



CONTROLLER API EXPLAINED

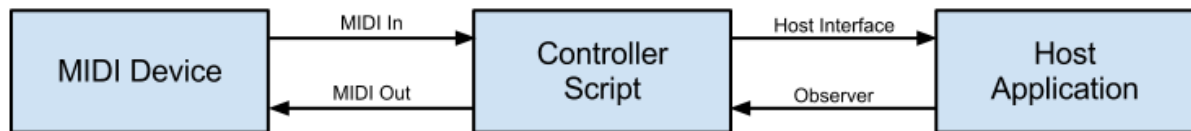
Table of Contents

Introduction.....	3
Where to put scripts.....	3
Controller Script Control-Flow.....	3
Defining & detecting the Controller.....	4
Loading the API.....	4
Defining the Controller.....	4
Implementing Auto-detection of the controller.....	4
1. Using the name of the MIDI input/output drivers.....	4
2. Specifying the expected SysEx Identity Reply.....	5
Implementing the Controller.....	5
Creating Views.....	5
Set Callback functions for MIDI / SysEx.....	6
Create Note Inputs.....	6
Adding Observers.....	7
Adding observers to a range (JavaScript scope/closure gotcha).....	8
Sending MIDI / SysEx.....	9
Example: SimpleTransport.....	9
Controller Script Console.....	10
Special console commands.....	11
Parameters.....	11
Tracks.....	11
User Control Mapping.....	11
Devices.....	12
Macros.....	12
Parameter Banks.....	12
Common Parameters.....	12
Envelope Parameters.....	13
Modulation Sources.....	13
Controller Feedback in the Bitwig Studio GUI.....	13
For Parameters.....	13
For Clip Launcher.....	13
Display Overlay.....	14

Introduction

Bitwig's Controller API provides support for rich controller hardware, where bi-directional communication between the controller and the host software is needed.

The API lets a script (implemented in JavaScript) act as an intermediate which talks to the MIDI hardware at one end and the host application interface in the other. In the rest of this document its expected that the reader has knowledge of JavaScript and the MIDI protocol.



Where to put scripts

OS	Location
Windows	%USERPROFILE%\Documents\Bitwig Studio\Controller Scripts\
Mac and Linux	~/Documents/Bitwig Studio/Controller Scripts/

They should be put in a sub-directory named after the device vendor and the file extension control.js.

Controller Script Control-Flow

An event-driven approach is used for execution of the controller scripts.

This means that the script has no run-loop of its own, instead the code within the script will only be executed when one of the following happens:

- The script is being initialized -> script is evaluated and `init()` gets called
- A MIDI (or SysEx) message has arrived on an In Port which was assigned a callback function in `init()`.
- Something on the document happened which has an Observer watching it. (note: observers send both the initial state and changes)
- After MIDI input and/or observer feedback has been sent to the script, the function `flush()` will be called once. This is the recommended point to send MIDI to drive displays and other things that use up much of MIDI's bandwidth.

In general, this means that all the script has to worry about is: "when x happens, do y". Threading is taken care of by the host application.

Defining & detecting the Controller

Loading the API

The first line of code in a typical controller script will be

```
loadAPI(1);
```

Which will initialize the script with the desired version of the API (1 in this case), and add an object called `host` to the global scope which implements the Host interface.

A set of JavaScript-files which provide various convenience and utility methods which make working with the API more convenient will also be loaded (located in `resources/controllers/api/**`).

The accompanying java-doc documentation acts as a reference on what is available from the host interface and the syntax for calling it.

Defining the Controller

```
// example  
host.defineController("ACME", "product 2000", "1.0", "INSERT UUID HERE");
```

This must be called for every script. It will define the vendor, name and version of the script. In addition it will define the UUID (Universally Unique Identifier) of the script, which is used to identify which script is which and must therefore be totally unique. If you don't know how to generate a UUID you can use this online generator (<http://www.famkruithof.net/uuid/uuidgen>).

```
// example  
host.defineMidiPorts(1, 1);
```

This defines the number of MIDI in- and out-ports the script is expecting.

Implementing Auto-detection of the controller

A controller script can be configured to allow the controller to be automatically detected in two ways:

1. Using the name of the MIDI input/output drivers

This method takes an array of strings representing the input driver names and an array representing the output names. (this must match the number of inputs / outputs specified to `defineMidiPorts`)

```
// example  
host.addDeviceNameBasedDiscoveryPair(["PANORAMA P4"], ["PANORAMA P4"]);
```

This can method can be invoked multiple times to register multiple possible driver name-combinations. One example why this would be necessary is if the controller has a different name on different platforms.

2. Specifying the expected SysEx Identity Reply

If the controller can't be detected based on MIDI driver names alone (which would be the case for a non-USB device), the script can also rely on the SysEx DEVICE INQUIRY. When scanning for MIDI Controllers Bitwig Studio will send the DEVICE INQUIRY message (F0 7E 7F 06 01 F7) on all ports in sequence, and see what responses it gets within a reasonable period of time.

By specifying the expected SysEx response from the controller, which can contain wild-cards in the form of the question mark '?' character, the application can match a specific MIDI IN/OUT-pair to a specific script.

```
host.defineSysexIdentityReply("F0 7E ?? 06 02 00 20 29 03 00 03 00 ?? ?? ?? ?? F7");
```

This detection method only works with scripts that expect 1 MIDI input and 1 MIDI output.

Implementing the Controller

Until now, we dealt with the code needed to define the controller and detect it. And unless it is being instantiated that is all the script needs to be able to do.

For a script to be instantiated and actually do anything there are two functions in the global scope which need to be implemented: `init()` and `exit()`.

`init()` will be called when a script is started and is expected to:

- create views via the host interface
- set callback functions for MIDI and SysEx messages on the desired in-ports we want to use.
- create note inputs
- add observers on the host interface for data we want to monitor
- send MIDI to initialize the controller to the desired initial state

`exit()` will be called as the script is shutting down. Usually you don't need to do anything here, but it can be used to restore the hardware controller to a certain state if needed. This is particularly useful for MIDI controllers which has a "native mode" which is different from the normal operating mode of the device.

Creating Views

A view is an object which provides a view onto an object in the document of that type. It doesn't equate to a single object, instead it will change the object it is assigned to at the applications whim, typically because the user changed the selection (cursorTrack), or because the script told it to. There are a multitude of views available for various areas like transport, application shortcuts, track, track-banks, groove, clip launcher, clip content editing.

For a full documentation of the views which are available from the host interface consult the javadoc. Below are just a few examples.

```

function init()
{
    transport = host.createTransport();

    // A track-bank view acts as a N-track window onto the document's tracks
    trackBank = host.createTrackBank(8 /* num tracks */, 1 /* num sends */, 0 /* num scenes
*/);

    // The cursor track view follows the track selection in the application GUI
    cursorTrack = host.createCursorTrack(4 /* num sends */, 4 /* num scenes */);

    userControls = host.createUserControls(8 /* numControls */ );
}

```

Views must be created within the call to `init()`. It is not allowed to create them dynamically via Observer or MIDI / SysEx callbacks.

Set Callback functions for MIDI / SysEx

In order to be able to react to incoming MIDI data we first need to provide a callback function which will be called when a MIDI / SysEx event is received on that port.

MIDI / SysEx callback example

```

function init()
{
    transport = host.createTransport();

    host.getMidiInPort(0).setMidiCallback(onMidi);
    host.getMidiInPort(0).setSysexCallback(onSysex);
}

function onMidi(status, data1, data2)
{
    // handle normal short MIDI messages here
    // when we receive a CC #40 with a value bigger than 0 we tell the transport to play
    if (isChannelController(status) && data1 == 40 && data2 > 0)
    {
        transport.play();
    }
}

function onSysex(data)
{
    // handle SysEx MIDI messages here
}

```

Callbacks must be created within `init()`.

Create Note Inputs

In order to use a MIDI controller as a note input device, a note input must be specified. It is given a name and a list of MIDI message filters (in hex, with ?? as wildcard). Any MIDI messages which match any of the filter strings will be sent directly to the application as note input and be excluded from the events being sent to the MIDI callback functions.

```
function init()
{
    noteIn = host.getMidiInPort(0).createNoteInput("ACME Keyboard", "80????", "90????",
"B001??", "B040??", "D0????", "E0????");
}
```

If no MIDI filters are provided, a set of defaults will be used for MIDI Channel 1 (0).

Note-inputs show up as a selectable note source within the application. They must be created within `init()`.

Adding Observers

The host interface doesn't provide a means of reading any values from the object it provides directly. Instead it provides an asynchronous model where you add an observer on values which you are interested in. When adding an observer you supply a callback function that you pass as an argument along with other parameters for how the observer should work. The callback function will be called with the initial state of the value you are observing but also be called whenever the observer value changes.

```
function init()
{
    transport = host.createTransport();

    // Add an observer which prints the transport PLAY / STOP state to the controller script
    console
    transport.addIsPlayingObserver(function(isPlaying)
    {
        println(isPlaying ? "PLAY" : "STOP");
    });

    cursorTrack = host.createCursorTrack(2, 0);

    // Add an observer which prints volume of the cursor track with 128 steps to the console
    cursorTrack.getVolume().addValueObserver(128, function(value)
    {
        println("VOLUME : " + value);
    });

    // Add an observer which prints a formatted text representation of the cursor track volume
    // with a maximum of 8 characters to the console
    cursorTrack.getVolume().addValueDisplayObserver(8, "", function(text)
    {
        println("VOLUME : " + text);
    });
}
```

Observers are only allowed to be added within `init()`. There is no need to clean up observers, they are all removed automatically as the script exits.

Adding observers to a range (JavaScript scope/closure gotcha)

When adding a range of observers using a for loop or other iteration mechanism there is a certain aspect of JavaScript that you need to be aware of. Consider the following example

```

function init()
{
    trackBank = host.createMainTrackBankSection(NUM_TRACKS, NUM_SENDS, NUM_SCENES);

    for(var t=0; t<NUM_TRACKS; t++)
    {
        // This Looks Like it would make sense, but it doesn't work
        trackBank.getTrack(t).addNameObserver(8, "", function(name)
        {
            println("Track " + t + " name: " + name);
        });
    }
}

```

The code above sure look like it makes sense, you would expect to get a bunch of updates coming in from the various tracks but due to the way JavaScript does scoping it wont work. Instead you will get a whole bunch of notifications coming to only the last track.

My IDE will give me the warning "Mutable variable is accessible from closure, and that is exactly whats happening. The variable t, as given in to the function we pass as a callback, is defined the scope of the for loop, and t is modified as the for loop goes on. So at the time when the callback is actually called, t will most likely be 8 regardless of what the value of t was when the callback was added.

To fix this, we need to wrap the inner part of the loop with a function in order to provide a scope where each value of t is remembered. To make this more readable and reusable in this example I added a makeIndexedFunction to the global scope, but this could also be done inline, the important part is that the function (makeIndexedFunction) provides a scope where each iteration has a different value of index in the callback.


```

function makeIndexedFunction(index, f)
{
    return function(value)
    {
        f(index, value);
    };
}

function init()
{
    trackBank = host.createMainTrackBankSection(NUM_TRACKS, NUM_SENDS, NUM_SCENES);

    for(var t=0; t<NUM_TRACKS; t++)
    {
        trackBank.getTrack(t).addNameObserver(8, "", makeIndexedFunction(t, function(name)
        {
            println("Track " + t + " name: " + name);
        })));
    }
}

```

More reading on JavaScript scoping/closure rules:

<http://stackoverflow.com/questions/13813463/how-to-avoid-access-mutable-variable-from-closure>

<http://stackoverflow.com/questions/111102/how-do-javascript-closures-work>

Sending MIDI / SysEx

MIDI / SysEx messages can be sent like this:

```

host.getMidiOutPort(0).sendMidi(status, data1, data2);
// status, data1, data2 are the 3 bytes of normal MIDI message

host.getMidiOutPort(0).sendSysex("F0 00 01 77 7F 01 09 00 00 00 01 00 75 F7");
// sendSysex expects a string formatted as Hex, Upper/lower-case is irrelevant and
spaces are ignored.

```

Most scripts will only use a single MIDI out port so we have provided a bunch of convenience methods which act on MIDI Out Port 0.

```

// Send MIDI / SysEx to Out Port 0
function sendMidi(status, data1, data2);
function sendSysex(data);

```

Example: SimpleTransport

In order to show the very basics about how communication both FROM and TO the controller work in practice we will here show how a script of the fictional device "ACME SimpleTransport" would look.

The imagined MIDI hardware works as a transport controller and provides the following (all on MIDI Channel 1 (0)).

- Play button which sends MIDI CC 40 with value 127 when pressed and 0 when released.

- Stop button which sends MIDI CC 41 with value 127 when pressed and 0 when released.
- LED next to the play button which lights when MIDI CC 40 is greater than 63.

Example controller script

```
loadAPI(1);

host.defineController("ACME", "SimpleTransport", "1.0", "F23ABCE1-7BA8-4996-A769-25A7E1F8211E");
host.defineMidiPorts(1, 1);
host.addDeviceNameBasedDiscoveryPair(["SimpleTransport"], ["SimpleTransport"]);

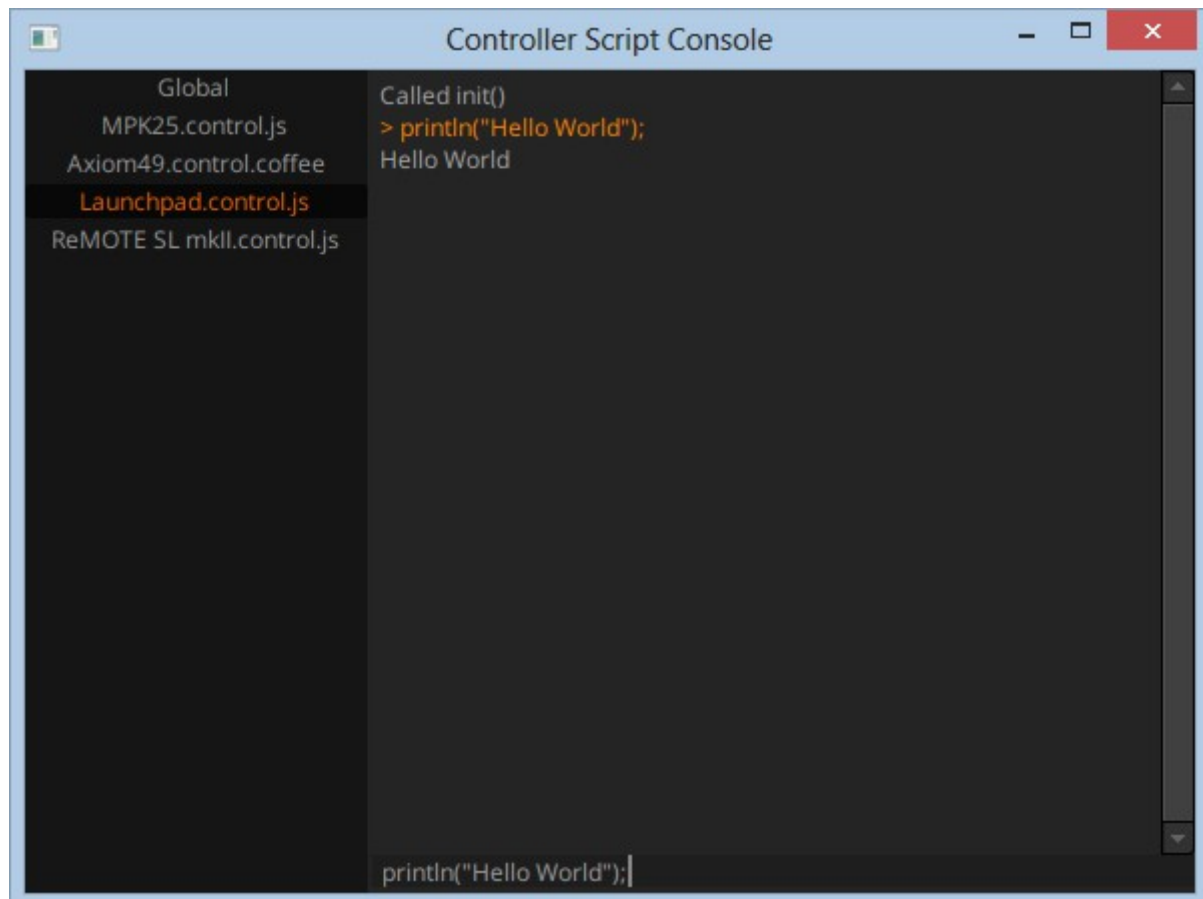
function init()
{
  host.getMidiInPort(0).setMidiCallback(onMidi);

  transport = host.createTransport();
  transport.addIsPlayingObserver(function(isPlaying)
  }
```

Notice how the code is entirely event-based. This means that it has no concept of a run-loop which keeps running. Instead it will react on events which happen either via the MIDI in port (controller sends midi -> onMidi) or the observers added to the host interface (isPlaying changes -> call the function sent in as an argument).

Controller Script Console

Select *View / Show Controller Script Console* to open the console window.



- The window will show you a tab for each script which has had any output since the application started.
- In each tab you can read the normal output (gray), error output (red) and user input messages (orange) for that particular script.
- With the text entry bar at the bottom of the windows you can type commands which are evaluated by the JavaScript interpreter instantiated for that script, or one of the special commands listed below.

Special console commands

restart will reload the script from disk and restart it.

Parameters

Most parameters you will come in contact with will implement the AutomatableRangedValue interface.

Some AutomatableRangedValue methods:

```
void addNameObserver(int maxChars, String textWhenUnassigned, Callable callback);
void addValueDisplayObserver(int maxChars, String textWhenUnassigned, Callable callback);
void reset();
void touch(boolean isBeingTouched);
void setIndication(boolean shouldIndicate);
void setLabel(String label);
void addValueObserver(int range, Callable callback);
void set(Number value, Number resolution);
void inc(Number increment, Number resolution);
```

Tracks

A track can either be accessed as a single track view (via `createCursorTrack`, `createMasterTrack`) or as a track bank view object. A track bank acts as a window onto N tracks of the application. You can either get a track-bank looking onto all tracks of the document or a bank looking at only the main or effect tracks.

The track bank can then be shifted around using the methods (`scrollTracksPageUp`, `scrollTracksPageDown`, `scrollTracksUp`, `scrollTracksDown`).

A usage example would be a mixer-controller with 9 motorized faders where the last one should act as a master fader. You would use a 8-track bank to get the 8 main tracks, make button presses for next/prev bank call `scrollTracksPageUp` / `scrollTracksPageDown` and make the 9th fader control (and observe the value of) the track object given by `createMasterTrack`.

A method for accessing all the tracks of the document isn't available.

User Control Mapping

In Bitwig Studio there is no concept of MIDI learn as is commonplace in traditional DAW

software. The application itself is MIDI agnostic in that sense, as MIDI is only used by the script to convert that into higher-level actions. This means that you can't just turn MIDI CC 56 and expect it to be learn-able. Which is good, since how is BWS supposed to know exactly how the data of CC 56 is encoded? It could be a relative change, or maybe its a MSB/LSB pair together with another controller and there is no way of knowing. Likewise the user is not really informed reading that CC 56 is mapped somewhere as it doesn't tell him which encoder to turn. There would also be absolutely no way of providing bidirectional feedback on the mapped parameters.

Instead it is required that a hardware control results in calling set (or inc) on some kind of value object. If not they're simply not map-able. Most value views obtained through the API can have their automatic assignment overridden by the user by selecting "Learn Controller Assignment..." from the context menu of a parameter in the GUI. But you can also create a user controls view which is intended for only this use, and has no automatic assignment. For most controllers it will make sense to have a dedicated user page/mode where the controls are freely assignable.

In contrast to the traditional MIDI mapping approach, observers still work for user mapped parameters so full bidirectional feedback is possible.

The list of parameters mappings for a control script in a document can be seen under the Studio I/O panel in the GUI. Make sure to provide a label for each value view to make it possible to see which control in the script the parameter in the application is mapped to.

Devices

There are currently two ways of getting access to a device through the API. Either get the cursor device (selected device in the GUI) or the primary instrument from a particular track.

Macros

Each device has 8 macro controls which can be freely assigned for each device preset. For simpler controllers this is probably everything that you need to assign.

Parameter Banks

The devices contain a data model of parameter pages containing 8 parameter mappings each. These parameters provide the primary interface to the main device panel. This is what allows a rich controller with a display to provide pretty much the same user experience as a hardware synth.

In rich controllers (something like Remote SL mkII or Panorama P4) which have enough on-panel controls, the common & envelopes parameters can be provided as a complement to this.

Common Parameters

The parameter mapping data model contains a set of 8 additional parameters which are intended to be used in conjunction with the parameter banks. This is will be mapped to

parameters which has a more global role. On the polysynth for example the oscillator mix is available from here in the default mapping, as it is such a parameter that you want to use at the same time as you are tweaking the settings of an individual oscillator.

Envelope Parameters

An additional set of 8 parameters that are mapped to parameters particularly suited for faders. (like 2 x ADSR or drawbars).

Modulation Sources

These can also be activated & observed, causing the controls to go into modulation mapping mode. When active, values changes from the controller API will change the modulation amount instead of the parameter value, and observers will provide information relevant for modulation.

Controller Feedback in the Bitwig Studio GUI

For Parameters

An age-old problem with controller mappings, and automatic mappings in particular, is that its very hard for the user to know that he's actually controlling. Only the more expensive controllers provide some kind of display to show the parameter names. We have solved this problem by providing a colored overlay on top of the parameters which are currently being controlled. That way it will be immediately obvious what the user is controlling, and it will be an intuitive experience to switch between different devices, parameter banks and modes of the controller.



A script should call `setIndication` on value views to indicate which are currently mapped to hardware controls by the script. When having different modes inside a script, it should only set this to true on the controls which are relevant for that mode. This will provide graphical feedback in the GUI to see what is currently being controlled and makes it much easier to navigate through tracks and parameter banks.

For Clip Launcher

By calling `setIndication` on the clip launcher view, a frame will be shown on the slots to indicate which area is currently mapped.

Display Overlay

By calling `host.showPopupNotification` a temporary popup overlay is shown. This can be used to provide quick feedback for controllers without a display (for example when switching modes, selecting tracks/devices/parameter banks or modifying parameter values). It will start fading out after 2 seconds.