The Cmajor Patch Format

Cmajor patches are a format for bundling together code and other resources, so that they can be loaded into audio hosts such as DAWs, to provide instruments or effect plugin functionality.

A Cmajor “patch” is essentially a bundle of files, including:

- A main “manifest” file, with the suffix `.cmajorpatch`. This file describes the patch’s properties and contains links to the other files in the patch
- One or more `.cmajor` source files containing the actual Cmajor code
- Some optional javascript control code for non-DSP housekeeping tasks
- Optionally some sub-folders containing resources such as HTML/CSS/Javascript GUI, audio files, etc.

The `.cmajorpatch` manifest file

The `.cmajorpatch` file contains JSON to describe the properties of the patch.

For example, the `HelloWorld.cmajorpatch` contains:

```
{
    "CmajorVersion":    1,
    "ID":               "dev.cmajor.examples.helloworld",
    "version":          "1.0",
    "name":             "Hello World",
    "description":      "The classic audio Hello World",
    "manufacturer":     "Cmajor Software Ltd",
    "category":         "generator",
    "isInstrument":     false,
```

```
    "source":           "HelloWorld.cmajor"
}
```

There are a few required properties that a patch must define:

- `CmajorVersion` - this is the version of Cmajor for which this patch was written
- `ID` - a universally unique ID for the patch, which should be in the form of a reverse-URL that includes the company name/website.
- `version` - a version number for your patch. This is just a string - there are no restrictions on its format.
- `name` - a human-readable name for your patch

Other optional properties include:

- `description` - a longer description that a host can display to its users
- `manufacturer` - the name of you or your company
- `category` - hosts will be given this string, but how they choose to interpret it will be host-dependent
- `isInstrument` - if specified, this marks the patch as being an instrument rather an effect. Some hosts may treat a plugin differently depending on this flag.

Cmajor source files

The `source` property in the manifest tells the host which `.cmajor` files to compile for the Cmajor source code. This property can either be a single string containing a file path (relative to the folder containing the patch), or an array of string filenames if there are multiple files.

e.g.

```
    "source":   [ "src/MainProcessor.cmajor",
                  "src/Utilities.cmajor" ],
```

All the source files will be loaded and linked together as a single unit, so they can freely refer to definitions in the other files without needing to explicitly import them.

Selection of the patch's main processor

One of the processors defined in your source files will be used as the patch's top-level processor. To help the host decide which one to use, you should decorate it with the `[[ main ]]` attribute, e.g.

```
processor HelloWorld  [[ main ]]
{
    ...
```

Patch Parameters

Patches are designed to be used as audio plugins, and an audio plugin generally has a list of "parameters" that the host can read and write to, so that a host/DAW can record and play back automated changes to these parameters.

The list of parameters for a patch is determined by looking at the input endpoints of its main processor. If the endpoint meets the following criteria, it will be revealed to the host as a parameter:

- It must be a `value` or `event` endpoint. Streams cannot be used as parameters.
- Its type must have a single type which is a `float32`, `float64`, `int32`, `int64` or `bool`.
- It must have an annotation that declares a `name` (and will probably also declare some range properties).

```
processor HelloWorld  [[ main ]]
{
    input stream float audioIn;  // Parameters cannot be streams, so this is ignored

    // This parameter is a float called "foo" and has the range 0.5 to 10
    input event float in1   [[ name: "foo", min: 0.5, max: 10.0 ]];

    // This parameter is an int called "foo2" between 1 and 99
    input value int32 in2   [[ name: "foo2", min: 1, max: 99 ]];
```

The following properties can be added to the endpoint annotation in order to give the host more information about the parameter:

- `name` (required) - the name to display to the user
- `min` - the minimum value for the parameter. If not specified, defaults to 0.0

- `max` - the maximum value for the parameter. If not specified, defaults to 1.0
- `mid` - the middle value in the parameter range. This is used by the standard rotary control to support non-linear scaling of the control if specified
- `init` - if specified, this value will be sent to the parameter when the patch is initialised
- `step` - the intervals to which the parameter value must “snap” when being changed
- `unit` - an optional string which the host will display as the units for this value. E.g. “%” or “dB”
- `boolean` - if this flag is set, the paremeter may be displayed as a toggle switch
- `hidden` - if this flag is set, the parameter won’t be displayed on user interfaces
- `automatable` - if this property is set to true or false, it will be passed to a host where possible, and if the host understands it, it may use it to decide whether to allow automation recording for this parameter
- `rampFrames` - this property can be set to an integer to indicate how many frames it should take to ramp to a new value
- `group` - an optional string to hint at a parent group to which this parameter belongs. Some hosts may use this to organise parameters into on-screen groups.
- `text` - a formatting string which is used to print the value in a custom style. The string is preprocessed a bit like “printf”, supporting the following format strings:
  - `%d` prints the parameter value as an integer
  - `%f` prints the parameter value as a floating point number
  - `%[digits]f` prints the parameter value with a maximum number of decimal places, e.g. `%2f` uses up to 2 decimal places
  - `%0[digits]f` prints the parameter value with exactly the specified number of decimal places
  - `%+d` or `%+f` prints the number with a `+` or `-` sign before it

  The `text` property can also contain a list of names separated by the pipe `|` character. In this mode, the names will be used as labels for the parameter values, and the host may choose to display them to the user in a drop-down menu or other list selector. For example `text: "low|med|high"` will map the strings “low”, “med” and high to the value range 0, 1, 2. If a `min` and `max` range is specified then the values will be spread across the range, e.g. `min: 0, max: 9` would map “low” to 0 → 3, “med” to 3 → 6, and “high” to 6 → 9. The `step` property is ignored and values are snapped automatically based on the number of items.

- `discrete` - if this flag is set, in addition to the `step` property, the parameter will behave similarly to the `text` property above, e.g. a discrete list of options

External variable data

Cmajor code can declare `external` variables whose values are supplied by the runtime environment when the code is loaded. In a patch, you should add entries in the manifest file to supply the data or the resource file that should be loaded into these variables.

e.g. In the “Piano” example, `Piano.cmajor` declares an external variable which is an array of 5 `PianoSample` objects:

```
namespace piano
{
    struct PianoSample
    {
        std::audio_data::Mono source;
        int rootNote;
    }

    external PianoSample[5] samples;
```

The `Piano.cmajorpatch` has an “externals” property which gives the runtime a JSON value that should be loaded into the `piano::samples` variable:

```
"externals": {
    "piano::samples": [ { "source": "piano_36.ogg",  "rootNote": 36 },
                        { "source": "piano_48.ogg",  "rootNote": 48 },
                        { "source": "piano_60.ogg",  "rootNote": 60 },
                        { "source": "piano_72.ogg",  "rootNote": 72 },
                        { "source": "piano_84.ogg",  "rootNote": 84 } ]
}
```

The runtime will attempt to coerce these JSON values to fit the data type of the target variable.

So in the example above, the `samples` variable has the type `PianoSample[5]`. The JSON value is an array containing 5 objects. So it then attempts to convert each of these objects into a `PianoSample` value.

Simple types like integers, floats, bools and strings are converted as you’d expect, and when objects are provided, the JSON objects should have members with the same names as the

Cmajor struct members.

The runtime also supports loading audio data from files, as is done in this example. When attempting to convert a string to some other kind of target object, the runtime will check whether the string is actually the name of an audio resource file in the patch, and if so will load it. It'll then attempt to copy the audio frames and sample rate into the target struct type.

In the example above, the audio files are being loaded into variables whose type is `std::audio_data::Mono`. This is a struct from the Cmajor standard library, which looks like this:

```
struct Mono
{
    float[] frames;
    float64 sampleRate;
}
```

Because it has a member called "sampleRate" the runtime will put the audio file's sample rate into this value. And because it has an array of floats (and the source file is also mono), the audio data will be loaded into this array.

The standard library provides a set of helper structs like this that you can use to load data from audio files, but they're not special - the runtime simply uses duck-typing to decide whether a struct might be able to contain audio data, so you can use your own types too.

It can also load audio file data directly into a raw array, so this would also compile:

```
processor MyProcessor
{
    external float[] audioData;
```

and

```
"externals": {
    "MyProcessor::audioData" : "piano_36.ogg",
}
```

..but since there's nowhere to put the file's sample rate, that information will be discarded.

Patch GUIs

Specifying a custom GUI for a patch

To add a custom GUI to your patch, your `.cmajorpatch` file must declare a `view` property, e.g.

```
{
    "CmajorVersion":    1,
    "ID":               "dev.cmajor.examples.helloworld",
    "version":          "1.0",
    "name":             "Hello World",
    "source":           "HelloWorld.cmajor",

    "view": {
      "src":       "patch_gui/index.js",
      "width":     800,
      "height":    700,
      "resizable": false
    }
}
```

The `view` property should contain a `src` property providing a relative URL to a javascript module.

We expect a patch to provide a web component for its GUI, as these are the basis of all modern browser DOMs, and can be easily embedded into a hosts's own user-interface.

To provide a web component, your javascript module's default export must be a function that returns a `HTMLElement` object. When the host wants to display your patch's GUI, it'll call this function and add the element that it returns to its own window.

The function you provide will be given a single argument when called. The argument provides a `PatchConnection` object that your view should use to communicate with the patch instance that it represents.

For example:

```
class MyAmazingPatchView extends HTMLElement
{
    // ...etc..
```

```
}

export default function createPatchView (patchConnection)
{
    return new MyAmazingPatchView (patchConnection);
}
```

RESIZABLE AND AUTO-SCALING GUI

The `.cmajorpatch` file's `view.resizable` property controls whether the patch window can be resized.

If you want your GUI to automatically scale when the window is resized, your web component should include a `getScaleFactorLimits()` method. This method should return an object specifying the minimum and maximum scale factors for your GUI.

```
class MyAmazingPatchView extends HTMLElement {
    // constructor and other methods...

    getScaleFactorLimits()
    {
        return { minScale: 0.5, maxScale: 1.25 };
    }
}
```

The `PatchConnection` object

The `PatchConnection` object is provided by the host and your `HTMLElement` class uses it to control and communicate with the running patch.

It provides a range of methods for controlling and querying the state of the patch:

STATUS-HANDLING METHODS

- `requestStatusUpdate()` Calling this will trigger an asynchronous callback to any status listeners with the patch's current state. Use `addStatusListener()` to attach a listener to receive it.
- `addStatusListener (listener)` Attaches a listener function that will be called whenever the patch's status changes. The function will be called with a parameter object containing

many properties describing the status, including whether the patch is loaded, any errors, endpoint descriptions, its manifest, etc.

- `removeStatusListener (listener)` Removes a listener that was previously added with `addStatusListener()`
- `resetToInitialState()` Causes the patch to be reset to its “just loaded” state.

METHODS FOR SENDING DATA TO INPUT ENDPOINTS

- `sendEventOrValue (endpointID, value, rampFrames)` Sends a value to one of the patch’s input endpoints. This can be used to send a value to either an ‘event’ or ‘value’ type input endpoint. If the endpoint is a ‘value’ type, then the rampFrames parameter can optionally be used to specify the number of frames over which the current value should ramp to the new target one. The value parameter will be coerced to the type that is expected by the endpoint. So for examples, numbers will be converted to float or integer types, javascript objects and arrays will be converted into more complex types in as good a fashion is possible.
- `sendMIDIInputEvent (endpointID, shortMIDICode)` Sends a short MIDI message value to a MIDI endpoint. The value must be a number encoded with `(byte0 << 16) | (byte1 << 8) | byte2`.
- `sendParameterGestureStart (endpointID)` Tells the patch that a series of changes that constitute a gesture is about to take place for the given endpoint. Remember to call `sendParameterGestureEnd()` after they’re done!
- `sendParameterGestureEnd (endpointID)` Tells the patch that a gesture started by `sendParameterGestureStart()` has finished.

STORED STATE CONTROL METHODS

- `requestStoredStateValue (key)` Requests a callback to any stored-state value listeners with the current value of a given key-value pair. To attach a listener to receive these events, use `addStoredStateValueListener()`.
- `sendStoredStateValue (key, newValue)` Modifies a key-value pair in the patch’s stored state.
- `addStoredStateValueListener (listener)` Attaches a listener function that will be called when any key-value pair in the stored state is changed. The listener function will receive a message parameter with properties `key` and `value`.
- `removeStoredStateValueListener (listener)` Removes a listener that was previously added with `addStoredStateValueListener()`.

- `sendFullStoredState (fullState)` Applies a complete stored state to the patch. To get the current complete state, use `requestFullStoredState()`.

- `requestFullStoredState (callback)` Asynchronously requests the full stored state of the patch. The listener function that is supplied will be called asynchronously with the state as its argument.

LISTENER METHODS

- `addEndpointListener (endpointID, listener, granularity)` Attaches a listener function that will receive updates with the events or audio data that is being sent or received by an endpoint. If the endpoint is an event or value, the callback will be given an argument which is the new value. If the endpoint has the right shape to be treated as “audio” then the callback will receive a stream of updates of the min/max range of chunks of data that is flowing through it. There will be one callback per chunk of data, and the size of chunks is specified by the optional granularity parameter. If sendFullAudioData is false, the listener will receive an argument object containing two properties ‘min’ and ‘max’, which are each an array of values, one element per audio channel. This allows you to find the highest and lowest samples in that chunk for each channel. If sendFullAudioData is true, the listener’s argument will have a property ‘data’ which is an array containing one array per channel of raw audio samples data.

- `removeEndpointListener (endpointID, listener)` Removes a listener that was previously added with `addEndpointListener()`

- `requestParameterValue (endpointID)` This will trigger an asynchronous callback to any parameter listeners that are attached, providing them with its up-to-date current value for the given endpoint. Use `addAllParameterListener()` to attach a listener to receive the result.

- `addParameterListener (endpointID, listener)` Attaches a listener function which will be called whenever the value of a specific parameter changes. The listener function will be called with an argument which is the new value.

- `removeParameterListener (endpointID, listener)` Removes a listener that was previously added with `addParameterListener()`

- `addAllParameterListener (listener)` Attaches a listener function which will be called whenever the value of any parameter changes in the patch. The listener function will be called with an argument object with the fields `endpointID` and `value`.

- `removeAllParameterListener (listener)` Removes a listener that was previously added with `addAllParameterListener()`

ASSET HANDLING METHODS

- `getResourceAddress (path)` This takes a relative path to an asset within the patch bundle, and converts it to a path relative to the root of the browser that is showing the view. You need you use this in your view code to translate your asset URLs to a form that can be safely used in your view's HTML DOM (e.g. in its CSS). This is needed because the host's HTTP server (which is delivering your view pages) may have a different '/' root than the root of your patch (e.g. if a single server is serving multiple patch GUIs).

Built-in javascript utility classes

The Cmajor runtime provides a collection of built-in helper classes that your module can use.

These are available via the `PatchConnection` object that is passed to your GUI or worker module when it is being created. The `PatchConnection` object has a `utilities` property containing various javascript objects for the modules in the API. To see what's available, have a look at the class in `cmajor/javascript/cmaj_api/cmaj-patch-connection.js`, and the various files in the `cmajor/javascript/cmaj_api` folder.

```
// In your GUI module:
export default function createPatchView (patchConnection)
{
    console.log (`Cmajor version: ${patchConnection.getCmajorVersion()}`);
    console.log (`MIDI message: ${patchConnection.midi.getMIDIDescription (0x924030)}`);

    const keyboard = new patchConnection.utilities.PianoKeyboard();
}
```

Patch Workers

A patch can specify a javascript program which is executed when the patch is loaded, and which can use timers and messaging to communicate with and control the running patch. It's obviously non-realtime, but can be used to do background housekeeping tasks when there is no GUI visible.

The API it uses to talk to the patch is the same `PatchConnection` class as described above.

To give your patch a worker, add a `worker` entry to the manifest file:

```
{
    "CmajorVersion":    1,
```

```
    "ID":               "dev.cmajor.examples.helloworld",
    "version":          "1.0",
    "name":             "Hello World",
    "source":           "HelloWorld.cmajor",

    "worker":           "./my_worker.js"
}
```

The value should be the path to a javascript module that will be executed. This module will be loaded at startup (or when the patch is reset),

The module must export a default function which will be called to do the work. This function will be passed a `PatchConnection` object which it can use to communicate with the patch, just like GUI code might do.

e.g.

```
export default function myWorker (patchConnection)
{
    patchConnection.addStatusListener ((status) => console.log (status));
    patchConnection.requestStatusUpdate();

    setTimeout (() => { patchConnection.sendEventOrValue ("myevent", 1234); }, 1000);
}
```

Depending on the VM that it's running in, this code may only have access to a restricted API, but a worker can expect to have at least:

- Basic built-in javascript language features
- `setTimeout()`, `setInterval()` and `clearInterval()` to manage timers

The `PatchConnection` object will have methods `async readResource(path)` and `async readResourceAsAudioData(path)` for reading data from files in the patch bundle. These take a path relative to the root of the bundle, and return a `Thenable` object which will deliver the data asynchronously.

The module can `import` other javascript modules, as long as these are built-in modules (e.g. `import * as midi from "./cmaj_api/cmaj-midi-helpers.js"` `) or files within the patch bundle.

Exporting patches

Building a native CLAP from a patch

The `cmaj` tool supports code-generation of a CLAP C++ project from a Cmajor patch. The project created is a pure static C++ project that doesn't perform any JIT compilation when it runs.

To use this feature, run the command line app in `generate` mode, e.g.

```
% cmaj generate --target=clap
                --output=[path to a target folder for the project]
                --clapIncludePath=[path to your CLAP include folder]
                MyAmazingPatch.cmajorpatch
```

It will create a folder containing some source files and a cmake project.

Building a native JUCE VST or AudioUnit from a patch

The `cmaj` tool supports code-generation of a JUCE C++ project that can be used to natively compile a VST/AudioUnit/AAX plugin for a patch. The resulting code doesn't do any JIT compilation, it simply translates the Cmajor code to pure C++ so that it can be built statically.

To use this feature, run the command line app in `generate` mode, e.g.

```
% cmaj generate --target=juce
                --output=[path to a target folder for the project]
                --jucePath=[path to your JUCE folder]
                MyAmazingPatch.cmajorpatch
```

It will create a folder containing some source files and a JUCE cmake project. This can be built as you would any other C++ cmake project, to produce VST/AU/AAX/standalone binaries.

Exporting a patch as Javascript/WebAssembly/Web Audio

The `cmaj` tool supports translation of a patch to a self-contained Javascript class (using WebAssembly for the DSP), and provides javascript helper classes to integrate with Web Audio and MIDI.

e.g.

```
% cmaj generate --target=webaudio-html
                --output=/path/to/MyAmazingSynthPatchFolder
                MyAmazingSynthPatch.cmajorpatch
```

There are several levels of javascript generation:

`--target=javascript`

This creates a single file containing a Javascript module which captures the patch's DSP behind an API similiar to the `cmaj::Performer` class. Internally, the generated class uses WebAssembly for its DSP, but this is hidden inside a javascript wrapper that provides a synchronous API to render the frames, handle endpoint i/o etc. The `--target=javascript` option exports only the patch's core DSP code, it doesn't provide any help with GUIs or playback.

`--target=webaudio`

This exports a folder containing both the javascript class for this patch (as created by `--target=javascript`) but also creates a set of helper modules for connecting it to web audio and MIDI.

This gives you a simple function `createAudioWorkletNodePatchConnection()` that returns a `PatchConnection` object that can be used to control it remotely.

The helpers also include a function to automatically connect this node to the browser's default audio and MIDI devices.

`--target=webaudio-html`

This option creates a folder which contains everything needed to run the patch and its GUI as a ready-to-go HTML/Javascript application. Just run a local webserver to server the generated folder, and open it in your browser. This is the easiest way to get started with javascript generation and to explore the generated classes.