserts` array must list each insert in order (slot 1 is the first dict, slot 2 the second, etc.). **The `name` of each insert must exactly match the plug-in's name as known to Logic**, otherwise Logic won't know which plug-in to load. (For stock plugins, this is the name that appears in the plug-in selector; e.g. `"ChromaVerb"` or `"DeEsser 2"` must match exactly [10].) The `preset` string should match the preset file's base name. If the preset file is not found or is malformed, Logic will silently skip loading that insert's settings – it might instantiate the plug-in with default settings or leave the slot empty. So the agent must ensure that any preset files referenced in the `.cst` are also generated and placed in the correct location.

Example: In the provided `BeatAware Vocal.cst`, the inserts array includes entries like:

- Name = "Multipressor", Preset = "BeatAware Vocal_Multipressor_" [11]
- Name = "ChromaVerb", Preset = "BeatAware Vocal_ChromaVerb_" [12]

Corresponding files `BeatAware Vocal_Multipressor_.aupreset` and `BeatAware Vocal_ChromaVerb_.aupreset` were provided, containing the actual parameter values for those plug-ins

in XML form. When Logic loads the channel strip, it reads each insert, loads the appropriate plug-in by name, then loads the `.aupreset` into that plug-in to apply all the parameter values.

To summarize, **constructing a Logic 11** `.cst` involves creating a plist (XML or binary) with the above keys/structure. This can be done by using Python's `plistlib` or simply formatting XML strings. You must then save it with the `.cst` extension. If generating in XML form, ensure the XML header and doctype are present as in the example [13]. (Alternatively, you can generate a binary plist using tools or libraries, which Logic will also accept – just be careful to get the format exactly right.)

Parameter Encoding and Considerations

Parameter identification: In `.pst` files, parameters are identified by position, not by name. The agent generating the file needs prior knowledge of the plug-in's parameter order and valid ranges. This usually comes from documentation or reverse-engineering. For stock Logic plug-ins, one approach is to save a preset manually and inspect it (via a hex editor or by converting to `.aupreset` if possible) to map out which bytes correspond to which setting. For example, by setting all knobs to known values and seeing the pattern in the `.pst` file, one can deduce the ordering [14] [15]. The Robert Heaton blog demonstrates this "differencing" technique – toggling one parameter at a time and observing which bytes change – to map bytes to parameters.

Data types: Nearly all continuous parameters (volumes, frequencies, times, etc.) are stored as 32-bit floats in little-endian. The reverse-engineered ES M synth used floats for everything (even values like Octave which were integer in concept) [5]. We can assume the same for most Logic plug-ins: even if a parameter is an integer or boolean, it may be stored as a float (e.g. 1.0 for true, 0.0 for false, or discrete integer values represented in a float). One should take care to convert any integer value to float bytes if the format expects it. (The exception might be very new plugins that use integer types, but in presets these typically still appear as floats or as part of a structure. When in doubt, use float.)

Enumerations: For parameters with a fixed set of options (e.g. compressor circuit type, reverb algorithm), the `.pst` will likely store an integer code. In `.aupreset` files, Apple uses the option's name (e.g. `"Plate"`), but internally the plug-in will map that to an index. The AI agent must use the correct index that the plug-in expects. These are usually zero-based. For example, if "Plate" is the third algorithm in ChromaVerb, its stored value might be `2.0` (assuming 0 = first, 1 = second, 2 = third) in the `.pst` float data. Without official docs, the safe method is to create example presets for each option and inspect the bytes to find the mapping.

Scaling: Use real-world units as the plug-in API uses. As noted, many Logic plug-in presets store real values (not normalized 0–1). If a frequency knob goes 20 Hz to 20kHz, and you set it to 5000 Hz, the preset likely contains `5000.0` (or a related encoded float if nonlinear). There are exceptions – sometimes a parameter might be internally scaled (for instance, dB could be stored as linear gain factor, though in Apple's own plug-ins they tend to store the displayed value). The best practice is to confirm by comparing `.aupreset` values (which are often the human-readable values) to the resulting `.pst`. In our inspection, the values in `.aupreset` (inside `<real>` or `<string>` tags) matched the floats in `.pst` in magnitude for the stock plugins.

Plugin identifiers: Ensure the correct plug-in ID in the `.pst` header. This is not a human-readable code – it's an internal constant. For Apple's own plugins, these IDs are consistent across systems (e.g., Channel EQ = 0xEC as seen above, Compressor might be another number, etc.). If this ID is wrong, Logic will treat the file as either corrupt or meant for a different plug-in. There is no public list of these IDs, so you obtain them by examining an existing preset file for that plug-in. (One trick: save a preset named "test.pst" from the plug-in in Logic, then open it in a hex viewer to find the ID in bytes 20–23, just after the `"TSPP"` magic.) Use that exact 4-byte value in any generated presets for that plug-in.

File paths and naming: When deploying the generated files, they must be placed where Logic can find them. For `.pst` (individual plugin presets), the usual location is `~/Music/Audio Music Apps/Plug-In Settings/<PluginName>/` for Logic's own plugin preset menu. For `.cst`, the user settings go in `~/Music/Audio Music Apps/Channel Strip Settings/<Type>/` (with `<Type>` being something like Instrument, Audio, Bus, etc., depending on the channel strip type). If using `.aupreset` files, those belong in `~/Library/Audio/Presets/<Manufacturer>/<PluginName>/`. However, if you keep all related files together (the `.cst` and its referenced presets) and use Logic's **Import** function, you can manually load a `.cst` from a folder and it will pull the presets from the same folder. Just note that the `.cst` only stores the preset file names, not full paths – Logic searches its standard preset directories or the current directory of the `.cst` for those names.

Tools and Techniques for Programmatic Generation

Because these formats are proprietary, **there is no official SDK or command-line tool from Apple to create `.pst` or `.cst` files**. We have to rely on manual or reverse-engineering techniques:

- **Hex editors and Diffing:** As mentioned, to understand a plug-in's `.pst` format, one can create multiple presets in Logic varying one parameter at a time, then compare the binary differences [14] [15]. This reveals how values are encoded. This is labor-intensive but effective. Once the format is known, you can hardcode the structure in a script. For example, Heaton's blog demonstrates writing a small Ruby class to output a valid ES M synth `.pst` after figuring out the structure [7] [8].

- `plutil` **and plist libraries:** For `.cst` files (and `.aupreset` files), you can use Apple's `plutil` tool or any plist-handling library (Python's `plistlib`, CFXMLElement in Swift, etc.) to read or write them. If Logic saves a channel strip and the `.cst` is in binary plist form, you can convert it to XML with `plutil -convert xml1 file.cst`. Similarly, you can generate an XML plist and convert it to binary plist for a smaller file (with `-convert binary1`). The key is to maintain the exact keys and structure that Logic expects (case-sensitive).

- **AppleScript or Automation:** If coding an AI agent on a Mac, one could automate Logic Pro to assist conversion – for instance, generate an `.aupreset` (which is easier, as it's just XML), then use AppleScript or the Logic API to load it into the plug-in and use the "Save Setting" command to produce a `.pst`. However, this is an indirect approach and requires Logic running. There is no simple command-line conversion (the LogicProHelp forum confirms that batch converting .aupreset ↔ .pst requires manually loading each preset in Logic or the plug-in's UI [16]).

- **Manual template approach:** Another strategy is to use an existing `.pst` file as a template. For example, take a known good preset and only modify the bytes that correspond to parameter values

(leaving header and structure intact). This requires knowing the byte offsets for each parameter, but can be faster once mapped. **Warning:** This can be risky if the target preset differs in plug-in version or parameter count. Always adjust the file length field in the header if you add or remove bytes.

- **Validation:** After generating a `.pst` or `.cst`, test it in Logic. Place the file in the appropriate folder and see if it appears in Logic's preset menu. If Logic ignores it, something is off (check the header values and byte counts carefully). Logic won't usually crash on a bad preset; it will just do nothing, so iteration is needed to troubleshoot.

`.pst` vs `.aupreset` and Conversion Considerations

Apple's `.aupreset` format (Audio Unit preset) is an XML plist that lists parameters by name with human-readable values. Many Audio Units (including Logic's built-in ones) can load either their native `.pst` or an `.aupreset`. However, **Logic does not automatically convert between the two formats**. If you have a JSON or Python representation of settings, it might be tempting to just produce an `.aupreset` (since it's easy to serialize a dict to plist) instead of the binary-heavy `.pst`. This is a valid approach if your goal is just to load presets into Logic somehow: you could drop the `.aupreset` in the proper Presets folder and load it from Logic's Library browser. In fact, for stock Logic effects, using `.aupreset` is often sufficient – Logic's Library will show them under the User Presets for that plug-in [17].

However, if the goal is specifically to **create** `.pst` **files** (perhaps for compatibility or because an AI agent is instructed to output `.pst`), you will likely need to do the full binary construction. There is no known tool that "converts" an `.aupreset` file to a `.pst` file automatically outside of Logic. The recommended workflow if full automation is too hard: generate the `.aupreset` (which is straightforward from JSON), then **manually load and save it as** `.pst` **within Logic**. This could even be semi-automated with GUI scripting or a small Logic script, but it's not a one-step conversion. A LogicProHelp forum answer confirms that the only direct way is to load the preset in the plug-in and then use the plug-in's own Save command for each preset [16].

Workaround suggestion: If full generation of `.pst` is impractical for many presets, consider using the `.cst` approach as a container. You could create a channel strip with one plug-in and have the `.cst` reference an `.aupreset`. For example, make a channel strip with just "Compressor" on it, and set the `.cst`'s inserts[0] name = "Compressor", preset = "MyPreset_Compressor_". Then generate `MyPreset_Compressor_.aupreset` via code (XML). This way, you skip making a binary `.pst` altogether – Logic will load the compressor, then apply the `.aupreset`. You can then even re-save that from Logic's plugin menu as a `.pst` if needed. This method leverages the fact that Logic's channel strip can load AU presets on startup.

Guidelines for an AI Coding Agent

To summarize actionable steps for an AI coding agent (e.g. Codex) to create these files:

1. **Choose Format Path:** Decide if you will output a `.pst` directly or a combination of `.cst` + `.aupreset`. For a single plug-in, generating a `.pst` is possible but involves careful binary handling. Generating an `.aupreset` (XML) is easier and may be sufficient if Logic can consume it.

For multiple plug-ins, generating a `.cst` (XML plist) that references multiple `.aupreset` files is a clear approach.

2. **Gather Required Data:** Ensure you have the plug-in's **parameter schema**:

3. Number of parameters and their order.
4. Expected unit scaling or encoding for each parameter.
5. The plug-in's **ID** for the `.pst` header (if going the .pst route).

6. The exact **plug-in name** string and manufacturer (for .cst/.aupreset placement). You might hardcode this info for Logic's stock plug-ins (based on prior reverse engineering). For example, Channel EQ has 51 parameters, ID 0xEC; Compressor has X parameters, ID maybe Y, etc. (If unknown, the agent cannot guess – this info must come from documentation or by analyzing example files.)

7. **Constructing a `.pst`:**

8. Calculate the total file size and parameter count, and format them as 32-bit little-endian ints for the header.
9. Use the known constant header template `"GAMETSPP"` and insert the correct plug-in ID (little-endian) in the header.
10. Convert each parameter value to a 4-byte IEEE-754 float (little-endian). Be mindful of byte ordering.
11. Concatenate header + all parameter bytes. (Make sure the file size in the header matches the actual byte length.)

12. Save with `.pst` extension.

13. **Constructing a `.cst` + presets:**

14. Build a Python dict or XML string for the channel strip plist. Include the name and an inserts list of dicts.
15. For each insert, set the `name` exactly to the plug-in name. Set `preset` to a unique identifier string (often you can use a pattern like `"PresetName_PluginName_"`).
16. Separately, for each plug-in, create either a `.pst` or an `.aupreset` file containing that plug-in's settings. If using `.aupreset` (recommended for ease), create an XML plist with a structure: `<dict><key>name</key><string>...preset name...</string><key>parameters</key><dict>... each param ...</dict></dict>`. The `parameters` dict should use the plug-in's parameter names as keys (this you can get by inspecting an existing .aupreset or using the Audio Unit API) and the desired values as `<real>` (for numeric) or `<string>` (for enumerated types like note values or option names). Ensure the preset file name matches what you put in the `.cst` and use the correct file extension.

17. Save the `.cst` (either as XML with the proper plist DOCTYPE or convert to binary plist) and save all the referenced preset files in the appropriate folders or together.

18. **Validation & Loading:** After generation, the agent (or a user) should place the files in the correct location and attempt to load in Logic. If the plug-in does not reflect the settings:

19. Check the `.cst` in a text editor or via `plutil` to ensure the structure is valid XML/plist.
20. Check that preset filenames in the `.cst` exactly match the actual files (excluding extension).
21. If a `.pst` doesn't load, compare its hex against a known-good `.pst` for the same plugin to spot discrepancies (especially in header fields).
22. Use Logic's **Recall from Library** or **Setting menu** to load the preset. If it doesn't appear in the Library, it might be in the wrong directory or the plist is malformed.

Warning: Logic will not explicitly warn about a bad preset file. It will just ignore it. For instance, if your `.cst` references "SuperDuperReverb" but you don't have such a plugin, Logic will simply leave that slot empty with no message. Similarly, if the preset file exists but is not correctly formatted, Logic may load the plug-in but with default settings. Therefore, meticulous attention to format is required.

In conclusion, it is technically possible to generate Logic's `.pst` and `.cst` files entirely from scratch by replicating their binary/plist format [18]. Successful examples (like the custom ES M patch generator) show that once the format is understood, writing out bytes to form a valid preset is straightforward [19] [20]. The key challenges are obtaining the necessary internal details (parameter order, IDs) and correctly encoding all values. If the direct route proves too difficult, leveraging `.aupreset` (which is much easier to formulate from a JSON-like structure) as an intermediary is a practical workaround – you can generate those in code and then use Logic to convert or load them. By following the structures and examples above, an AI coding agent can produce presets that Logic will accept and apply to its stock plug-ins.

Sources: Reverse-engineering details of Logic preset format [2] [5]; example Logic Pro channel strip preset structure [3] [9]; and user/community insights on preset conversions [16].

---

[1] [2] [4] [5] [6] [7] [8] [14] [15] [18] [19] [20] Reverse engineering LogicPro synth files | Robert Heaton
https://robertheaton.com/2017/07/17/reverse-engineering-logic-pro-synth-files/

[3] [9] [10] [11] [12] [13] BeatAware Vocal.cst
file://file-9h4Z6Qs2aWEv3sZASxUaeX

[16] Is there a fast/direct way to convert .au presets to the Instrument .patch format? - Logic Pro - Logic Pro Help
https://www.logicprohelp.com/forums/topic/144603-is-there-a-fastdirect-way-to-convert-au-presets-to-the-instrument-patch-format/

[17] How to access .aupreset files in Logic (NI Massive)?
https://discussions.apple.com/thread/6790454