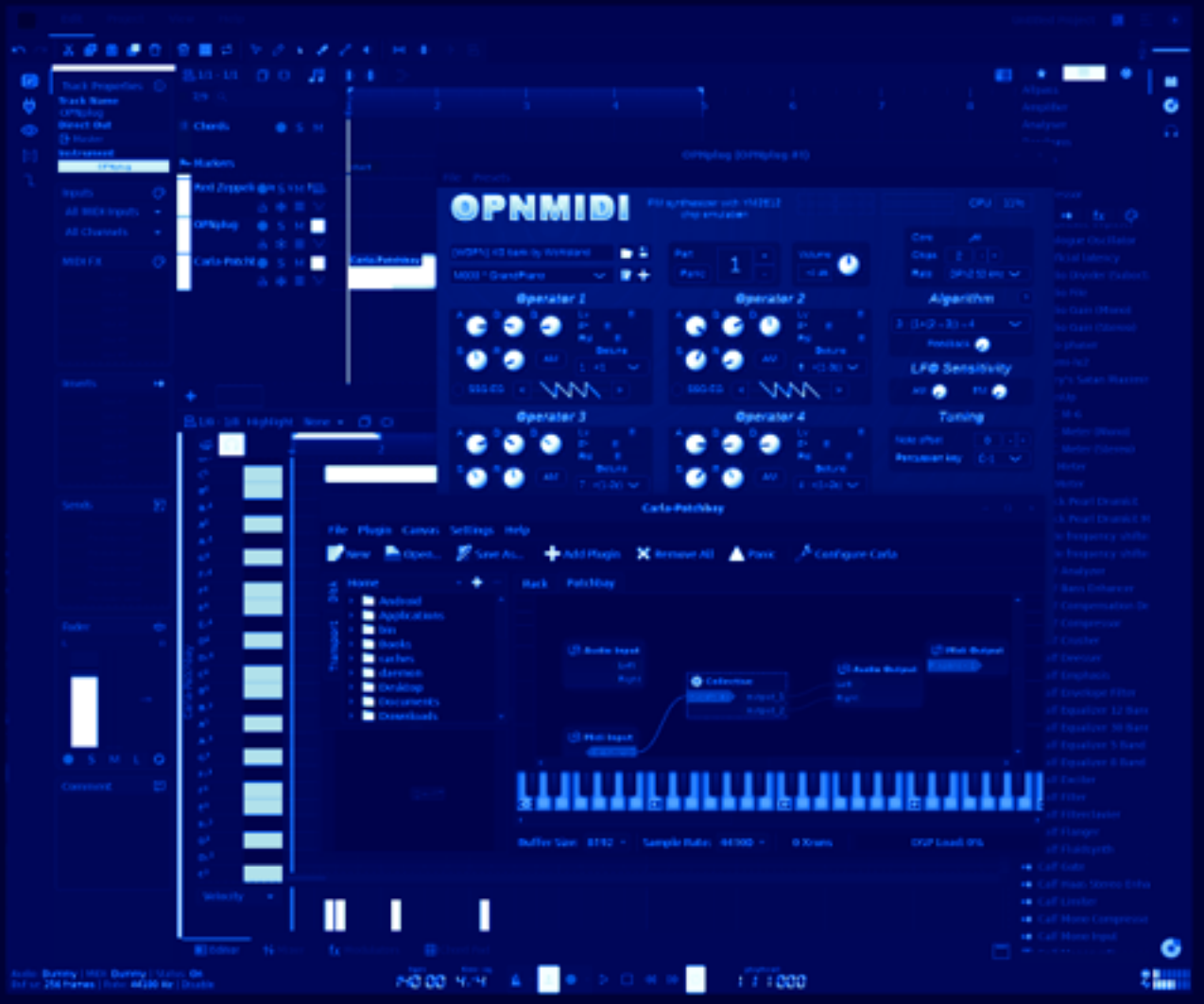
CLAP

Audio Plugin Developers Guide



CLAP オーディオプラグイン開発者ガイド

atsushieno　著

2022-10-30 版　　オーディオプラグイン研究所 発行

本書について

本書は CLAP オーディオプラグインの、特にプラグインとホストの開発に興味のあるプログラマーが、読んで参考になるような本を目指して書かれています。

プラグインや DAW の開発ができるようになるための全ての情報を本書で説明することはありません。特にどのプラグインフォーマットを採用する場合でも不可欠である DSP や GUI の作り方について、本書では全く言及しません。

CLAP プラグインを「使う」ための知識については、正直それほど必要ありません…というより、CLAP はまだ正式版（バージョン 1.0）が公開されたばかりなので、CLAP をサポートしている製品も多くはなく、またエンドユーザーにとってのオーディオプラグインの「使い方」は、DAW によって操作が複数のプラグインフォーマットの間で共通化されていることが多いためです。CLAP プラグインの「使い方」を学ぶ最短の方法は、VST3 プラグインや AU プラグインの使い方、ひいては各 DAW の使い方を学ぶということになるでしょう。

CLAP「ならでは」の「使い方」を学ぶのは、オーディオプラグインの使い方や一般的な機能を覚えてからで十分ですし、特にプラグインフォーマット固有の機能に依存する楽曲の打ち込みは、そのプラグインフォーマットを利用できない環境へのポータビリティを犠牲にすることとイコールであることを意識すべきです。もし CLAP を使用していて何らかの問題が発生した場合（DAW のバグなど）、一般的な機能のみを利用していたら同じプラグインの VST3 版や AU 版に移行して事なきを得るかもしれませんが、CLAP に依存した打ち込みになっていたら、そこでゲームオーバーです。同様に、プラグインフォーマットにおける特殊な機能は、複数のプラグインフォーマットをサポートする DAW ではサポートされないかもしれません。もし VST3 を開発している Steinberg の DAW である Cubase でしか利用できない VST3 の機能があったら、それをあえて使うことで Cubase にベンダーロックインされるべきことがあるでしょうか。同様に、Bitwig Studio でしか使えないような CLAP の機能をあえて使うようなことは、本書では推奨しません。

もちろんこれは「楽曲の打ち込み」という限定されたコンテキストでの警句であり、事後的な再現が不要なデモンストレーションやプラグイン開発においては別の視点が必要です。CLAP ならではの機能をサポートすることは全く悪いことではありません。どのようなプラグイン機構の新機能も、その設計が DAW とプラグインの双方で実現すべきものであれば、他の拡張可能なプラグインフォーマットでも追従して実装するかもしれません。誰かが先駆者として先行事例を作っておくことは DTM の世界にとって有用です。

関連書籍など

筆者はオーディオプラグインに関連する同人誌をいくつか発行しており、多分 CLAP の位置付け等を理解するうえでそれなりに参考になると思うので並べておきます。

「DAW・シーケンサーエンジンを支える技術(第2版)」では、DAW がどうやってプラグインを利用するのか、オーディオプラグインはざっくりどういう仕組みとして作られているのか、楽曲のシーケンスはどう作られて保存されているのか、といった話をふわっと書いています(「ふわっと」というのは主観的な表現ですが、コードが出てこない程度の抽象論に終始する内容です)。

「LV2 オーディオプラグイン開発者ガイド」は日本語でまとまった情報が多くない LV2 オーディオプラグインについて解説し、プラグインの開発方法を説明しています。(プラグイン開発に携わっていないと難しく、プラグイン開発をやっていると物足りないレベルかもしれません。)

CLAP については、2022 年 7 月 6 日にオーディオプラグイン規格勉強会を開催して、スライドをもとに参加者各位からさまざまな情報を共有いただき、本書の内容にも各所で反映されています。

https://speakerdeck.com/atsushieno/audio-plugin-format-study-meetup-2022-dot-7-6-jp

目次

第 1 章

CLAP について

1.1 CLAP とは

CLAP は 2022 年 6 月にバージョン 1.0 が公開されて幅広く認知されるようになった、新しいオーディオプラグイン機構（あるいはオーディオプラグインフォーマット）です。比較できる技術としては、Steinberg の VST (2, 3)、Apple の AudioUnit (v2, v3)、LV2 などが挙げられます。

1.2 CLAP の特徴

ここでは、CLAP プロジェクトが特徴としてアピールしているポイントをいくつか紹介します。本書で解説する要点とは特に関係がありません。本書では主に CLAP 開発のための API や勘所を解説します。

1.2.1 商用開発に向いているライセンス

CLAP は MIT ライセンスで利用可能です。VST3 の開発に必要な vst3_sdk は GPLv3 と Steinberg の商用ライセンスとのデュアルライセンスで公開されており、VST2 はオープンソースでは公開されませんでした。AudioUnit は Apple のプロプラエタリな SDK の一部であり、Apple プラットフォーム（macOS、iOS）でしか利用できません。

vst3_sdk を GPLv3 ライセンスに基づいて利用するということは、これを利用して開発した VST3 プラグインのソースコードは公開して誰でも利用できるようにしなければならないということであり、商用製品を販売してもそのソースコードを誰でもビルドでき、そのバイナリプログラムを頒布・販売できるということになります。これは商用プラグイン開発者にとっては実質的に商用ライセンスで公開されているのと同様です。CLAP のリベラルなライセンスは、実質的には Steinberg の商用ライセンスからの解放であると評価できます。

LV2 は CLAP と同様、商用利用にやさしい ISC ライセンスで提供されています。GNU フリーソフトウェアの世界では、VST3 も LV2 も CLAP も特に問題なく受け入れられています。

1.2.2 モダンな機能のサポート

オーディオプラグインに求められる機能は多様化しています。オーディオプラグインはインストゥルメントあるいはエフェクトとして機能するものであり、オーディオプラグイン以前は MIDI 1.0 がそれらを部分的に実現していたものですが、2022 年のオーディオプラグインシステムには、MIDI 1.0 が実現していた機能に加えて、32 ビット float（ときに 64 ビット float）でのパラメーター、ノート別パラメーター（ MIDI であれば MPE など）、ノートのアーティキュレーション（ ノート別の特殊表現）、アンビソニックや 3D のような多チャンネルオーディオなど、さまざまな追加機能が求められるようになっています。CLAP はこれらの表現を可能にしていますし、また新機能を追加しやすいような拡張機能の仕組みが整備されています（ この点は VST3 や LV2 と同様です）。

2020 年には MIDI 2.0 の正式版が公開され、MIDI 1.0 および 2.0 との適切な相互運用性も求められています。特に MIDI 入力メッセージを自分のイベント機構のメッセージに変換する VST3 の仕組みが、この点でうまく機能していない側面があり、CLAP は問題が生じないように設計されています。

1.2.3 マルチコア CPU の効率的なサポート

CLAP は CPU のマルチコアの効率的な利用をアプリケーションのコーディングで可能にできる 2020 年代にふさわしい設計になっています。詳しくは thread-pool 拡張機能の節で言及します。

1.2.4 高速なプラグインのスキャン

CLAP 開発チームは、CLAP のプラグインスキャンではプラグインのインスタンスを生成する必要がないため、プラグインのスキャンは高速に完了すると主張しています。これについては CLAP プラグイン開発の章で改めて解説・評価します。

1.2.5 プラグインの状態を破壊的に変更しないオートメーションのサポート

DAW でプラグインパラメーターをリニアに変更する、いわゆるオートメーションの仕組みを利用すると、一般的にはオートメーションの完了後にパラメーターの状態は変更されたままの「 破壊された」状態になります。CLAP ではパラメーターの変更の由来がユーザーの操作によるものかオートメーションによるものかといった情報をホストとプラグインの間で伝達でき、オートメーション完了後にパラメーターを「 元の値」に戻せるようになります。

これは DAW の実装の仕方しだいではありますが、本書の序文でも書いたとおり、筆者はこのような（ 現時点で）CLAP にしか存在しない機能に依存した楽曲の打ち込みは推奨しません。DAW の CLAP サポートなどに問題があった場合、逃げ場がなくなってしまうからです。

1.2.6 オープンなコミュニティ

CLAP は Bitwig Studio DAW を発売しているドイツの Bitwig 社とドイツの u-he 社（ Heckmann Audio GmbH）が中心になって設計・開発しています。CLAP はコミュニティ主導の規格である、と

CLAP 開発チームは主張しています。CLAP には一般開発者向けの Discord サーバー（ The Audio Programmer の#clap）などがあり、公開リポジトリでの issue や pull request を通じたやり取りも可能です。

VST3 を開発している Steinberg や AudioUnit を開発している Apple は、このようなコミュニティを開発者向けフォーラム以外では特に公開していないので、相対的に CLAP はコミュニティのやり取りが活発であると評価できるかもしれません。もっとも、コミュニティの Discord サーバーに開発チームのメンバーは参加していません。LV2 にはオープンなコミュニティが存在し、LV2 の開発者も含めてやり取りできます。ただし、2022 年でもメーリングリストと IRC が主な手段であるようです。LV2 は github と gitlab にリポジトリが存在し、開発者個人とのやり取りはこれらでも可能です。

一方で CLAP には Bitwig 社と u-he 社の開発者のみがアクセスできる開発者限定の Discord サーバーの存在も issues のやり取りから認知され（ たとえば https://github.com/free-audio/clap/issues/147#issuecomment-1206654119 ）、完全にオープンなコミュニティというわけではありません。github の freeaudio/clap リポジトリで公式サイトとされている cleveraudio.org も u-he の代表である Urs Heckmann の個人サイトで、ソースは公開されていません。LV2 はサイトのソースも公開されており、pull request も送れます。

1.3　CLAP 正式版（バージョン　1.0）が登場した背景とその意義

CLAP は 2022 年 6 月に正式版が公開されましたが、CLAP の開発そのものの歴史は長く、2014 年に github で開発が始まっており、Hacker News でも作者自らコメントを求めるかたちで記事化されています。CLAP が急速に注目されるようになったのは、プラグインベンダーである u-he が、VST3 を置き換えうる次世代フォーマットとして期待し、開発者が所属する Bitwig と提携してプラグインのエコシステムを作り上げるという目論見で、大々的にアナウンスするようになったためです。

u-he で次世代フォーマットを求めるようになった背景には、Steinberg が VST3 への移行促進のために行った VST2 のサポート廃止と VST2 SDK の非公開化があると考えられます。これは VST2 SDK をバンドルしているさまざまな VST プラグインプロジェクトの GitHub リポジトリに対して Steinberg が削除要求を送るほど徹底していました。従前から VST2 プラグインを開発していたベンダーは引き続き VST2 プラグインを開発できますが、新たな開発者は Steinberg の SDK をダウンロードできなくなったので、VST2 SDK を利用した製品が公開できないということになりました。（ LMMS プロジェクトで使われていた、VST2 互換の API をオープンソースで実装した VeSTige というコードが存在するので、これを利用しているプロジェクトは今でも存在します。）

一方で移行先となるはずの VST3 は、VST2 と比べると複雑で根本的なプログラムの書き換えが必要になり、その努力が必要になる割には VST3 非対応 DAW も多い状態でした（ ここには、VST3 はホスト側の仕様があまり明確にドキュメントされていないという問題も影響したと考えられます）。VST が特に Windows プラットフォームでは支配的なプラグインフォーマットであることを考えると、驚くほど浸透していませんでした。「 VST2 の次を VST3 の代わりに CLAP で取りに行こう」という計画が立ち上がっても、不思議ではなかったといえます。もっとも VST3 は 2022 年にはだいぶ DAW でサポートされるようになりプラグイン市場でも普及してきたので（ たとえば Kontakt など

の Native Instruments 製品が VST3 への移行を完了しています)、VST3 が座るべきポジションを奪うことにはならなかったというべきかもしれません。

オーディオプラグインフォーマットのエコシステムが成立するためには、サポートする DAW とサポートするプラグインの双方が揃っていることが不可欠です。DAW を開発する Bitwig とプラグインベンダーとしてそれなりのポジションにある u-he が提携して促進していることには一定の強みがあります。特に LV2 はこの面では強くなかったので、LV2 が獲得できそうな立ち位置に届きつつあるともいえます(一方で LV2 の主戦場である Linux デスクトップ環境のオープンソースプラグイン市場では VST3 も使われつつあります)。

いずれにせよ、オーディオプラグインフォーマットにとって、バージョン 1.0 公開は大きな意味をもちます。プラグインフォーマットはプラグインベンダーと DAW ベンダーが採用してその規格に則って製品を開発し販売することで利用されるようになります。そこでは ABI すなわちバイナリレベルの互換性が絶対に必要になります。プラグイン API を破壊的に変更することはできなくなります。互換性のない API には、DAW ベンダーもプラグインベンダーも追従してくれません。逆にいえば、DAW ベンダーもプラグインベンダーも、ABI が安定化したのであれば、安心してそのプラグインフォーマットをサポートできるようになります。

1.4 他のプラグインフォーマットとの比較

オーディオプラグインフォーマットとしては VST、AudioUnit、LV2 といったものが存在します。どのフォーマットにもさまざまな長所・短所があります。簡単に列挙しておきます。

- VST2: C/C++、多重継承による拡張機能、Win/Mac 用、20 世紀からありプラグインは最多、プロプラエタリライセンス、もう SDK をダウンロードできない
- VST3: C++、COM ライクなクエリーインターフェース(やや難)、多数のプラグイン、OSS (steinbergmedia/vst3sdk, GPLv3) or 商用ライセンス
- AUv2: Objective-C / C++、GitHub 上にソースがある (apple/AudioUnitSDK - 完全性は不明)
- AUv3: Objective-C / C++、Mac or iOS、App Extension によるプロセス分離、 ソースは無さそう
- LV2: C、クエリインターフェースによる拡張機能(やや難)、OSS (gitlab/lv2/lv2, ISC ライセンス)
- CLAP: C, クエリインターフェースによる拡張機能、OSS (free-audio/clap, MIT)

Apple の AudioUnit 以外はデスクトップのクロスプラットフォーム(Win/Mac/Linux) 用と考えて良いです(VST が Linux をサポートするようになったのはだいぶ最近になってからではあります)。

この他、Reaper の ReaPlug や ProTools 用の AAX など、汎用ではなく特定の DAW 専用のプラグインフォーマットで著名なものもいくつかあります。

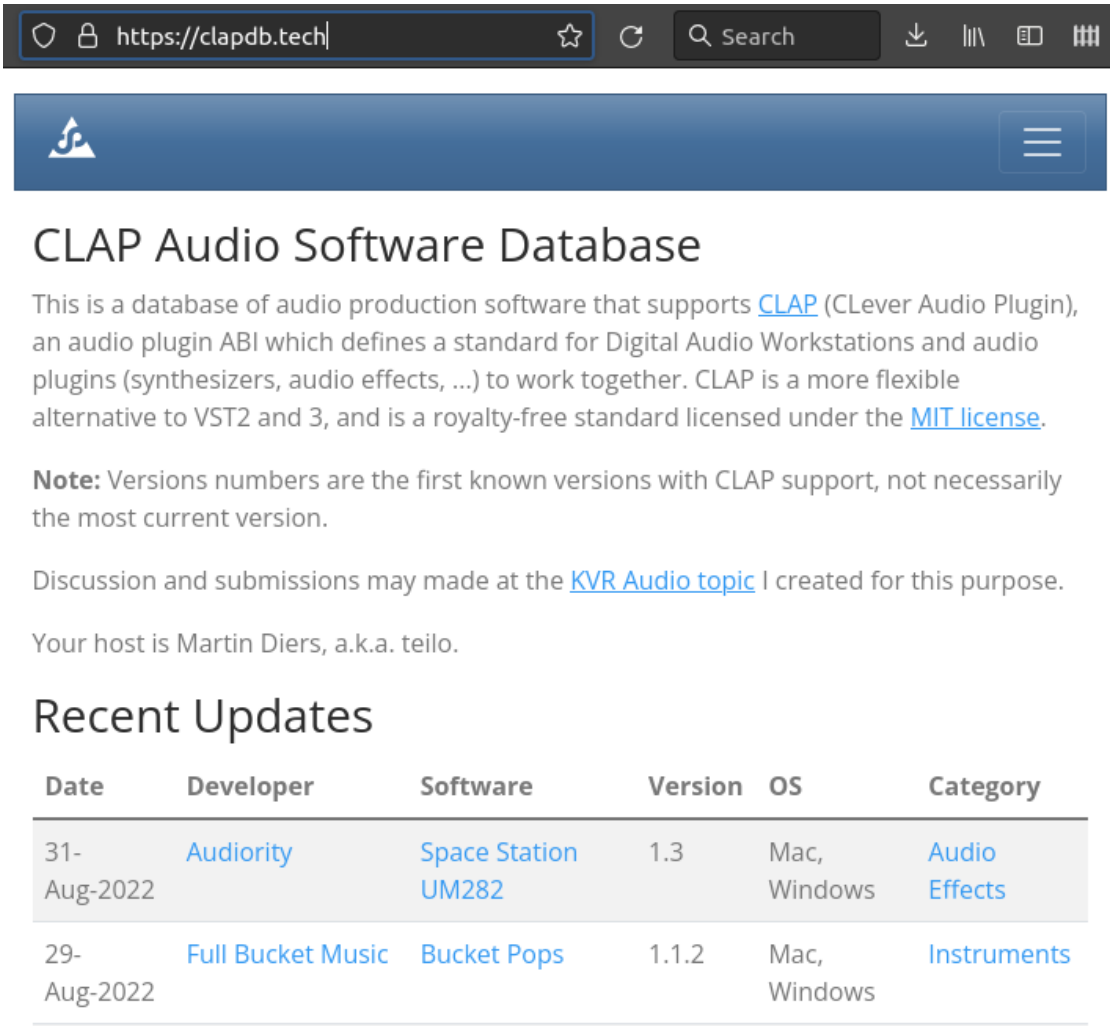
第 2 章

CLAP を使う

2.1　CLAP をサポートしているソフトウェアを発見する

ここでは、筆者が知っている CLAP 対応ソフトウェア（ DAW およびプラグイン）、およびそれらの探し方をいくつか紹介します。すでに知られているものを数多く紹介することはありません。

2.1.1 clapdb



▲図 2.1　CLAPDB

2022 年 10 月の時点では、CLAP をサポートしているソフトウェアを発見する、最も手軽な方法は、CLAP 公式リポジトリからもリンクされている clapdb https://clapdb.tech/ を見ることです。7 月〜8 月で合わせて 30 本ほどエントリーされています。9 月にも 30 本近く追加されています(うち半分ほどは日本の A.O.M. 社の製品)。

実際には、このリストは限定的で、またオープンソースではないソフトが多く、筆者はこれらについてはほぼ確認できていません。

2.1.2 JUCE プラグインの CLAP 対応ビルド

JUCE はオーディオプラグインのようなオーディオアプリケーションを開発する時に便利なフレームワークです。オーディオ処理と GUI 処理を抽象化して独自のフレームワークに吸収して、複数のオーディオプラグインフォーマットに対応するグルーコード（抽象化された JUCE の部分とプラグインフォーマット API の繋ぎこみの部分）の開発を肩代わりするもので、オーディオプラグイン開発で幅広く使われています。

CLAP 開発チームがこの JUCE の CLAP プラグインフォーマットへの繋ぎこみ部分を実装した clap-juce-extensions というモジュールを公開していて、これを使うことで JUCE プラグインが CLAP をサポートできるようになっています。surge synthesizer がこれを利用しているリーディング・ユースケースです。

GitHub 上に存在している JUCE アプリケーションとしてのオーディオプラグインのいくつかが、この clap-juce-extensions を利用してすでに CLAP 版のビルドに対応しています。以下は筆者が個別に CLAP 対応を確認した JUCE プラグインの GitHub リポジトリです（2022 年 8 月時点、clap-host あるいは qtractor で確認）。

- Chowdhury-DSP/BYOD
- TheWaveWarden/odin2

（以降、本書では GitHub のリポジトリを `xxx/yyy` と表記することがあります。）

クローズドソースのプラグインでも、Vital 1.5 などが CLAP 版の実験的なビルドを提供しています（バージョン 1.5 はソース非公開です）。

他にも asb2m10/dexed などが CLAP 対応ビルドのサポートを含めていますが、オートメーション対応の問題で開発継続中となっているのと、実際に試してみたらクラッシュが頻発したので、このリストには含めていません。jatinchowdhury18/AnalogTapeModel も CLAP 版のビルドに対応しているのですが、少し使ってみたら音声出力がなくなってしまう結果が何度か発生したので、実用する前に修正を待ったほうが良さそうです（プラグイン側の問題とは限らず、ホストとの相性でクラッシュすることもあります）。

これらはいずれも 2022 年 8 月時点で clapdb に含まれていないものです。JUCE を利用したオーディオプラグインはこの小さなリストよりもずっと多く存在し、それらに clap-juce-extensions をモジュールとして追加して CLAP 対応ビルドを追加するのは比較的簡単なので（ただし CMake で構成されている必要があるので、プロジェクトによっては CMake に移行するのがより大きな課題かもしれません）、今後 CLAP 対応プラグインがこの領域で増加することは大いに考えられます。

2.1.3 DPF プラグインの CLAP ビルド

LV2 プラグイン開発でたまに使われている DPF でも、2022 年 9 月から CLAP をサポートするようになりました。これを受けて Dragonfly-Reverb も正式に CLAP に対応しています。DPF は jpcima/string-machine や clearly-broken-software/ninjas2 といったプラグインでも使われており、僅かな変更で CLAP 版をビルドできる見込みが高いでしょう。

2.1.4 その他のプラグイン開発フレームワークの CLAP 対応（待ち）

同様に、プラグイン開発フレームワーク iPlug2 も公式リポジトリの clap ブランチで CLAP サポートの開発が進められているので、iPlug2 で開発されたプラグインが CLAP 対応するのも難しくはなくなるでしょう。（ iPlug2 を利用したプラグインは GitHub にはあまり多く存在しないので、すでに利用している開発者が試すという筋書きが中心になりそうです。）

2.2 CLAP プラグインのセットアップ環境

一般的に、CLAP を使うためには、CLAP プラグインと、CLAP をサポートしている DAW を PC にインストールしなければなりません。DAW のセットアップについては、一般的なアプリケーションのセットアップと何ら変わることは無いので、ここではプラグインのセットアップについてのみ説明します。

CLAP プラグインは単一のバイナリファイル`*.clap`で構成されます。DAW などの CLAP ホストは、CLAP プラグインを所定のパスから検索します。CLAP ヘッダーファイル `clap/entry.h` にこの記述があります。CLAP の「仕様」は独立したドキュメントとしては整備されておらず、コードコメントが仕様を説明しているということになっています。

- 環境変数 CLAP_PATH で指定されたパス（ Windows では';'、それ以外では':'で区切る ）
- システムパス
  - Linux: `~/.clap:/usr/lib/clap`
  - Windows: `%CommonFilesFolder%/CLAP/;%LOCALAPPDATA%/Programs/Common/CLAP/`
  - MacOS: `/Library/Audio/Plug-Ins/CLAP:~/Library/Audio/Plug-Ins/CLAP`

環境変数 CLAP_PATH は開発以外の目的で使用すべきではありません。

2.3 CLAP をサポートする DAW

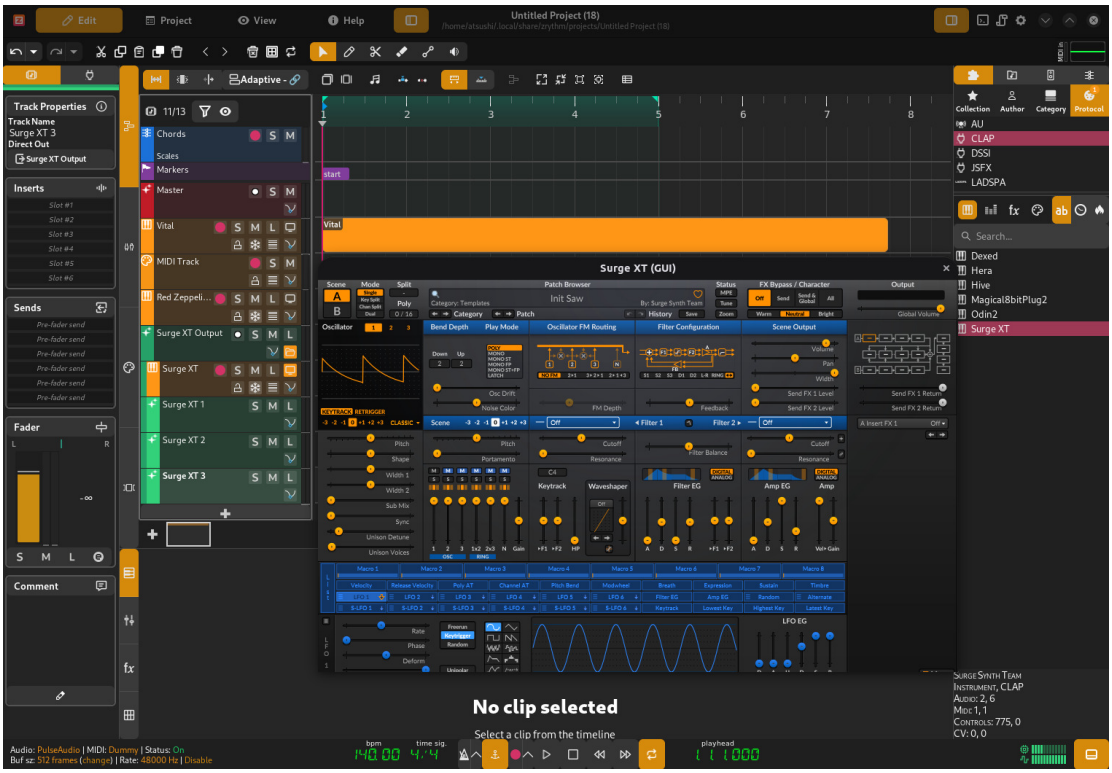
2022 年 8 月末時点で、CLAP をサポートしている DAW には次のものがあります。なお、開発用にはよりシンプルなホストを利用できます。複雑な DAW 上にデバッガーをアタッチする必要はありません。

2.3.1 Bitwig Studio（ 4.3 以降）

CLAP の開発者は Bitwig の従業員であり、CLAP を全面的に推しているのが Bitwig です。Bitwig Studio は、VST3 にとっての Cubase のような存在です。CLAP 正式版のサポートは Bitwig Studio 4.3 以降でのみ有効です。筆者は Bitwig Studio4.2 までのライセンスしか有していないので最新版での動作は確認していません。

2.3.2　Zrythm

Zrythm は比較的新しい DAW で、オーディオプラグインのホスティング機能は DPF の作者が開発している Carla というエンジンを利用しています。Carla が CLAP をサポートするようになったので、Zrythm にもその新機能が取り込まれて CLAP がサポートされるようになりました。

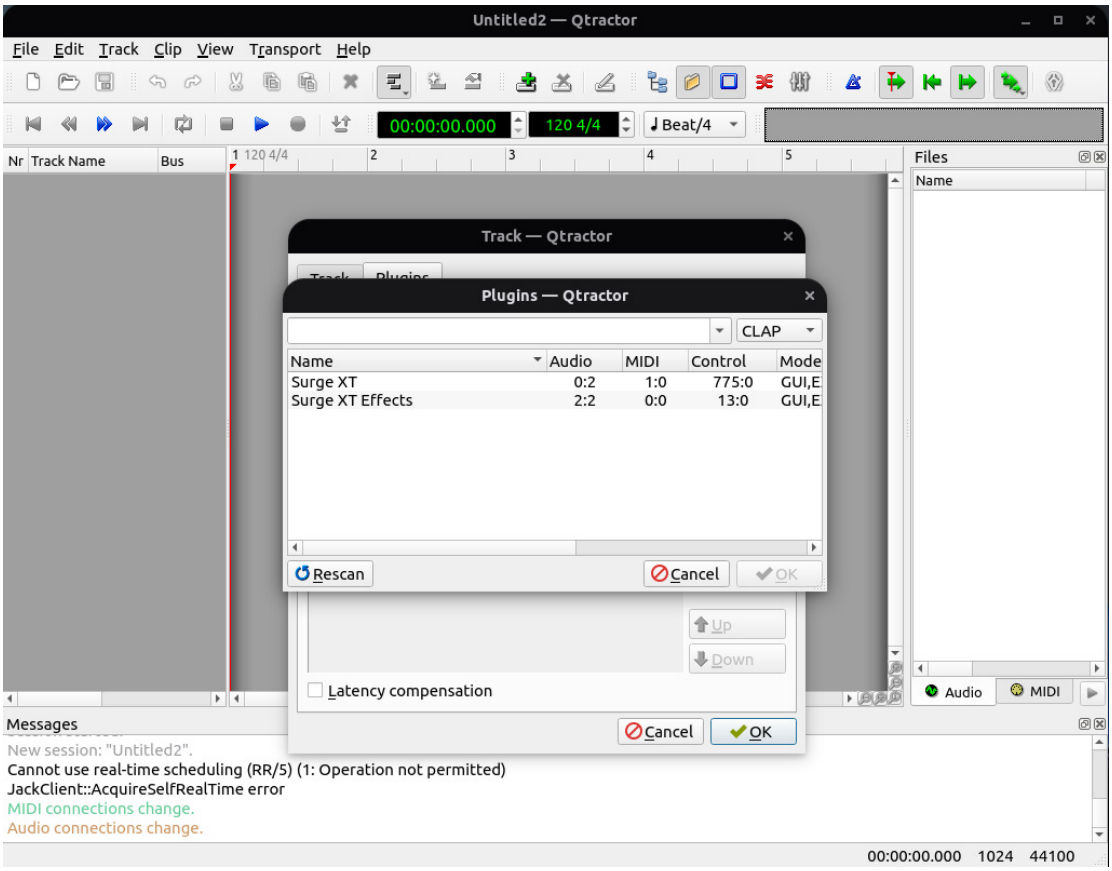


▲図 2.2　Zrythm の CLAP サポート

2.3.3　QTractor

QTractor は Qt で作られていて、比較的シンプルな機能からなる DAW です。開発者が特に CLAP 開発者コミュニティに属している様子はなく、VST3 対応も比較的近い時期に行われていたため、類似の対応作業が早かったのではないかと筆者は想像します。

QTractor でプラグインのスキャンが実行されると、CLAP プラグインがリストアップされるようになります。表示するプラグインの種別を CLAP に限定するとわかりやすいでしょう。



▲図 2.3　QTractor の CLAP サポート

2.3.4　その他

2022 年 10 月の本書執筆時点で CLAP をサポートしているのは以上ですが、Cockos が Reaper で CLAP をサポートしようとしているという情報が CLAP コミュニティで観測されるので、いずれ Reaper でもサポートが追加されるかもしれません。

2.4　CLAP をサポートするプラグイン

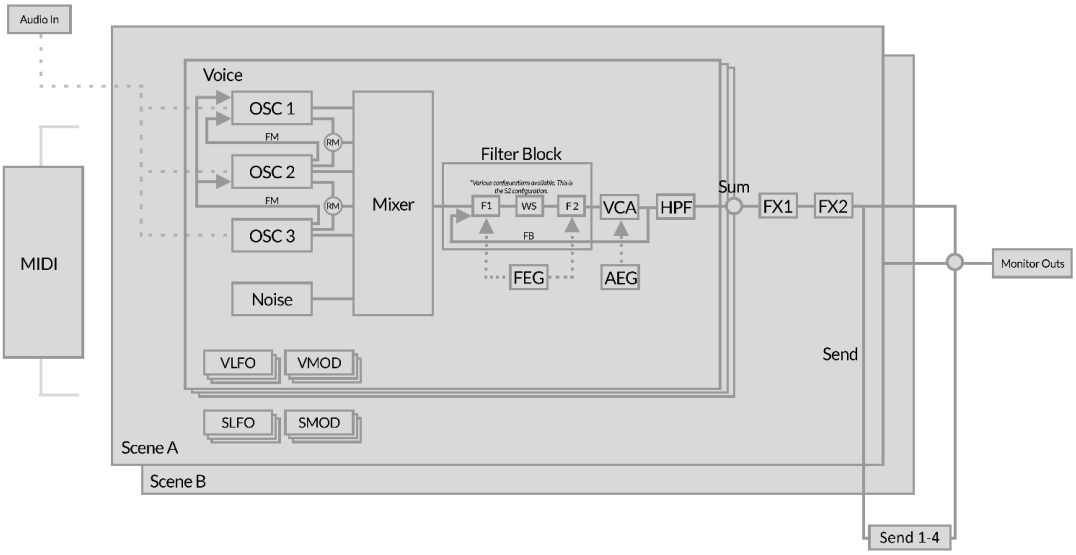
DAW と比べるとプラグインの数は多いですが、CLAP をサポートするプラグインはまだまだ少なく、ほしい音源が手に入るという状況ではありません。それでも VST や LV2 と同じ音源がすでに使えるというのは比較的恵まれている状況です。

本書は CLAP 音源をくまなく紹介するためのものではなく、筆者の手元でも動作が確認できた JUCE 製のプラグインの、主に存在のみを紹介します（音源の特色などを詳しく語ることは本書の趣旨からは外れるので割愛します）。

2.4.1 surge-synthesizer/surge

Surge は 2000 年代前半からあるオープンソースのシンセサイザーで、現在でも活発に開発されています。もともとは bitwig の創業メンバーが Vember Audio という屋号で有償製品として開発したものです。メイン開発者が CLAP コミュニティでもアクティブに活動しています。その特徴を言葉で説明するのは難しいので(Surge が自称するカテゴリーも"Hybrid Synthesizer"です)、スクリーンショットと信号処理構成図を掲載するにとどめます。

▲図 2.4　surge screenshot



▲図 2.5　surge の信号処理構成

2.4.2　TheWaveWarden/odin2

odin2 は 2020 年にリリースされたオープンソースのシンセサイザーで、もともとは 2016 年に（ソース非公開の）無償ソフトとして公開されていました。JUCE 製で、LV2 ビルドもサポートしていて、CLAP のサポートも clap-juce-extensions を利用して積極的に対応したことが想像できます。公式サイトで CLAP 版を同梱した Windows/Mac/Linux 用のパッケージがすでに配布されているので、手っ取り早く試すことができるでしょう。これも特徴を説明するのが難しいので（公式サイトの最初の説明が "Odin 2: A Synth to Please the Gods / The sound of this 24-voice polyphonic beast will take you from your studio right to Valhalla." なので察してください）、これもスクリーンショットの紹介にとどめます。

▲図 2.6　odin2 screenshot

2.4.3　Chowdhury-DSP/BYOD

BYOD はさまざまなコンポーネントを自由に定義してつなぎ合わせることができるギターディストーションのエフェクターです。こちらのスクリーンショットでは Nirvana を選択していますが、いくつかのギタリストやバンド名のプリセットが用意されていて、セットアップ済のプリセットを簡単に選択できるので、その範囲だけでもわかりやすく使えるでしょう。

▲図 2.7　BYOD screenshot

第 3 章

CLAP 開発ことはじめ

この章では、CLAP プラグインや CLAP ホストの開発のスタートラインに立つための情報をまとめます。

3.1 開発環境

CLAP の開発には CLAP API をネイティブコードとして実装できる何らかのコンパイラツールが必要になります。CLAP の API は C ヘッダーで定義されていますが、一般的には clap-helpers などを利用して C++ で開発することになるでしょう。Rust で CLAP プラグインを開発する clap-sys というライブラリも存在します。(筆者は Rust 開発の知見があるとはとてもいえないので、以降も clap-sys を用いた開発については基本的に言及しません。)

C/C++/Rust 開発環境は自分の手慣れたものを使えれば十分でしょう (emacs, vim, vscode, CLion, Xcode, Visual Studio など)。プラグインのデバッグには、プラグインのテスト用ホストを別途ビルドしてデバッグし、プラグインをロードするアプローチが有効です。

C/C++/Rust 以外の言語でオーディオ処理で求められるリアルタイム処理に適性のある言語はそう多くないと思われます。筆者には Zig と Google が 2022 年に公開した Carbon くらいしか思い当たりません(Carbon はまだ実用段階ではありません)。ガベージコレクションを含む動的メモリ確保の機構があったり、実行時コード生成(これも停止時間が未知数) を伴う JIT コンパイラーがあったりすると、リアルタイム要件を満足できなくなります。Go や Julia やはこれらが理由で対象外になります。Swift の動的メモリ確保も停止時間が未知数でアウトになる厳しい世界です。Swift forum の Realtime Threads with Swift という議論スレッドが参考になります。

3.2 CLAP 関連の github リポジトリ

3.2.1 free-audio/clap

CLAP の API 仕様を定義するヘッダーファイルがここにまとまっています。ドキュメントとしての「仕様」は整備されておらず、コードコメントが仕様を説明しているということになっています。API 仕様のうち拡張機能(この意義については CLAP プラグインの基本的な仕組みの章で説明します) のドラフト仕様は、`include/clap/ext/draft` ディレクトリに (バージョン 1.0 の時点では)

含まれており、ここに含まれる API は将来破壊的に変更される可能性があります。

最新版はもはやバージョン番号が 1.0.0 ではないのですが、API には破壊的変更が加えられてはいないはずです。コードコメントには説明（実質的には「仕様」）の変更がどんどん加えられています。

ちなみに、C だけで作られたプラグインは、筆者には（本書執筆時点では）発見できませんでした。

3.2.2 free-audio/clap-helpers

CLAP プラグインやホストを C++ で実装するときに有用なユーティリティ機能をいろいろまとめたライブラリのリポジトリです。CLAP 本体同様、header only のソースとなっています。たとえば、CLAP1.0 仕様で規定された拡張機能をプラグイン側で実装できるスタブが用意されていて、clap-helpers を使ってプラグインを実装する場合はそれらのメンバーをオーバーライドして実装するだけで済むようになります。あるいは、ホストからプラグインに渡されたホスト情報のポインターをもとに、ホスト側の拡張機能が実装されているという前提でそれらを呼び出せるようなプロクシー（`HostProxy`）も利用可能です。

CLAP のリポジトリ（free-audio/clap）にあるヘッダーファイルは、プラグインとホストの API を規定するものであり、インターフェースは基本的に実装を含みませんし、どんなプラグイン・どんなホストでも必要・有用なコードであっても、インターフェースのヘッダーに含めるよりは、（この free-audio/clap-helpers のように）実装基盤として切り離すほうが理にかなっています。（これと比べると、LV2 Atom の Utilities などは、仕様の中に実装の便利機能が含まれている例で、あまり治安が良くないと評価せざるを得ません。）

3.2.3 free-audio/clap-info

ローカル環境にインストールされているプラグインを列挙して、その詳細情報を表示する小型ツールです。LV2 を使った経験があれば、lv2-list と lv2-info に相当すると考えるとよいでしょう。こちらは引数なしで実行すると詳細なヘルプが表示されます。`-l` でプラグインを列挙してファイルパスを取得し、今度はそのファイルパスのみを引数に渡して詳細情報を表示するのが、典型的な使い方になるでしょう。

もしプラグインではなくプラグインホストを開発するなら、プラグインをロードしてスキャンする部分とプラグインのインスタンスを生成する部分を理解する必要がありますが、このツールのソースを追跡すれば基本的な流れがわかるでしょう。

3.2.4 glowcoil/clap-sys

CLAP プラグインを Rust で開発できるように CLAP API 互換の API を再現したライブラリです。このライブラリを使うのは実質的に CLAP の C API でそのままプログラミングするのとあまり変わらないでしょう。もう少し構造化されたライブラリとして prokopyl/clack などもあります（ただし API 設計がまだ実験的であり安定していないことに注意すべきです）。

3.2.5 robbert-vdh/clap-validator

CLAP プラグインが適切に実装されているかをテストできるユーティリティが含まれている Rust アプリケーションです。

3.2.6 free-audio/clap-plugins

CLAP プラグインの実装例が含まれています。CLAP プラグインはすでに surge をはじめ複数公開されているので、このリポジトリに含まれるプラグインを使いたい場面はそれほどないかもしれません。

また、GUI に Qt を使用しているのは悪い実装なので、このリポジトリを CLAP プラグインの見本として使うべきではありません。この作者も（ CLAP 仕様の開発者なのですが）後になって「 プラグインの GUI に Qt を使うのは危険」と README に明記しています。プラグインホスト（ DAW）が GTK などでプラグインとは別のアプリケーションループを構築していると、正常に動作しないためです。Linux デスクトップ上で動作するプラグインでは、どのような GUI フレームワークでも正常に動作するための最大公約数として X11 の API を使うことが推奨されています。JUCE や DPF はこの制約に従って X11 の API で実装されています。

3.2.7 free-audio/clap-host

シンプルな CLAP ホストの実装例が含まれています。プラグインのデバッグ用途では、これくらいシンプルなホストを使うのがよいでしょう。ドキュメントはあまり整備されておらず、基本的な使い方の説明もありませんが、コマンドラインで`-p`と`-i`の引数を使用して特定のプラグインをロードして適用する使い方になります。

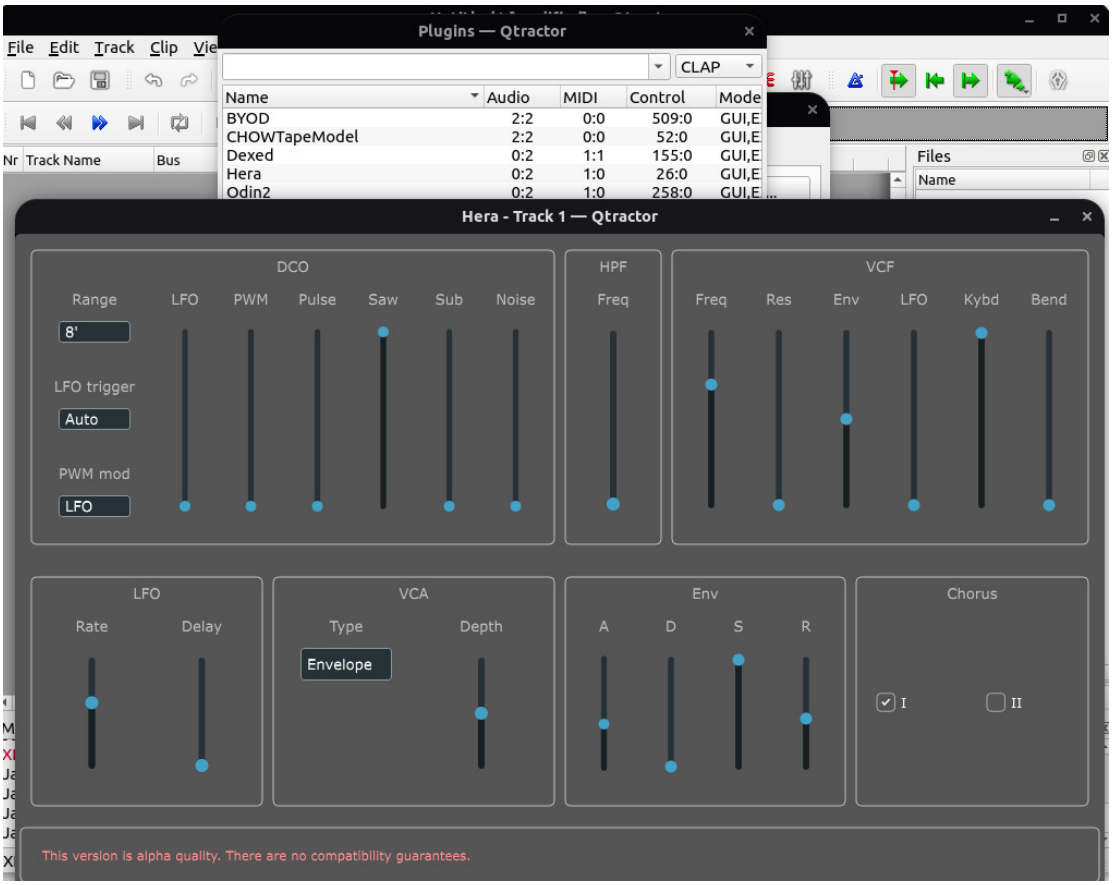
`./build/host/clap-host -p（.clapファイル）-i（プラグインのインデックス）`

`-h`または`--help`で使い方が表示されます。`-?`では何も出てきませんし、引数を指定しないと何も機能しないウィンドウが表示されます。

3.2.8 free-audio/clap-juce-extensions

clap-helpers を活用して作られた、JUCE プラグインを CLAP 対応にするための JUCE モジュールです。JUCE7 を前提に開発されていて、CMakeLists.txt に組み込むやり方であれば簡単に利用できます。Projucer を使うプロジェクトで利用する方法も README に記載されています（ 本書ではそこまで踏み込まないので、興味のある読者は独自に試してみてください）。

筆者は JUNO-60 のエミュレーター音源である Hera (jpcima/Hera) を clap-juce-extensions をドキュメント通りに CMakeLists.txt にパッチを当てて CLAP 版をビルドできています。



▲図 3.1　Hera synthesizer を筆者が独自にビルドした CLAP 版

3.3　CLAP プラグイン開発を始めるアプローチ

3.3.1　clap-juce-extensions や DPF を使って開発する

CLAP プラグインの機能の大半は一般的なオーディオプラグインの機能です。オーディオプラグインは、複数のホスト（ DAW）が複数のプラグインフォーマットを一様にサポートするのが一般的であり、またプラグイン開発者も特定のプラグインフォーマットでしか利用できない機能を特別に実装するメリットは薄いので、よほど CLAP の新機能を活用したい、あるいは売り込みたい動機が無い限りは、汎用オーディオプラグイン SDK がサポートする範囲の機能を使って CLAP プラグインを開発するほうが、開発の知見も流用できて楽になるはずです。

JUCE を使ってオーディオプラグインを開発した経験があれば、clap-juce-extensions をプロジェクトの CMakeLists.txt でちょっと追加するだけで CLAP 版をビルドできる可能性があります（ 前述の筆者の CLAP 版 Hera は 4 行のパッチで出来ました）。

DPF の CLAP 対応も、筆者は自ら試したことはありませんが、DPF の開発者が dragonfly-reverb に適用した CLAP 対応に必要な変更を見ると、ひとつのプラグインにつき、`Makefile` の `TARGETS`

に clap を追加して、#define DISTRHO_PLUGIN_CLAP_FEATURES "audio-effect", "reverb", "stereo"のような定義を 1 行追加して、追加でプラグイン ID を DISTRHO_PLUGIN_CLAP_ID として別途定義するだけで対応できているようです。

このアプローチであれば、CLAP プラグインの開発方法を勉強するというのは、実質的に「 JUCE でのプラグインの開発方法を勉強する」とか「 DPF でのプラグインの開発方法を勉強する」というのとほぼ変わらず、CLAP 独自のやり方を学ぶ必要は無いということになります。ビルドした CLAP プラグインは*.clap という独特な拡張子なので、ソースツリーで find . -name *.clap などを実行すればすぐに発見できるでしょう。

3.3.2 フルスクラッチで始める場合

汎用オーディオプラグイン SDK の類を使うのは性に合わない、可能な限り依存性を排除して CLAP の SDK だけでフルスクラッチで開発したい、という場合、プラグイン開発は次のような clap_entry を定義する C/C++ プログラムを作るところから始めることになるでしょう。

```
#include "clap/clap.h"

const  clap_plugin_entry_t  clap_entry = { ... }
```

clap_plugin_entry_t は CLAP ヘッダーファイル clap/entry.h に定義されています(それ以上の情報は読者が IDE 等で定義を追及していってください)。clap-helpers を使って C++ でプラグインを定義する場合、この C のエントリーポイントの API とつなぎ合わせる部分は、free-audio/clap-plugins のソースツリーに含まれる plugins/clap-entry.cc というソースファイルが参考になります。この clap_entry が DAW によってロードされる仕組みについては、次章で説明します。

第 4 章

CLAP のコア部分の仕組み

4.1 オーディオプラグインの一般的な仕組み

4.1.1 概要

CLAP は汎用オーディオプラグインフォーマットです。汎用オーディオプラグインフォーマットは、任意の DAW で任意のプラグインを利用できるようにするために作られる規格です。オーディオプラグインは、プラグインフォーマットが規定する API を実装し、DAW は、プラグインフォーマットが規定する API の実装コードをバイナリプログラム（ DLL など）からロードします。

2022 年の本書執筆時点では、オーディオプラグインフォーマットは複数の規格が並立していますが、基本的な機能は大まかには次の通りで、大きく違うものではありません。

- 動的ライブラリに類するかたちでロードでき、複数のインスタンスを生成できる
- ステレオ等のオーディオ入力と MIDI 等のコントロール入力を「 オーディオ処理」のサイクルで受け取って、オーディオ出力およびコントロール出力に変換する
- 外部から操作でき、変更を GUI に通知できる float のパラメーターがある（ その他の型をサポートすることもある）
- MIDI などの「 イベント 」を受け取ったり通知したりできる（ オーディオ処理と連動するかしないかは規格による）
- プラグインのパラメーター等を状態として保存・復元できる
- メタデータとしてプラグイン名やベンダー名をもち、プラグインホスト（ DAW など）がシステム上に有るプラグインをリストアップできる
- ホスト DAW からサブウィンドウに表示できる GUI をもつ

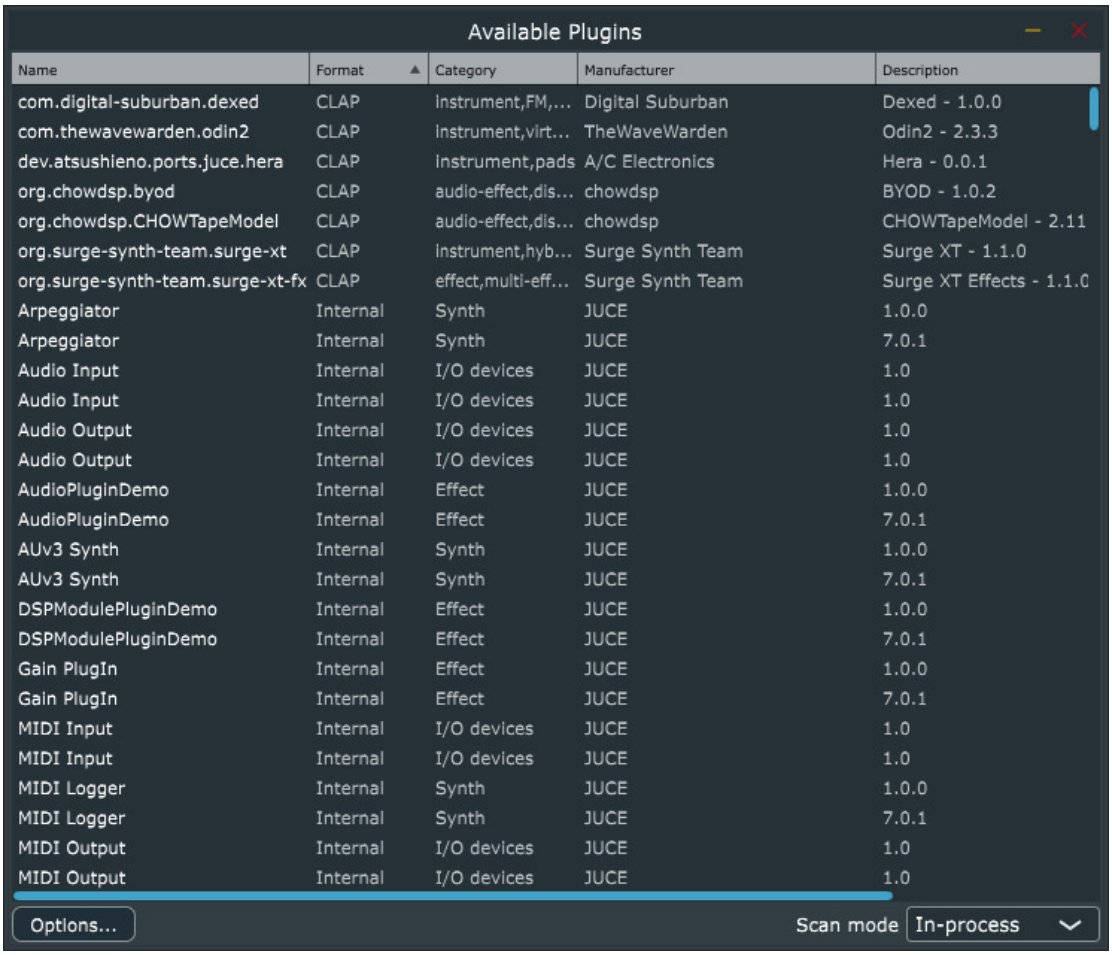
これらだけであれば、素朴に設計するのも実装するのもそんなに難しくはありません。特にプラグインフォーマットとは API であって実装コードである必要はないので、宣言してしまえば大部分の仕事は終わりです。free-audio/clap のリポジトリの内容も、ほぼ API のみなのでシンプルです。

もちろん、適当に API を作っただけではプラグインでも DAW でも対応してもらえないので、実際にプラグインフォーマットを定着させるためには、それ以上の仕事をすることになります。魅力的な追加機能を用意したり、それらが多数になっても一貫性を保てたり、簡単に実装できたり…といった設計の調整力が必要になります。CLAP はここに小さからぬ労力を割いており、CLAP の長所が

出ている側面でもあります。

4.2 CLAP プラグインのスキャンとロード

DAW ユーザーがオーディオプラグインを使うとき、最初にローカル環境にインストールされているプラグインのリストを取得する必要があるのが一般的です。リストには次のスクリーンショットのような内容が含まれます。



▲図 4.1　インストールされているオーディオプラグインのリスト

これは JUCE の AudioPluginHost というプログラムに筆者が手を加えて CLAP プラグインも列挙できるようにしたもので、実用性はありません（プラグインを正しくロードできません）が、DAW を利用したことのある読者は、このようなリストをセットアップした経験があるでしょう。

第 2 章で説明したとおり、プラットフォームごとに CLAP プラグインを検索するパスが決められているので、その中から拡張子が `.clap` であるものを、`dlopen()` や `LoadLibrary()` などプラットフォーム別のやり方で動的ライブラリとしてロードしていきます。ひとつの `.clap` ファイルには複数

のプラグインが含まれていることがあります。

4.2.1 プラグインのロードとインスタンス生成

ホストを開発するとき、プラグインのライブラリをロードした後のプラグインインスタンス生成については、`include/clap/entry.h` のコードコメントを参照すると良いでしょう。このヘッダーファイルには、プラグインのエントリーポイントとなる `clap_entry` の宣言が含まれています。この宣言通りの変数を、自分のプラグインのライブラリで定義すれば、それがホストからロードされることになります。ロードしたら `init()`、正常終了時にアンロードする前には `deinit()` を呼ぶ必要があります。

実際のプラグインインスタンスの生成は `get_factory(CLAP_PLUGIN_FACTORY_ID)` という `clap_entry` のメンバー呼び出しで `clap_plugin_factory_t` 型の変数を返させてから行います。この型と `CLAP_PLUGIN_FACTORY_ID` という定数は `entry.h` ではなく `plugin-factory.h` という別のヘッダーファイルで定義されています。

高度な話題: 高速なプラグインのロード

CLAP は公式発表では fast scanning をサポートしていると公言しているのですが、CLAP のメタデータは実のところライブラリに格納されていて DLL 等をロードしないと取得できず、この点では LV2 に劣ります。さらに、VST 3.7.5 も LV2 のようにプラグインのメタデータを `moduleinfo.json` というファイルに出力できるようになったので、最新の VST3 の仕組みよりも遅いことになってしまいました（とはいえ、VST3 プラグインもホストも最新仕様に追従するまではしばらくかかるでしょう）。プラグインの factory のインスタンスさえ取得すればメタデータを取得できるので、プラグイン本体のインスタンスを生成しなくても済む、という点においてのみ VST より高速だといえることになります。DLL をロードすると全ての static データの初期化が発生するため、JUCE の BinaryData 初期化などで DLL の mmap ロード以上のコストがかかるということもありますし、プラグインによってはこの DLL ロードとメタデータ取得の時点でライセンスアクティベーションダイアログを出してくるものがあり、これらが迅速なプラグインリストの生成を邪魔してくるわけです。DLL をロードする方式だとそういった問題があるため、JSON 等でメタデータを生成して解析できるようにしたほうが良い、というコミュニティ（筆者も含む）からはフィードバックが出ているのですが、開発チームは「JSON を解析しないといけなくなるから嫌だ」というレベルで否定的な見解を示しており、改善が施されるかは不透明なところです。

ちなみに、プラグインの列挙が仕組み上高速に完了するとしても、プラグインリストをキャッシュする DAW の仕組みがなくなるというとは限りません。プラグインリストキャッシュには「ロードに失敗した」プラグインをリストから除外する、いわゆる allowlist/denylist を管理することも期待されるためです。

また、筆者が JUCE のホスティング API の応用として CLAP プラグインのホスティングを実装しようとしたところ、JUCE のプラグインメタデータを表す `PluginDescription` クラスには `numInputChannels`、`numOutputChannels` などのポート情報が含まれており、これらは CLAP プラグインでもインスタンスを生成しないと取得できない情報なので、結局プラグインのインスタンス生成を行わなければならず、高速なプラグインのスキャンは実現できませんでした（ポート情報は取得でき

ないということにも出来なくはないですが、JUCE の AudioPluginHost のようなアプリケーションで CLAP プラグインノードを生成したときにポート情報が正しく反映されない可能性があります)。

4.2.2 プラグインメタデータの定義とプラグインファクトリー

プラグインを開発するときは、ここまで説明してきたのとは「逆の」実装が必要になります。`get_factory()` 関数を自分で定義し、`clap_plugin_factory_t` 型の変数を返さなければなりません。この型の変数を返すためには、次の一連の関数を関数ポインターメンバーとして指定する構造体を定義しなければなりません。

- そのプラグインライブラリに含まれるプラグインの数を `get_plugin_count()` で返す
- `get_plugin_descriptor()` という関数で、指定されたプラグインのメタデータを `clap_plugin_descriptor_t` 型の変数で返す
- `create_plugin()` でプラグインのインスタンスを `clap_plugin_t` 型で返す

2 番目の `get_plugin_descriptor()` が返す `clap_plugin_descriptor_t` には、プラグインのさまざまなメタデータが含まれることになります。ID、名前、ベンダー名、バージョン番号など典型的な情報が多いです。詳しくは `include/clap/plugin.h` に含まれている `clap_plugin_descriptor_t` の定義を参照してください。

そして、`get_factory()` などのメンバーを含む `clap_plugin_entry` 型の変数 `clap_entry` をグローバル変数として定義して、他のプログラムが `dlopen()` などでこの CLAP プラグイン(`*.clap`)をロードした時に参照できるシンボルとしてエクスポートする必要があります(一般的には CLAP ライブラリのビルドが最終工程になって `clap_entry` シンボルが hidden 扱いで隠蔽されるようなことは無いので、エクスポートのために特別にビルド設定を調整することは無いでしょう)。

4.3 CLAP の拡張機構

続いてオーディオ処理の説明に入りたいところですが、CLAP ではオーディオ処理に必要不可欠なデータ構造が拡張機構に依存する珍しい規格なので、先に拡張機構について説明します。

オーディオプラグインの拡張性 (extensibility) とは、シンプルにいえば機能追加のための仕組みです。プラグイン規格に追加機能を持ち込むには、API を拡張しなければなりません。これを無計画に行うと、後から「やっぱりこの機能はいらなかった」とか「やっぱりこの API だとイマイチだから仕切り直そう」と思っても、後方互換性を全面的に破壊することになってしまいます。つまり、古いプラグインのコードが新しいプラグイン規格 SDK のバージョンでビルドできなくなったりします。

VST2 がこの無計画なスタイルで開発されてきましたが(20 世紀に作られた仕様であり、誰もが API の後方互換性を強く意識するような時代ではありませんでした)、VST3 ではこれが大きく変わりました。VST3 ではコア機能と拡張機能を切り分けて、拡張機能は Windows の COM 技術におけるクエリインターフェースの仕組みで動的に取得するようになりました。もし拡張機能の一部で API の進化が行き詰まったとしても、それ以外の部分は後方互換のままで利用できることになります。これは VST-MA (module architecture) と呼ばれる仕組みです。

よく VST3 は VST2 と比べて複雑だと言われますが、この拡張性に関する設計方針は、フレーム

ワーク開発者の間では広く受け入れられていて、LV2 でも CLAP でも同様の機構を C で実現しています。オーディオプラグイン規格以外でも、OpenSL などにも見られます。

何をもってコア機能とし、何をもって拡張機能とするかは、これらモダンなプラグインフォーマットの間でも違いがあります。CLAP はだいぶ強めに「拡張機能化」を実現していて、オーディオポートやノートイベント・パラメーターイベント用ポートの定義すらも全て拡張機能で実現しています。

4.3.1 CLAP 拡張機能へのアクセス

CLAP では、拡張機能がクリーンに整備されています。CLAP の拡張機能にはプラグイン拡張とホスト拡張がありますが、全てのプラグイン拡張機能が `clap_plugin_*_t` として定義されており、同様に全てのホスト拡張機能が `clap_host_*_t` として定義されています。プラグイン拡張の API は「プラグインが実装し、ホストが呼び出す」ものであり、ホスト拡張の機能は「ホストが実装し、プラグインが呼び出すもの」と理解しておけば OK です。

拡張の種類	実装者	利用者
プラグイン拡張	プラグイン	ホスト
ホスト拡張	ホスト	プラグイン

プラグインが実装する拡張は、`clap_plugin_factory_t` の `create_plugin()` で返される `clap_plugin_t` の `get_extension()` メンバー（関数ポインター）の実装となる関数で、ID に対応するものを返すかたちで実装します。たとえばプラグインで state を保存する機能（パラメーターが存在するプラグインではほぼ実装することになるでしょう）を実装する場合はこうです。

```
clap_plugin_state_t state_ext{my_plugin_state_save, my_plugin_state_load}; // それぞれ関数
void* my_plugin_get_extension(const clap_plugin_t* plugin, const char* id) {
    if (!strcmp(id, CLAP_EXT_STATE))
        return &state_ext;
    ...
    return nullptr;
}
```

プラグインがホスト拡張を呼び出す方法については少し追加説明が必要でしょう。これに関連してはまず、拡張ではありませんが、CLAP の plugin factory となる `clap_plugin_factory_t` の `create_plugin()` の定義について説明します。

```
const clap_plugin_t *(*create_plugin)(const struct clap_plugin_factory *factory,
  const clap_host_t *host,
  const char *plugin_id);
```

この 2 番目の引数は `clap_host_t*`という型になるのですが、これはホスト実装のポインターである必要はなく、プラグインが必要とするホスト機能を提供する実装でさえあれば十分です。`clap_host_t` には次のようなメンバーがあります（コメントを削っています）：

```
typedef struct clap_host {
   clap_version_t clap_version;
   void *host_data;
   const char *name;
   const char *vendor;
   const char *url;
   const char *version;
   const void *(*get_extension)(const struct clap_host *host, const char *extension_id);
   void (*request_restart)(const struct clap_host *host);
   void (*request_process)(const struct clap_host *host);
   void (*request_callback)(const struct clap_host *host);
} clap_host_t;
```

ホストのメタ情報のほか、ホスト拡張機能を取得するための `get_extension()`、ホストの main (UI) スレッドで処理を呼び出すための `request_callback()`、オーディオ処理を開始させるための `request_process()` や `request_restart()` などが定義されています。ホスト拡張機能はこの `get_extension()` で取得して使います。

4.4 CLAP プラグインのオーディオ処理

4.4.1 オーディオ処理部分の基本形

オーディオプラグイン規格は、それぞれ互換性が無いものですが、楽器やエフェクターとしての基本的なオーディオ処理の部分には共通の部分が多いです。以下に擬似コードで列挙します。

- `create()`: プラグインを生成する
- `activate()`: プラグインを有効化し、一般的にはリアルタイム処理モードに移る
- `process(audio_buffers, midi_buffers)`: オーディオ入力と MIDI 入力を受け取って出力を生成する
- `deactivate()`: プラグインを無効化し、一般的にはリアルタイム処理モードを脱ける
- `destroy()`: プラグインを破棄する

「MIDI 入力」と書いている部分は、実際には MIDI ではなくプラグイン規格ごとに異なりますが、これについては次の節でじっくり説明します。ここではオーディオ入力について掘り下げます。

4.4.2 `clap_plugin_t`: プラグインの実装

CLAP プラグイン開発者は、自分のプラグインを `clap_plugin_t` という型で定義することになります。これを `clap_plugin_factory_t` の `create_plugin()` で返すことで、プラグインのロードからの一連の流れが完成します。

`clap_plugin_t` は次のような内容になっています。このメンバー全てを説明すると長くなりすぎてしまうので割愛しますが、先の疑似コードと似たようなメンバーが含まれていることだけ確認してください。

```
typedef struct clap_plugin {
   const clap_plugin_descriptor_t *desc;
   void *plugin_data;
   bool (*init)(const struct clap_plugin *plugin);
   void (*destroy)(const struct clap_plugin *plugin);
   bool (*activate)(const struct clap_plugin *plugin,
                    double                    sample_rate,
                    uint32_t                  min_frames_count,
                    uint32_t                  max_frames_count);
   void (*deactivate)(const struct clap_plugin *plugin);
   bool (*start_processing)(const struct clap_plugin *plugin);
   void (*stop_processing)(const struct clap_plugin *plugin);
   void (*reset)(const struct clap_plugin *plugin);
   clap_process_status (*process)(
       const struct clap_plugin *plugin, const clap_process_t *process);
   const void *(*get_extension)(
       const struct clap_plugin *plugin, const char *id);
   void (*on_main_thread)(const struct clap_plugin *plugin);
} clap_plugin_t;
```

4.4.3 オーディオデータとバス、ポート、チャンネル

「オーディオデータ」に渡ってくるのは、「全オーディオチャンネル」分の float あるいは double の配列です。最近は double を使う状況もありますが、一般的には float が使われるでしょう。「全オーディオチャンネル」とは、1 つ以上の「オーディオバス」ごとに割り当てられた「チャンネル数」です。オーディオバスは少しややこしい概念で、難しければ飛ばしても何とかなる概念です（飛ばした場合は「ステレオ」になると考えれば良いです）。もう少しちゃんと説明すると、オーディオバスとはチャンネルの構成に名前と特性がいくつか付いた構造です。具体的なものを挙げたほうがわかりやすいでしょう。

- mono
- stereo
- 5.1ch
- 7.1ch
- ambisonic

これらの名前は暗黙的に「メイン入出力」を含意していることが多いです。これらはそれぞれ固有のチャンネルを有しており、一般的にはチャンネルにも名前が付いています（"L", "R", "front left", "rear back right" など）

「メイン」があるということは補助的なものもある…というわけで、オーディオプラグインには「サイドチェイン」として直接再生するわけではなく波形の計算に補助的に利用するオーディオデータを渡すこともできます。LV2 では CVPort がサイドチェインを実現するためのものです。

ホスト DAW は、各トラックについて、オーディオプラグインに「利用できるオーディオバスの情報」を提示してもらい、トラックで利用できるバスを調整し、バスのオーディオバッファを準備します。メインのバスは一般的には 1 つになりますが、補助的にサイドチェインのバスが複数有効化される可能性があります。そして実際にオーディオ処理 `process()` を呼び出すとき、有効なオーディオバス全てに関連付けられたオーディオチャンネル分のバッファが、引数として渡されることになります。

VST の場合はバスと呼ばれますが、CLAP ではポートと呼ばれています。LV2 でもポートと呼ばれるのですが、これはバスに相当する概念ではなくチャンネルに相当する概念なので(つまり「 ステレオ」ポートにはならず「 左」ポートと「 右」ポートになる)、混乱しないように区別して捉えておかなければなりません。

CLAP では audio_ports 拡張機能を使用してポートの詳細情報を取得します。この拡張機能の構造体は次のように定義されています。

```
typedef struct clap_plugin_audio_ports {
   // [main-thread]
   uint32_t (*count)(const clap_plugin_t *plugin, bool is_input);
   // [main-thread]
   bool (*get)(const clap_plugin_t    *plugin,
               uint32_t                index,
               bool                    is_input,
               clap_audio_port_info_t *info);
} clap_plugin_audio_ports_t;
```

count() 関数でポートの数が返され、get() で clap_audio_port_info_t という詳細情報の構造体が返されます。これにはそれなりの量の項目が含まれるので詳しくは割愛しますが、識別子 id とチャンネル数 channel_count が含まれるので、ホストはこの情報をもとに、実際のオーディオ処理で、有効なポートのチャンネルの数だけオーディオデータのバッファをプラグインに渡して処理させることになります。詳しくは process() の項で説明します。

4.4.4 オーディオプラグインにおける MIDI サポートの基本

CLAP 開発チームがよく宣伝している機能のひとつが高度なノート命令ですが、これを理解するためには、そもそもオーディオプラグインはどうやって MIDI 命令を処理しているのか、理解しておく必要があるでしょう。

実のところ、オーディオプラグインでなまの MIDI メッセージをそのまま処理することはあまりありません。オーディオプラグインフォーマットごとに MIDI より高度な(データ幅や追加情報が多い)ノートイベントやコントロールチェンジイベントなどが規定されていて、MIDI 入力は DAW によって変換されてプラグインに渡されるのが一般的です。プラグインを開発できる SDK によっては、受け取ったイベントメッセージを MIDI メッセージに(DAW とは逆の方向に)変換して、VST や AU や LV2 の共通コードとして実装できるようにすることも多いです(JUCE など)。

オーディオプラグインが MIDI のようなイベントを受け取る入口は主に 2 つあります:

- イベントとして受信する: DAW で PC に接続された MIDI キーボードの鍵盤を押すと今のトラックのオーディオプラグイン設定で非リアルタイムに演奏されます(発音します)
- 演奏命令として受け取る: DAW に打ち込んだ内容を再生するとき、DAW は対象の全トラックでリアルタイムオーディオスレッドを用いて「 オーディオループ処理」を回します。1 回の処理はリアルタイムと言える間隔(10 ミリ秒程度)の間に全て完了しなければなりません。そうしないと不自然な「 空き」が生じてしまい、遅延やノイズの原因になります。このループの中に、演奏命令としてイベント列も含まれることになります。

ここで一つ気をつけないといけないのは、オーディオループはリアルタイムスレッドで回っていて、一方で MIDI キーボード等の入力は I/O を伴う非リアルタイムスレッドから受け取るということです。リアルタイムでオーディオ処理を回しているときに、ちょっとだけ時間を借りて MIDI 入力に対応する音声を生成して戻ってくる…ということはできません。

実際に DAW を使っているときは、トラックを再生しながら同時に MIDI キーボードを叩いていることがあり、この意味では受信した入力イベントは演奏命令にマージされると考えて良いでしょう。ただしそのタイミングは 1 回のオーディオループで処理される実際の演奏時間の長さによって変わります。もし仮に 1 回のオーディオループで 1000 ミリ秒分のオーディオデータが処理されるとしたら、オーディオループは 1 秒に 1 回しか回りません。その間のどのタイミングで MIDI キーボードの鍵盤が押されたとしても、ノートイベントが発生するのはその 1 秒単位のスライスの始点でのみということになります。

4.4.5 DSP コードの共通化

VST も AU も CLAP もそれぞれバラバラなパラメーターとイベントの機構をもっていますが、VST でも AU でも LV2 など他のフォーマットでもプラグインをリリースしたいと思ったら、それぞれのフォーマットに固有の DSP 処理を書くよりも、MIDI のような共通の音楽演奏命令のデータを使って記述するようにしたほうが、再利用性が高いです。

DSP でコードを共通化できるということになったら、JUCE のようなクロスプラットフォーム・マルチプラグインフォーマット用フレームワークのほか、FAUST や SOUL（ SOUL の権利は ROLI 社に残ってしまったので創始者の jules は今は新しく c-major という言語を作っているようですが）といったオーディオ処理に特化した言語、あるいは MATLAB のような言語環境も適用できる可能性が高くなります。

例として、オーディオプラグインフォーマットのようなものが登場すると、MDA という定番 DSP モジュールの集合体が mda-vst, mda-vst3, mda-lv2 といった感じで移植されますが、このオリジナルの MDA のコードは汎用的な DSP コードとして書かれています。各プラグインフォーマットのプロジェクトは、それぞれのプラグインの API を使った「ラッパー」となっているわけです。

この領域で新しく開発されているのが、MIDI 2.0 サポートの追加です。共通コードで MIDI 1.0 の表現力しか得られないのは残念な状態だったわけですが、MIDI 2.0 が利用できるようになれば、note expression や 32 ビットパラメーター（ CC や NRPN = Assignable Controller）、アーティキュレーションなどを処理できるようになります。CLAP では、規格のレベルで MIDI 2.0 メッセージをそのまま（ VST3 とは異なり、そのまま）処理できるようになっています。

4.4.6 各プラグインフォーマットにおけるイベント定義

VST2 の時代は、MIDI 入力はそのままのかたちでプラグインが受け取って処理できるようになっていました。一方でプログラムチェンジとパラメーターも利用できるようになっていたので、ある意味役割がかぶっていた状態でした。

Steinberg はこれを問題だと考えて、VST3 では MIDI 入力をパラメーターとしてプラグイン側でマッピングして処理させることにしました。その結果、VST3 ではなまの MIDI メッセージを受け取

ることができなくなりました。これはそれなりに大きな副作用があり、まず VST3 プラグインでどんな DAW でも一意に復元できるような MIDI 出力が出せなくなりました（ 全て VST イベントとして出力されるため）。そして JUCE のように VST3 イベントを MIDI メッセージに変換したうえでオーディオ処理に渡す仕組みがあって、かつプログラム番号も MIDI のプログラムチェンジとならずにそのまま渡される仕組みになっていると、DAW には MIDI のプログラムチェンジを入力したはずなのに、プラグイン側にはプログラムチェンジとして渡されない…といった問題もありました。JUCE のように「 プラグイン側ではホストからのイベントは全て MIDI メッセージに変換してその範囲で処理する」仕組みになっていると、この影響をストレートに受けることになります。

CLAP は、VST3 とは異なり、MIDI 1.0 イベント、MIDI 2.0 イベント、CLAP イベントの 3 種類がサポートされており、DAW はどの入力もそのままプラグインに渡せば良いということになります。

VST3 は「 役割が重複したら困るだろうからホストが全部われわれの VST イベントに変換するのでそれを使え」という姿勢ですが、CLAP の場合は「 複数の入力イベントの種類で役割が重複するかもしれないが、その解決方法は自分でやれ」ということになります。言い換えると、CLAP の場合、CLAP イベントと MIDI イベントが排他的かどうかは明記されていません。これに比べると、MIDI 2.0 UMP 仕様では、「 MIDI 1.0 プロトコルで処理可能な UMP」と「 MIDI 2.0 プロトコルで処理可能な UMP」が明確に規定されています。たとえば、MIDI 2.0 プロトコルで MIDI 1.0 チャンネルボイスメッセージを送ることは許されません。

4.4.7 CLAP の process 関数

この節の冒頭で、オーディオプラグインの一般的な処理フロー（ create, activate, process, deactivate, destroy）について説明しましたが、CLAP プラグインでは、`clap_plugin_t` にこれらの一部が含まれています。`process()` 関数については特に `clap_process_t` という複雑な引数構造体を持っているので、説明が必要になるでしょう。この構造体は `include/clap/process.h` に定義されていますが、ここではコードリストに一部のメンバーのみを抜粋して示します。

```
typedef struct clap_process {
   ...
   uint32_t frames_count;
   ...
   const clap_audio_buffer_t *audio_inputs;
   clap_audio_buffer_t       *audio_outputs;
   uint32_t                   audio_inputs_count;
   uint32_t                   audio_outputs_count;
   const clap_input_events_t  *in_events;
   const clap_output_events_t *out_events;
} clap_process_t;
```

処理対象フレーム数（ オーディオバッファの長さ）、オーディオ入力と出力、MIDI 等のイベント入力と出力が含まれています。実際にはさらに演奏中の楽曲の詳細情報なども取得する窓口があるのですが、今は省略します（ `clap_event_transport_t`、大まかには VST3 の `ProcessContext` に相当します）。

4.4.8 CLAP オーディオバッファ

オーディオバッファ clap_audio_buffer_t の定義は別のヘッダーファイル include/clap/audio-buffer.h に、イベントバッファ clap_input_events_t と clap_output_events_t も別のヘッダーファイル include/clap/events.h に定義されています。

オーディオバッファ構造体の定義は難しくありません。

```
typedef struct clap_audio_buffer {
   // Either data32 or data64 pointer will be set.
   float  **data32;
   double **data64;
   uint32_t channel_count;
   uint32_t latency;       // latency from/to the audio interface
   uint64_t constant_mask;
} clap_audio_buffer_t;
```

32bit float と 64bit double で別々のフィールドを持たせられるようになっているのがひとつの特徴です。もうひとつ特徴的なのは latency のフィールドをこの構造体のレベルで指定できるところで、つまりオーディオデバイスごとに生じている遅延を反映できるようになっています。

4.4.9 CLAP イベントバッファ

イベントバッファ構造体は少し詳しく説明する必要があります。入力イベントリスト構造体 clap_input_events_t は次のように定義されています。

```
typedef struct clap_input_events {
   void *ctx;
   uint32_t (*size)(const struct clap_input_events *list);
   const clap_event_header_t *(*get)(const struct clap_input_events *list, uint32_t index);
} clap_input_events_t;
```

コンテキスト情報 ctx はこの構造体を生成するホスト側が任意に使います。size() はリストのサイズを返し、get() は個別のメッセージを返します。size が実際に何を示すのか、バイト数なのか入力イベントの「数」なのか、CLAP 仕様は曖昧です（これは仕様に問題がありますが、すでに GitHub Issues にバグ登録してあるので、いずれかに解決されると期待して話を進めましょう）。get() では、指定された index にある個別のイベント構造体の可変長データのポインターが返されます。

出力イベントリスト構造体 clap_output_events_t 構造体はもう少し簡単です。try_push() でイベントリストとイベントを引数に渡すと、リストにイベントを追加できます（ただしメモリ不足で失敗することもあるので、失敗時には false を返すというかたちになります。

```
typedef struct clap_output_events {
   void *ctx;
   bool (*try_push)(const struct clap_output_events *list, const clap_event_header_t *event);
} clap_output_events_t;
```

4.4.10 CLAP イベントデータの基本構造

個別のイベント データの構造について、さらに掘り下げて説明する必要があるでしょう。イベントは、include/clap/events.h に含まれるものだけでも次の 8 種類があります。

- clap_event_note_t
- clap_event_note_expression_t
- clap_event_param_value_t
- clap_event_param_mod_t
- clap_event_param_gesture_t
- clap_event_transport_t
- clap_event_midi_t
- clap_event_midi2_t

これらのイベントは CLAP コアイベントと呼ばれ、CLAP_CORE_EVENT_SPACE_ID という名前空間のような ID で識別されます。独自の"space ID"を定義すれば、プラグインあるいはホスト独自のイベント定義を使うことも可能です。ただし、その構造を解析できるよう、最低限のルールに準拠する必要があります。

これを説明するために、まず CLAP 固有の情報が少ない MIDI イベントの構造体 clap_event_midi_t を例に説明しましょう。

```
typedef struct clap_event_midi {
   clap_event_header_t header;
   uint16_t port_index;
   uint8_t  data[3];
} clap_event_midi_t;
```

3 バイトのメッセージデータが入っていて、システムエクスクルーシブ以外のメッセージを送れる典型的な MIDI イベントです。clap_event_header_t はどのイベント構造体でも使われる重要な部分で、この header フィールドが必ず先頭に入っています。定義内容は次のとおりです。

```
typedef struct clap_event_header {
   uint32_t size;
   uint32_t time;
   uint16_t space_id;
   uint16_t type;     // event type
   uint32_t flags;    // see clap_event_flags
} clap_event_header_t;
```

- size はこのヘッダー自身も含むメッセージのバイト数が含まれます。
- time はそのイベントが処理されるべきタイムスタンプをオーディオ処理におけるサンプル数で指定します。たとえばオーディオサンプルレートが 44100Hz で処理されているとき、441 であればその 1 回の処理サイクルにおける 1/100 秒の位置にそのイベントがあるということ

になります。

- space_id は前述の CLAP_CORE_EVENT_SPACE_ID などが指定されます。標準外のイベントであれば、ここに独自の値を設定します。そのようなイベントが処理されるかどうかはホスト次第、プラグイン次第です。
- type に何を指定するかはイベント種別次第ですが、CLAP コアイベントなら CLAP_EVENT_NOTE_ON などの定義済みシンボルを指定します。

clap_input_events_t の get() で返されるポインターの最初の 4 バイトは常に size の値になるので、このバイト数だけ飛ばせば次のメッセージのポインターを格納できることになります。実際にそのように詰め込まれた構造体になるかは clap_input_events_t の実装次第なので、データが連続していることを期待してはいけないでしょう。しかし、こういう API 構成によって、clap_in_events_t は、そのデータリストの実体を単一のメモリ領域として確保しておけばよく、オーディオ処理中に動的にメモリを確保する必要がなくなります。ちなみに、このようなデータ構造は LV2 のイベントシーケンスを規定する LV2 Atom とほぼ同様です。

clapeventflags

clap_event_header_t の最後のフィールド flags については、もう少し細かい説明が必要でしょう。ここに指定可能な値は clap_event_flags という enum で定義されています。

```
enum clap_event_flags {
   CLAP_EVENT_IS_LIVE = 1 << 0,
   CLAP_EVENT_DONT_RECORD = 1 << 1,
};
```

CLAP_EVENT_IS_LIVE のビットが有効になっていたら、それはユーザーが GUI などで操作した入力値であることを示します。それ以外は楽曲のシーケンスに含まれていたり、オートメーションで自動生成されていたりすることになります。CLAP_EVENT_DONT_RECORD は、データ入力として記録（譜面に入力）する必要のないイベントであることを示します。これもオートメーションが例になりますが、自動生成イベントなどが複数同時に走っていると、それらが記録された時に意図しないコンフリクトが生じてしまうので、それを回避するために用いられます。

4.4.11 CLAP ノートイベント　：オン、オフ、チョーク、終了通知

CLAP が仕様として推奨するイベントは、（MIDI1/MIDI2 ではなく）clap_event_note_t 構造体で表されるもので、具体的には次の 4 つがあります。

- CLAP_EVENT_NOTE_ON
- CLAP_EVENT_NOTE_OFF
- CLAP_EVENT_NOTE_CHOKE
- CLAP_EVENT_NOTE_END

ノートオン、ノートオフは一般的な MIDI 命令と同様ですが、CLAP_EVENT_NOTE_CHOKE と CLAP_EVENT_NOTE_END は説明が必要でしょう。まずチョークは基本的にはノートオフと同じ機能を実現

するものですが、一般的にノート オフはその時点で直ちに消音するのではなくエンベロープにおけるノートリリース処理を開始し、すなわち（ 一般的には）短い時間ながらも段階的に音量を下げていくのに対して、チョークの場合は直ちに消音します。これは特にドラムのクローズド ハイハットとオープンハイハットが排他的に発音する構造になっている場合に有用である、と説明されています。

チョークのもうひとつの利用例として仕様（ コメント ）に記述されているのは、MIDI システムメッセージに含まれる all notes off のような消音の命令です。これも音をリリース無しでただちに遮断したいときに使うものなので、choke が適切でしょう。

もっとも、実際にチョークを楽曲のデータとしてどのように打ち込むべきなのかは、ホストの設計次第であり、この命令を使って（ この命令に反応して）チョークを実装するのが適切なのか、筆者は懐疑的です。VST などですでにドラム系プラグインがハイハットをリアルに再現できているということは、このようなイベント種別が無くても MIDI 打ち込みだけで実現できているということを示しています。MIDI イベントをサポートするプラグインでは、ノート オンとノート オフだけでこれが再現できなければならないはずです。序文で書いた問題意識と繋がる話ですが、MIDI メッセージであれば CLAP から他のプラグインフォーマットに切り換えても一貫した処理を維持できるでしょうし、CLAP イベントに特化した打ち込みになっていると、一貫性が維持できません。CLAP の特長を押し出したいというプラグイン開発者のエゴによって作曲者・ ユーザーが不利益を被ることになってはならない、というのが筆者の持論です。

最後のイベント種別 `CLAP_EVENT_NOTE_END` は、これまた CLAP 特有のイベントで、これはホストがプラグインに指示するものではなく、ノートの発音終了時にプラグインからホストに通知するためのものです。ホストは通常、発声処理中のノートに関する情報を保持していますが、それらのノートリリース処理がいつ完了するかは、オーディオ処理の API 単体では明らかではありません。しかしいつまでもノート処理が続いているとみなすわけにはいかないので、一般的にはリリース処理でどれくらい音が維持されるかをあらわす tail length という数値をプラグインの拡張 API で取得できるようになっています（ CLAP にも tail 拡張機能として定義されています）。CLAP はこれをもう一歩進めて、プラグインからホストにこの `CLAP_EVENT_NOTE_END` イベントを経由して通知できる仕組みにしたというわけです。

4.4.12 単一のイベントポート、複数のイベントポート

`clap_process_t` の構造の特徴として、イベント入力とイベント出力はどちらも単一のイベントストリームとして定義されています。これは複数のイベント ストリームソースを持たせないための、CLAP の意図的な設計のあらわれです。

イベントの処理順序は重要な意味をもちます。ある種のコントロールチェンジ（ たとえばピアノのダンパーペダルの値）を送ってからノート オン／オフを送る場合と、ノート オン／オフを送ってからコントロールチェンジを送る場合では、出力される音が変わってしまいますが、複数のイベントが同じタイムスタンプで並んでいるとき、その結果が一意に定まらないと、一貫した楽曲の打ち込みとして成り立ちません。この問題を解決する一番シンプルな方法が、少なくともホストとプラグインの間のやり取りではイベント キューを 1 つに限定することです。

DAW では楽曲の打ち込みデータとユーザー入力による変更を合わせてプラグインに処理させる必要がありますが、そのマージ処理は DAW が責任を持って処理することになります。

一方で、MIDI デバイスなど複数の入力・出力ポートが存在する状況はあり、それらを 1 つしかサポートできないというのでは不便です。CLAP は実際には複数のイベント入出力のポートをサポートしています。ただし、それは複数の `clap_in_events_t` と `clap_out_events_t` を通じてではなく、イベント構造体のメンバーとしての `port_id` を指定するというかたちになります（イベントの種類によっては存在しません）。同じ「ポート」という単語が、別々のセマンティクスをもって使用されていて、設計としてはあまり良くないところです。

4.4.13 ノートエクスプレッション

ノートエクスプレッション (Note Expression) またはノート別エクスプレッション (Per-Note Expression) とは、プラグインに送信された全てのノートではなく、特定のノートだけに適用されるプラグインのパラメーターを実現する仕組みです。MIDI 2.0 UMP には含まれている命令ですが、MIDI 1.0 の原規格には含まれておらず、MPE (MIDI Polyphonic Expression) という派生規格で別途実現しています。もし MPE が無ければ、一般的に MIDI 2.0 がサポートされていない OS では MIDI 入力デバイスから受け取ることができないでしょう。

CLAP ではイベント種別 `CLAP_EVENT_NOTE_EXPRESSION` に関連付けられた `clap_event_note_expression_t` というイベント構造体があり、また `clap_event_param_value_t` などのパラメーター変更イベントには `key` や `note_id` が指定できるようになっています。`note_id` が指定されていたら、同じ `key` でもパラメーターの適用対象を個別のノートに限定できます。

4.5 オーディオスレッドと main スレッドとそれ以外のスレッド

オーディオプラグインにおいて、スレッドの適切な利用は重要事項です。リアルタイム処理は長時間（全ての処理で 10 ミリ秒程度が限度の目安）かかってはならず、特にメモリ確保など処理時間が不確定な処理は容認されません。一方で GUI 処理は、いわゆる UI スレッド上で処理されなければ、GUI フレームワークによってエラーとなってしまいます。

CLAP では UI スレッドではなく main スレッドと呼ばれていますが、GUI 処理に限らず、ホストとプラグインのインタラクションが必要となる機能の一部は main スレッドで処理されることが仕様上求められています。たとえば `audio_ports` 拡張でオーディオポート情報を取得する処理は main スレッドで実行する必要があります。

プラグインの処理が適切なスレッド上で実行できるように、CLAP にはいくつかの API が定義されています。

- `clap_host_t` の `request_callback()` は、プラグインがホストに「main スレッドで実行したい処理がある」と通知できるようになっています
- `request_callback()` を呼び出されたホストは、呼び出した `clap_plugin_t` の `on_main_thread()` を main スレッド上で呼び出します
- `clap_plugin_t` には `request_process()` という関数があり、プラグイン側が（ホストの操作によって）停止していても、プラグイン側からの事情（I/O など）でホストにオーディオ処理を要請できます
- `thread-check` 拡張には、ホストが実装してプラグインが呼び出せるスレッドチェックの関数

を含む clap_host_thread_check_t 構造体が定義されていて、具体的には is_main_thread()、is_audio_thread() という関数が用意されています

clap-helpers では、CLAP 1.0 仕様で定義された拡張機能を実装したスタブクラスが含まれていますが、その拡張機能を実装するメンバー関数によっては、この動作スレッドのチェックが自動的に行われるようになっています。

第 5 章

CLAP の各種機能

この章では、ここまで説明してこなかった CLAP の拡張機能について、各論的に解説していきます。全てを網羅的に解説するには時間がかかるので、本版では一部の CLAP 拡張機能に限定して説明します。ある程度一般論の CLAP に文脈を限定しない体裁でまとめています。

5.1 状態 (state) の保存・復元

DAW で音楽を打ち込んだ内容をセーブすると、各トラックに設定されたプラグインのパラメーター等をひと通り操作した状態 (state) を保存することになります。逆に楽曲データをロードすると、プラグインのパラメーター等を保存データから復元することになります。

DAW はひとつのプロセスの中にさまざまなプラグインをロードするので、他のアプリケーションに比べると頻繁にクラッシュします。そのため、楽曲をセーブする前に予防的に状態を保存することがあります。

状態の保存先は大抵のプラグインフォーマットではただのバイトストリームですが、LV2 では保存する項目と値を構造的に格納するスタイルの仕様になっています。LV2 は全体的に Semantic Web の流儀に従っており、LV2 State の仕様もその影響を受けていると考えられます。

5.1.1 `clap_plugin_state_t`

CLAP の state プラグイン拡張は `clap_plugin_state_t` で定義されています。ホストの実装用には `clap_host_state_t` があります。

```
typedef struct clap_plugin_state {
   bool (*save)(const clap_plugin_t *plugin, const clap_ostream_t *stream);
   bool (*load)(const clap_plugin_t *plugin, const clap_istream_t *stream);
} clap_plugin_state_t;
```

`clap_ostream_t` と `clap_istream_t` は `clap/stream.h` で定義されています。`save` や `load` を実装するためには、これらの実装もプラグイン開発者が提供することになります。

```
typedef struct clap_istream {
   void *ctx; // reserved pointer for the stream
   int64_t (*read)(const struct clap_istream *stream, void *buffer, uint64_t size);
} clap_istream_t;

typedef struct clap_ostream {
   void *ctx; // reserved pointer for the stream
   int64_t (*write)(const struct clap_ostream *stream, const void *buffer, uint64_t size);
} clap_ostream_t;
```

CLAP ではこのように、C の stdio や C++ の fstream などの特定の API に依存することなくストリームにアクセスするインターフェースを API で利用しています。

5.2 プリセットの利用

プラグインの中には MIDI でいうところの「プログラムチェンジ」に相当する機能を実装しているものがあります。シンセの場合は、これは単純にパラメーターの設定値の集合だけで実現していることも多く、その場合はプリセットと呼ぶほうが適切ともいえます。この機能をどう呼ぶかはプラグインフォーマット次第です。ここでは便宜上プリセットと呼びます。JUCE AudioProcessor なら Program と呼ばれます。

パラメーター設定の集合であると考えるとピンとくるかもしれませんが、プリセットが実装すべき機能は実のところ状態の機能とほぼ重複します。プリセットのロードとは、プログラムナンバー、プリセットの番号といったものを状態の代名として指定して状態を復元するのとほぼ同じです。

プラグインフォーマットによっては、ユーザープリセットの保存のような機能を可能にすることも考えられます（JUCE にはそのような機能が存在します。`AudioProcessor::ChangeProgramName()` など）。

5.2.1 clap_plugin_preset_load_t

CLAP にはプリセットを番号で選択するような機能は用意されていません。「プリセットをファイルからロードする」ための拡張がありますが、まだ最終版ではないドラフト仕様です。

```
typedef struct clap_plugin_preset_load {
   bool (*from_file)(const clap_plugin_t *plugin, const char *path);
} clap_plugin_preset_load_t;
```

この仕様では、そもそもプリセットが何なのかを全く規定していません。各プリセットの実態は state であるといえますが、`clap_plugin_state_t` の API で表出する型の中に state を表す型はなく、`void * buffer` としてのみ渡されています。そして、この `clap_plugin_preset_load_t.from_file()` でロードされたプラグインのプリセット情報は、この API でホスト側に返す必要が無いのです。プラグインがその関数を呼び出されたときにその内部でロードできていればよく、それ以上の操作は CLAP の API としては存在しないということになります。

ロードしたプリセットを実際にプラグインで適用するようにする方法としては、筆者の思いつく限りでは次のやり方が考えられます。

- GUI 上で選択できるようにする: この場合、全てがプラグイン内部で完結するので、ホスト側が関知する必要は何もありません。
- パラメーターの変更によって指定する: この場合、ホストからの操作によってプラグインの状態が変わることになりますが、パラメーターを設定する手段が用意されていればよいので、それ以上固有の API を定義する必要はありません。

プリセットとしてどのような機能が存在すべきかについては、定着した考え方が無いので、現状ではこの `load` のみをもつ API が妥当であるとされているのでしょう。LV2 の場合は、ホストから Atom ポートにファイル名を渡してプラグインに渡すことも可能ですが、リアルタイムで処理できることを前提とするパラメーターとして文字列データを渡すのは原則としては筋が悪いので、リアルタイム性を前提としない API として別途定義している CLAP の仕様には、相応の合理性があります。

5.3 GUI

GUI はオーディオプラグインの重要な機能のひとつですが、「無くても良い」機能でもあります。GUI が無い場合でも、外部のエディターからパラメーターを操作できる「エディットコントローラー」（これは VST3 の用語です）の機能があれば、DAW がプラグインのパラメーター方法をもとに自前でパラメーター操作の UI を用意できるためです。とはいえ、それでもプラグインはユーザーが操作しやすい UI を提供するのが一般的です。

プラグインフォーマットで専用の GUI フレームワークを提供することは多くありません。汎用プラグインフォーマットでは皆無に近いでしょう。プラグインフォーマットで専用の GUI フレームワークを提供するということは、GUI はその仕組みの上に則って構築するということになります。しかし、一般的には DAW が利用する言語・開発環境は決め打ちにできないので、プラグインの GUI はその GUI と密結合できません。GUI フレームワークを提供しないプラグインフォーマットにできることは、せいぜい GUI 操作においてホストとなる DAW とその小窓で出現するプラグインの間で生じる（その必要がある）インタラクションを、呼び出しや通知コールバックのかたちで定義するくらいです。

GUI フレームワークを開発するというのは大規模な作業になりうるもので、実際大規模な作業を経ずに基本機能だけで済ませた GUI フレームワークではさまざまな問題が噴出します。日本人向けにわかりやすい例を挙げれば、日本語入力にまともに対応できないことが多いです。アクセシビリティ対応、HiDPI 対応、マルチプラットフォーム対応など、さまざまな難題があるのです。Steinberg は VSTGUI というオーディオプラグイン向けの汎用フレームワーク（これは VST 専用ではありません）を作りましたが、やはりデスクトップ向けの一般的な GUI フレームワークと比べたらさまざまな点で妥協の産物です（たとえば 2022 年現在でも Cairo が使われていたりします）。

プラグイン UI 開発に最適な銀の弾丸は存在せず、プラグイン開発者は、自分のプラグインの最適解に近い任意の GUI フレームワークを利用する、という以上の一般化はできないといえます。

オーディオプラグインのオーディオ処理はリアルタイムで完了する必要があります。このリアルタイムとは「必ず一定の時間以内に完了する」というものであり、よく hard realtime ともいわれるものです。一方で GUI 処理には一般的に「UI スレッドで動作しなければならない」という制約があります。必然的に、オーディオ処理と GUI 処理は別々のスレッドで動作することになります。

さて、一般的にプラグインのオーディオ処理と GUI は別々のスレッドで別々の処理ループによって動作することになりますが、プラグインフォーマットによっては GUI の分離がスレッドの分離より一層強く設計されていることがあります。LV2 はこの意味では分離アプローチの最右翼で、UI のためのライブラリを別ファイル上で実装して、オーディオ処理部分とはコードを共有できないようにしています。オーディオ処理のコードを参照しなくても、TTL メタデータの情報をもとに UI を実装することが可能であるためです。もちろんそうはいっても、UI のライブラリを参照してその API を利用するコードを書くのを妨げることはできません。

GUI サポートをクロスプラットフォームで一般化するのは、可能ではありますが、技術的にいくつかのアプローチがあり、これがまた一つ難しい要因です。プラグインフォーマットとして何か 1 つを規定しないわけにはいきません。

- VST3 ではプラグインの `IEditController::createView()` から `IPlugView` というインターフェースの実体としてプラットフォーム別の View を生成して、それをホストに返します。ホストは GUI の View を（一般的には）自前のウィンドウに reparent して使うことになります。
- CLAP ではホストが `clap_plugin_gui_t.create()` を呼び出すとプラグインが内部的に GUI を生成しますが、結果は bool でしか帰ってきません。それをホスト側の GUI に統合するには、reparent するウィンドウのハンドルを `clap_plugin_gui_t.set_parent()` で渡す必要があります。あるいは floating window として扱うという選択肢もありますが、プラグインがサポートしていなければこれは利用できません。clap-juce-extensions で（つまり JUCE で）構築したプラグインだと floating には対応していません。

CLAP では、LV2 のような UI と DSP のコード分離ポリシーを API として強制してはいません。これは意図的な設計であるとコミュニティでは説明されています。コードをどのように分離するかは各アプリケーションのアーキテクチャ次第ともいえます。

5.3.1 `clap_plugin_gui_t` と `clap_host_gui_t`

GUI サポートのために必要な拡張の API はそれなりに大きなものです。`clap_plugin_gui_t` には 15 件の関数がありますが、これでもだいぶ API としては薄いほうというべきでしょう。API 定義は次のようになっています。

```
typedef struct clap_plugin_gui {
   bool (*is_api_supported)(const clap_plugin_t *plugin, const char *api, bool is_floating);
   bool (*get_preferred_api)(const clap_plugin_t *plugin, const char **api, bool *is_floating);
   bool (*create)(const clap_plugin_t *plugin, const char *api, bool is_floating);
   void (*destroy)(const clap_plugin_t *plugin);
   bool (*set_scale)(const clap_plugin_t *plugin, double scale);
   bool (*get_size)(const clap_plugin_t *plugin, uint32_t *width, uint32_t *height);
   bool (*can_resize)(const clap_plugin_t *plugin);
   bool (*get_resize_hints)(const clap_plugin_t *plugin, clap_gui_resize_hints_t *hints);
   bool (*adjust_size)(const clap_plugin_t *plugin, uint32_t *width, uint32_t *height);
   bool (*set_size)(const clap_plugin_t *plugin, uint32_t width, uint32_t height);
   bool (*set_parent)(const clap_plugin_t *plugin, const clap_window_t *window);
   bool (*set_transient)(const clap_plugin_t *plugin, const clap_window_t *window);
```

```
   void (*suggest_title)(const clap_plugin_t *plugin, const char *title);
   bool (*show)(const clap_plugin_t *plugin);
   bool (*hide)(const clap_plugin_t *plugin);
} clap_plugin_gui_t;
```

ホストとプラグインのインタラクションのために、プラグインウィンドウの生成・表示・非表示・破棄、サイズ変更やサイズ変更可否情報の取得、スケール（縮尺）の変更などの実装が求められます。

そしてこれらを実装する際には、`clap_plugin_host_t` のインスタンスを（プラグインの `clap_plugin_factory_t.create_plugin()` の引数として渡された）`clap_host_t` から取得して、そのメンバー関数を呼び出す必要が、少なからずあります。

```
typedef struct clap_host_gui {
   void (*resize_hints_changed)(const clap_host_t *host);
   bool (*request_resize)(const clap_host_t *host, uint32_t width, uint32_t height);
   bool (*request_show)(const clap_host_t *host);
   bool (*request_hide)(const clap_host_t *host);
   void (*closed)(const clap_host_t *host, bool was_destroyed);
} clap_host_gui_t;
```

たとえばプラグインウィンドウを閉じたり非表示にしたりする操作がホストのウィンドウ表示 API ではなくプラグイン GUI の画面要素（たとえば「閉じる」ボタン）から行われた場合は、プラグインの実装から `clap_host_t` の拡張として `clap_host_gui_t` を取得し、`request_hide` や `closed` などのメンバーを呼び出す必要があります。そうしないとホストとプラグインの間で一貫した GUI 表示状態を保てないためです。

いずれにせよ、これだけのメンバーを自分で全て正しく実装するのは、やや骨の折れる作業です。既存のプラグイン開発用 SDK を使う開発スタイルは、自分でこの辺りの面倒を見る必要がなくなるのが魅力のひとつです。

5.4 ホストから提供される「楽曲の」情報

オーディオプラグインは基本的にオーディオ処理関数（CLAP の `process()` 関数など）に渡されるオーディオ入力やイベント入力をもとにオーディオ・イベント出力を出力するリアルタイムな処理であり、渡される時間情報は基本的に SMTPE に基づく時間（マイクロ秒など）の即値あるいはそれを変換したサンプル数となります。そこにテンポや拍子（time signature）に関する情報は一般的には不要ですが、プラグインによっては、テンポ等の値をもとに生成する音声や MIDI イベントを調整したいことがありえます。これを実現するためには、DAW からの情報提供機能が不可欠です。この情報はトランスポートとかプレイバックと呼ばれることがあります。各プラグインフォーマットでは、それぞれ次に示す型で実現しています。

- VST3: `ProcessContext`
- LV2: Time 拡張機能
- CLAP: `clap_event_transport` (`events.h`)

CLAP の `clap_event_transport` は（拡張ではなく）オーディオ処理で渡されるイベントの種類

で、トランスポート情報にアップデートがあったときにホストから渡されます。現在の小節位置なども含まれる = 更新の必要が頻繁に生じるので、このイベントをサポートする DAW からは process() で送られる clap_process_t の in_events に含まれることが多いと考えて良いでしょう。

CLAP には track-info というトラック情報を取得できる API もありますが、これは DAW 上の表示色など、だいぶ性質の異なる情報を取得するためのものです。

5.5 パラメーター設定関連イベント

CLAP ではプラグインパラメーターの操作も拡張機能として定義されています。少しメンバーが多いので、一部を省略して記します。

```
typedef struct clap_plugin_params {
   uint32_t (*count)(const clap_plugin_t *plugin);
   bool (*get_info)(const clap_plugin_t *plugin,
                    uint32_t             param_index,
                    clap_param_info_t   *param_info);
   bool (*get_value)(const clap_plugin_t *plugin, clap_id param_id, double *value);
   ...
   void (*flush)(const clap_plugin_t        *plugin,
                 const clap_input_events_t  *in,
                 const clap_output_events_t *out);
} clap_plugin_params_t;
```

パラメーターの情報をホストが取得する get_info()、値を取得する get_value() は、clap_plugin_params_t に含まれています。一方で、値を設定するためのメンバーはありません。値の設定は clap_plugin_t.process() によって行われることになります。

CLAP のパラメーター設定イベントもある程度バリエーションがあります(イベントについては前章で基本部分を説明しました)。

- CLAP_EVENT_PARAM_VALUE: 単純なパラメーターの設定
- CLAP_EVENT_PARAM_MOD: パラメーターのモジュレーション操作(変化率を指定) : 開発チームが "non-destructive automation" と呼んでいるもので、モジュレーションが完了したらパラメーターの値を元に戻せる(オートメーションをかけ終わった後に当初のパラメーター設定がなくならない) ことになります
- CLAP_EVENT_PARAM_GESTURE_BEGIN, CLAP_EVENT_PARAM_GESTURE_END: ユーザーが DAW 上のツマミなどでパラメーター操作を開始したことをプラグインに通知するイベント : この間に呼び出されたパラメーター変更イベントは履歴の記録などで厳密にトラッキングする必要がない、と考えられます

モジュレーションとジェスチャーは、表現力を高めるためのものではなく、DAW を利用するときの UX を改善するためのものといえます。(他の規格にも同様の機能を実現するものがあるかもしれません。)

また、パラメーターではありませんが、ノートエクスプレッションも CLAP_EVENT_NOTE_EXPRESSION で設定できます。対象パラメーターの代わりに以下のいずれかを「 エクスプレッション ID」として指定します:

```
enum {
  // with 0 < x <= 4, plain = 20 * log(x)
  CLAP_NOTE_EXPRESSION_VOLUME,
  // pan, 0 left, 0.5 center, 1 right
  CLAP_NOTE_EXPRESSION_PAN,
  // relative tuning in semitone, from -120 to +120
  CLAP_NOTE_EXPRESSION_TUNING,
  // 0..1
  CLAP_NOTE_EXPRESSION_VIBRATO,
  CLAP_NOTE_EXPRESSION_EXPRESSION,
  CLAP_NOTE_EXPRESSION_BRIGHTNESS,
  CLAP_NOTE_EXPRESSION_PRESSURE,
};
```

5.6 ボイス(発音)数の管理　(voice-info) と tail 情報

CLAP にはプラグインの発音数をホスト側で取得できる voice-info という拡張機能があります。

```
enum {
  CLAP_VOICE_INFO_SUPPORTS_OVERLAPPING_NOTES = 1 << 0,
};
typedef struct clap_voice_info {
  uint32_t voice_count;
  uint32_t voice_capacity;
  uint64_t flags;
} clap_voice_info_t;
typedef struct clap_plugin_voice_info {
  bool (*get)(const clap_plugin_t *plugin, clap_voice_info_t *info);
} clap_plugin_voice_info_t;
```

これが使えると、ホストでプラグインが現在利用可能な発音総数 (voice_count) や最大発音数 (voice_capacity) を取得できます。これ単独で音声処理に影響があるとはいえません。CLAP 仕様の他の機能と合わせて使わない限り、雰囲気でパフォーマンスのある種の指標を得る程度の用途しかないでしょう(筆者も本書の初版執筆時点ではそう理解していました)。これは非常に紛らわしいですが、voice_count は現在発音中のボイス数ではありません。
(本書の初版執筆時点ではドラフト仕様でしたが、2022 年 10 月の本版執筆時点では正式版になりました。)

5.6.1 ノートとボイスの違い

「ノート」についての捉え方も、「ボイス」ついての捉え方も、複数の解釈がありえます。理解に齟齬が生じないようにここで確認しておきましょう。

ノートにはノートナンバーがあり、特に伝統的な MIDI インストゥルメントでノートの状態を管理する(どんなノートがオンになっているかを把握する) 目的で、このノートナンバーが使われていました。これには「1 つのノートナンバーについて、あるノートオン中に別のノートオンが発生することはない」という前提があります。この前提は、MPE に代表される「ノート別エクスプレッション」の時代になって特に大きく崩れました。あるノートでノートオンの後でドラッグしながら、同じノートでノートオンの後で別の方向にドラッグするような操作が可能になりました。この同じノートナン

バーで発声している 2 つのものを「 ノート 」と特に区別せずに呼ぶこともあれば、別の概念として「 ボイス」と呼ぶこともあります。

（ これは本当は MPE 以降の MIDI デバイスに限らず、その前から存在していたはずの問題ではあります。ASDR の無い固定のサイン波などでもない限り、異なるタイミングで生じたノート オンに対応する波形は、特定の時間領域においては以前のノート オンのものとは異なるはずです。複数並行して発音していても問題ないはずなのです。）

一方で、1 つのノート メッセージに対して複数のユニゾンなどを指定できるシンセサイザーでは 1 つの「 ノート 」(あるいは上記「 ボイス」) につき、4「 ボイス」、6「 ボイス」、8「 ボイス」…と発声するのが一般的です。1 つのノート ナンバーに割り当てられている 1 つ以上発生しうる鍵盤押下に対応するものを「 ボイス」と呼ぶのであれば、このユニゾンによって生じる「 ボイス」と混乱することになります。いずれにしろ、これらの「 ノート 」や「 ボイス」の意味を、文脈に応じて適切に把握する必要があります。

CLAP の voice-info 拡張が想定している挙動は「 1 ノート オン処理につき 1 ボイス」です。CLAP における「 ノート 」と「 ボイス」の使い分けは、同一ナンバーのノートを複数処理する（ 1 つのノート ナンバーについて、あるノート オン中に別のノート オンが発生することがある）という考え方になっています。1 つのノート オンに対して 4 ボイス、6 ボイス…と生じる「 ボイス」の意味もありません。

5.6.2　tail: 一般的な発音状態の報告手法

一般的なホストにとって重要なのは、プラグインが内部的に保持している「 発音数」より、プラグインが「 発音状態にあるかどうか」でしょう。発音状態にないプラグインについては、ホストはそのプラグインへの音声入力がゼロでノートの無い状態であれば、「 このプラグインのオーディオ処理を呼び出しても何も返ってこない」と判断して処理を止める最適化を施せる可能性があります。ただし、オーディオ入力やノートが 0 の状態であっても、ディレイやリバーブなどが残っていることはあるので、直ちにオーディオ処理を止めてしまうと、あるべき残響音のオーディオ出力が欠落してしまうことになります。

これを防ぐためにあるのが「 完全な消音までどれくらいの時間が必要かプラグインから取得してその分だけ消音モードへの切り替えを待つ」tail 処理であり、この時間情報をホストが取得するためのプラグイン拡張機能が用意されているのが一般的です。CLAP にも tail という拡張があります。

```
typedef struct clap_plugin_tail {
   uint32_t (*get)(const clap_plugin_t *plugin);
} clap_plugin_tail_t;

typedef struct clap_host_tail {
   void (*changed)(const clap_host_t *host);
} clap_host_tail_t;
```

リバーブのパラメーター値が変更される等によってこの tail の値が変わった場合は、`clap_host_tail_t` の `changed()` を呼び出してホスト側に変更を通知する必要があります。

5.6.3 ホストが管理するボイス情報 ？

さて、ここでひとつ読者に思い出してもらいたいのが、前章で説明した CLAP ノート終了イベント CLAP_EVENT_NOTE_END です。これは、ノートが完全に終了してこれ以上音声を生成しない状態になったときに、プラグインからホスト側に通知されます。このイベントがあれば、ノートの状態を問わずざっくりした数値のみを表す tail length の情報は必要ないともいえます。とはいえ、このイベントは CLAP にしか無く、一方で tail length のプロパティはさまざまなプラグインフォーマットで実現している機能なので、複数のプラグインフォーマットをサポートする一般的なホストでは tail のみがサポートされている可能性がそれなりに高く、最大公約数を考えれば tail length が実装されているべきです。

一方で、消音制御とは無関係に、voice-info 拡張に基づいてボイス情報の取得が可能であれば、プラグイン側がボイス処理のキャパシティオーバーを把握することができて、ホストからプラグインのキャパシティを超えるボイス処理を生じさせないためにノート処理を控えるような制御が可能になるかもしれません。

CLAP ノート終了イベント CLAP_EVENT_NOTE_END の仕組みが想定する「ホストによるボイス管理」は、ホスト側でノートイベントの状態をプラグインと同期することを想定しており、CLAP 仕様では voice-info 拡張の C ヘッダーに次のような説明が記されています（バージョン 1.1）。

```
// It is useful for the host when performing polyphonic modulations,
// because the host needs its own voice management and should try to follow
// what the plugin is doing:
// - make the host's voice pool coherent with what the plugin has
// - turn the host's voice management to mono when the plugin is mono
```

このような記述がなければ、極端な話、ホスト側はノートオンに対応するノートオフさえ送れるなら、ノートイベントを fire and forget 方式で「送って、そのことを忘れて」いてもよいわけです。これは一般的なオーディオプラグインの挙動といえます。ホストはプラグインが最大処理可能ボイス数を気にすること無くノートオンとノートオフのペアを送ることについてのみ責任を負い、ノートオフに対応するノートオンがあったかどうかを気にするのはプラグイン側の責務とするのは、ホストとプラグインの間で状態管理を複雑化させないために非常に合理的です（CLAP の仕様のほうが不合理です）。

そもそも、ボイス数が多すぎてプラグイン側が発音しきれないと判断した場合、どの音を取捨選択するかはプラグインが内部的に判断する事項です。FIFO になっているかもしれませんし、音量が最小のものなど、影響が最小のものを消音するかもしれません。ホストが判断すべき事項とは限らないのです。

CLAP 仕様コメントでも、引用した通り "should"としか書いておらず（そもそもプラグインとホストのいずれかが voice-info に対応していない可能性は十分に高いでしょう）、ホストでボイス管理できることによるメリットが具体的に発生しないうちは、特別に対応する価値は高くない、というのが、筆者のこの仕様についての理解です。

5.7　リアルタイム並列処理の制御　(thread_pool 拡張)

u-he で頻繁に主張している CLAP のアドバンテージのひとつが「ホストによって制御されるスレッドプール」です。これについて筆者は「スレッドプールは LV2 Worker などでも実装されているし、さすがにそれはおかしいんじゃないか」と思ってだいぶコミュニティで掘り下げて議論して分かったのですが、結論からいえば彼らの主張は間違ってはいません。というのは、CLAP でいうところのスレッドプールとは一般的なアプリケーション開発におけるスレッドプールでは全くないためです。すなわち、これはプラグインが非同期実行を実現するための仕組みではありません。

では何なのかというと、CLAP の `thread_pool` 拡張の API は、リアルタイム処理を並列で実行するための **API** です。プラグインがオーディオスレッドで動作している `process()` の中からホストの機能を呼び出すかたちで利用します。次のような流れになります:

- プラグインが `clap_host_thread_pool_t` 型のホスト拡張を `clap_host_t.get_extension()` で取得し、これが `nullptr` なら並列処理ではなく逐次処理を行う
- `clap_host_thread_pool_t` を取得できたら、プラグインは続けて `request_exec(host, numTasks)` メンバーを呼び出す
- ホストの `request_exec(host, numTasks)` の実装では、`numTasks` で指定された本数のタスクを OpenMP などの並列実行機構を用いて並列化できるか検討して
  - できないようなら、それ以上は何も実行せずに `false` を返す
  - 並列化できるようなら、そのプラグインの `clap_plugin_thread_pool_t` 型の拡張機能を `clap_plugin_t.get_extension()` で取得する。これが `nullptr` なら `false` を返す
  - `clap_plugin_thread_pool_t` を取得できたら、ホストは続けてその `exec(plugin, task_index)` を `numTasks` で指定された回数だけ呼び出し、`request_exec()` の戻り値として `true` を返す
  - `request_exec()` を呼び出したプラグインの `process()` では、もし `request_exec()` の結果によって続く処理が変わることになる
  - true であれば、並列処理が成功し処理が完了した状態でホストから返ってきたのでそれ以降の処理を実行すればよいということになる
  - false であれば `clap_plugin_thread_pool_t` で行いたかった処理は何一つ始まっていないので、改めて、並列処理を行わずにプラグインの処理を続行する必要がある

`exec()` で呼び出されるプラグインのタスクは、`process()` のサイクルで完了しなければならないものなので、並列であれ逐次であれ、処理全体をリアルタイムで実行完了しなければなりません。

CLAP の `thread_pool` 拡張とは、こういった機能を実現するためのものです。一般的な意味でのスレッドプールの API はありません。一般的なスレッドプールの API であれば、タスク／ジョブのオブジェクトを生成してハンドルを渡すような API になっていないと意味を為さないところですが、CLAP の場合は `numTasks` という並列実行スロットの本数を渡すのみで、プラグイン側のタスクの呼び出しも同期的です。「thread pool という名前がおかしい」というのは概ねコミュニティにおける共通理解だと思ってよさそうです。

5.8 tuning

tuning は microtonal（ 微分音）を実現するための拡張機能です。この機能がオーディオプラグインフォーマットの一部として規定されるのは珍しいといえるでしょう。一般的に、これが拡張機能として規定されないのは、MIDI 1.0 に基づく MMA の仕様として MTS (MIDI Tuning Standards）というものがあって、DAW はこれに沿って MIDI メッセージを送信し、プラグインはこれを受け取ったらその内容に応じた周波数変換テーブルを適用すれば良いので、独自にイベントを規定する必要がなかったためです。

CLAP の場合、MIDI イベントではなく CLAP イベントで全てを処理するユースケースに対応することを考えると、MTS に相当するメッセージを規定する必要があるといえるでしょう。`tuning.h` には `clap_event_tuning` という MTS 相当のイベントが規定されています。

CLAP オーディオプラグイン開発者ガイド

2022 年 9 月 11 日　技術書典 13 版 v1.0.0
2022 年 10 月 30 日　M3-2022 秋版 v1.1.0

著　者　atsushieno
編　集　atsushieno
発行所　オーディオプラグイン研究所