^ Introduction

^ Special Variables

^ Functions

^ ReadMe

^ Valid Commands for wwr_req

SET/TRACK/index/WIDTH/value
SET/TRACK/index/MUTE/value
SET/TRACK/index/SOLO/value
SET/TRACK/index/FX/value
SET/TRACK/index/RECARM/value
SET/TRACK/index/RECMON/value
SET/TRACK/index/SEL/value
SET/UNDO/message
SET/UNDO_BEGIN
SET/UNDO_END/message
SET/REPEAT/val
GET/REPEAT
GET/REPEAT \t val
SET/TRACK/x/SEND/y/MUTE/value
SET/TRACK/x/SEND/y/VOL/value
SET/TRACK/x/SEND/y/PAN/value
SET/POS/value_in_seconds
SET/POS_STR/value_string
LYRICS/trackindex
SET/PROJEXTSTATE/section/key/value
GET/PROJEXTSTATE/section/key
GET/EXTSTATE/section/key
SET/EXTSTATE/section/key/value
SET/EXTSTATEPERSIST/section/key/value
GET/command_id
OSC/oscstring[:value]
MARKER
REGION

---

^ Introduction

Reaper can be controlled via a web browser interface: Options->Preferences->Control/OSC/Web->Add->Web Browser Interface

where you can choose between several interfaces, included with reaper(basic, click, index, lyrics). The Webinterface can be accessed i.e. via the local internet-adress localhost:8080 or localhost:8080/(interfacename(like click)).html

It can also be accessed via Wireless Lan as well as using rc.reaper.fm(for i.e. cough-buttons with long distance Skype/Studiolink-interview partners).

The Web-Interface-HTML-files can be found, altered or put to the folder Reaper/Plugins/reaper_www_root/ The folder Plugins/reaper_www_root in the application folder is for the standard "built in"-Reaper-WebAPI-files. If you want to place your own files, you should use the reaper_www_root-folder in the ressources folder or in the application-folder(the latter, if you have a portable installation of Reaper).

You can place the files into the application-folder Plugins/reaper_www_root as well, but if you want to export them with the export-function within Reaper, they will simply be ignored.

^ How it works - basics

The Web-Interface is programmed in JavaScript. You need to include main.js first, then call wwr_start(), which tells the Reaper-Server to accept requests. You can now give Reaper requests, using commands like i.e. wwr_req(ActionCommandID) for one time requests. The ActionCommandID tells Reaper, what you want to call, i.e. the ActionCommandID 1007 starts play in Reaper. You can find a full Action Command ID-list in the Action-Dialog in the menu Actions -> Show Action List. Example Code that presses the Play-Button:

```
<script src="main.js">
    /*includes the main.js, which contains a lot of helper functions*/
</script>
<script type="text/javascript">
    wwr_start();   /*tells Reaper to accept requests*/
    wwr_req(1007); /*presses the Play-Button*/
</script>
```

You can also start Reascripts or custom actions. You just need to go to Actions->Show Actions List, then you search for the script or action you want to use. Click on it with right-mouse-click and choose "Copy Selected Action Command ID". Paste it into the following example-code:

```
<script src="main.js">
    /*includes the main.js, which contains a lot of helper functions*/
</script>
<script type="text/javascript">
    wwr_start();   /*tells Reaper to accept requests*/
    wwr_req("_ActionCommandCode"); /*replace "_Action Command Code", by
                                     the action-command-code you copied f
                                     It could look something like this:
                                             _RS52af5bc23917ea88f42d82f06
                                     Please note: the Action-Command-Code
                                                  start with an underscor
</script>
```

^ How to exchange data and get states from Reaper?

Sending data to Reaper

Sending messages to Reaper, using requests, is easy, as it is only a normal request, where you exchange the parameters with your own. The parameters can be written inside of the request or exchanged by variables. For instance, if you want to set a persistent external state, you could use the request SET/EXTSTATEPERSIST/section/key/value and it would look like the following example:

Example1

```
<html>
   <head>
       <script src="main.js"></script>
       <script type="text/javascript">
           var section="IamASection";
           var key="IamAKey";
           var value="IamTheValue";

           wwr_start();
           wwr_req("SET/EXTSTATEPERSIST/"+section+"/"+key+"/"+value);
       </script>
```

This extstate can now be read from ReaScript like this:

```
Value = reaper.GetExtState("IamASection", "IamAKey")
reaper.MB(Value, "The Value, that has been sent from WebRC", 0)
```

Receiving data from Reaper

When you call a request, that returns data/values/states, like (TRANSPORT, MARKER or GET/EXTSTATE/section/key), the main.js automatically calls a function called wwr_onreply(results).
This function isn't there by default but must be rather defined by you in your script.
In that function, the variable "result" contains the response from Reaper, which can either be a one-liner(like with TRANSPORT) or multiliner(like when getting the data for all markers in your project with MARKER).
Multiliners are separated by a newline "\n", the individual entries in each line are separated with tabs "\t". So if you want to get the individual values, newline and tab will make you very happy.
Every line in the response starts with the exact request you've sent(a "TRANSPORT"-request returns a line, that begins with "TRANSPORT"), so you know exactly, from what request the response came-from. You can also differentiate between different kinds of requests that way, as different requests return different ways of data, states, etc.
There are exceptions to this rule though(e.g. "MARKER", which starts with "MARKER_LIST"), so, if in doubt, just check the "results"-variable using an alert-message and look for yourself, how the response starts. But relax, they keep the same, means: "MARKER" will always return "MARKER_LIST" and not something like Jamboreepop out of a sudden.
The second example in this chapter will show you how to do it.

Example1:
This is a simple example, that shows how to get data from Reaper using the wwr-request "TRANSPORT" and display it with the Javascript-alert-function. Feel free to modify and experiment in the wwr_req()-line, by exchanging "TRANSPORT" with another request(more requests later in this documentation). This example uses code from the lyrics.html page, as included with Reaper. Always include the main.js!

```
<html>
    <head>
        <script src="main.js"></script>

        <script type="text/javascript">

            function wwr_onreply(results) {
              //main.js calls wwr_onreply with the response, as sent from Re
              //wwr_req("TRANSPORT") is called.

              alert("Reaper returned: "+results); //shows a messagebox with
            }
```

Example2:
The next example shows you, what to do with the returned data, how to handle it, separate it(also multiline-requests) and putting it into global variables to be used later. It also displays you the content of the variables with an alert()-message. Press the button to execute the request again. Again, this example works with the "TRANSPORT"-request, but covers briefly the request "MARKER" as well. Feel free to experiment with other requests as well to get an idea. This example uses code from the lyrics.html page, as included with Reaper. Always include the main.js!

```
<html>
    <head>
        <script src="main.js"></script>

        <script type="text/javascript">

        // Setup the global variables, that will contain the Transport-State
            var playstate="";
            var playpos="099";
            var isRepeat="";
            var position_string="";
            var position_string_beats="";
```

---

^ Security Aspects

The internal server in Reaper can be killed with a Denial of Service-Attack. It seems, as it can process 1000 requests in a short time without a problem. More requests than 1000 can lead Reaper to simply ignore them. An even higher amount of requests(i.e. 10.000 and more) could cause Reaper to hang for a few seconds(maybe even longer?). What effects this has during recording or playback, needs to be researched more.
Setting a username and password in the Reaper-preferences for the Web-Interface is definately a good idea.

If the Web-Interface-API has any kind of https-support needs to be researched.

---

^ Functions and Variables

The main.js-file, which manages the exchange of requests and data, includes some little helpers with it, that you can use.

Note: If you create your own functions and variables, avoid names with a wwr or mk in it, as the Webinterface-functions and variables are named with them. That way, you can avoid name-conflicts with future versions of the Reaper-Webinterface.

---

^ 6.13 g_wwr_timer_freq

Javascript: `g_wwr_timer_freq`

A global variable, that stores the timer-frequency for requests. Default is 100 for 100ms, but can be changed to any other value in milliseconds, before calling wwr_start(). Don't choose a too low value, as this could cause Reaper becoming stuck and unresponsive!

---

^ 6.13 wwr_start

Javascript: `wwr_start()`

Starts the server. After that, requests can be send to Reaper.

---

^ 6.13 wwr_req

Javascript: `wwr_req(string request)`

Sends a one-time request to Reaper. Can be a commandID-number, an "_ActionID" or one of the requests listed in the "Readme"-chapter later below.

If a request returns values, they can be accessed in the function wwr_onreply. Refer the Receiving data from Reaper earlier in this document.

Parameters:

string request

The request to be sent to Reaper. Can be a commandID-number, an "_ActionID" or one of the requests listed in the "Readme"-chapter.

---

^ 6.13 wwr_req_recur

Javascript: `wwr_req_recur(string request, float timer_frequency)`

Just like wwr_req(), but resends the request automatically with a frequency set in timer_frequency. Must be stopped with wwr_req_recur_cancel()

Parameters:

string request

The request to be sent to Reaper. Can be a commandID-number, an "_ActionID" or one of the requests listed in the "Readme"-chapter.

float timer_frequency

the timer frequency in milliseconds, that decides, how often this request shall be repeated. Don't choose to low, or Reaper might get stuck and unresponsive!

---

^ 6.13 **wwr_req_recur_cancel**

Javascript: `wwr_req_recur_cancel(string request)`

Stops a recurring request, that has been started with wwr_req_recur. Can be a commandID-number, an "_ActionID" or one of the requests listed in the "Readme"-chapter later below.

Parameters:

string request

The request to be stopped from being sent to Reaper. Can be a commandID-number, an "_ActionID" or one of the requests listed in the "Readme"-chapter.

---

^ 6.13 **mkvolstr**

Javascript: `string db_value = mkvolstr(float vol)`

converts a volume-value(as returned by e.g. the request TRANSPORT") into a db-value and returns it. The string could be like "123.45 db" or "-inf db"(if the vol-value is too low).

Returnvalues:

string db_value

the converted value as dB

Parameters:

float vol

a volume-number as returned from a request like e.g. TRANSPORT. Only positive values are allowed. Minimum 0.00000002980232. Lower returns -inf.

---

^ 6.13 **mkpanstr**

Javascript: `string pan_percentage_value = mkpanstr(float pan)`

returns the pan value converted into percentage. The returned value could look like "10%L" or "25%R" or "center". The function allows values that produce more than 100% left or right, i.e. "1234%L". Keep that in mind!

Returnvalues:

string pan_percentage_value

the converted pan-value as percentage

Parameters:

float pan

a pan-number as returned from a request like e.g. TRANSPORT. Negative values=left, positive values=right. -1=100%left, 1-100%right, 0.001 to -0.001=center.

---

^ 6.13 **simple_unescape**

Javascript: `string unescaped_values = simple_unescape(string v)`

unescapes \\n to \n and \\t to \t and \\ to \ within a string. Important, as some responses from requests escape tabs, newline and backslashes that way.

Returnvalues:

string unescaped_values

the string where all newlines, tabs and backslashes are unescaped

Parameters:

string v

a string, that contains not-yet-unescaped newlines, tabs and backslashes

---

^ 6.13 **Readme from Reaper\plugins\reaper_www_root\main.js**

1) call wwr_start() to begin requests to the server
2) you can call wwr_req_recur("REQUEST", interval) to set recurring requests (such as status updates). Interval is in milliseconds.
3) you can call wwr_req_recur_cancel("REQUEST") to remove a recurring request that was previously set.
4) you can call wwr_req("REQUEST") to send a one-time request.
5) g_wwr_timer_freq can be overridden before calling wwr_start() to increase the timer frequency (the default is 100, for 100ms)
6) Responses:
You should define a function named wwr_onreply:

```
function wwr_onreply(results) {
  var ar = results.split("\n");
  var x;
  for (x=0;x < ar.length;x++)
  {
    var tok = ar[x].split("\t");
    // tok is a list of parameters, the first being the command
  }
}
```

See index.html for an example.

^ Reference

All interaction with the server is done via
    wwr_req("command;command;command")
or
    wwr_req_recur("command;command",interval):
(which are internally sent as /_/command;command;command;command to the server)
There can be any reasonable number of commands, separated by semicolons. They will be executed in order, and responses may be given, depending on the commands and the server.
The format of the response is a list of lines (separated by \n), and each line has tokens separated by \t.

---

^ Valid commands

There's a variety of commands available, that can be send to Reaper as request.

---

^ numeric values 0-65535

REAPER command IDs. (todo: support for registered names, too). These typically do not have any response.
_123456789abcdef...
REAPER plug-in registered command IDs (also used by ReaScripts and custom actions)

---

^ TRANSPORT

Returns a line including (note that the spaces are here for readability, there is actually only tabs between fields):

    TRANSPORT \t playstate \t position_seconds \t isRepeatOn \t position_string \t position_string_beats

- playstate is 0 for stopped, 1 for playing, 2 for paused, 5 for recording, and 6 for record paused.
- isRepeatOn will be nonzero if repeat is enabled.
- position_string is always in the project timeline format (time, beats, etc), position_string_beats is always in measures.beats.hundredths format.

---

^ BEATPOS

Returns a line:

    BEATPOS \t playstate \t position_seconds \t full_beat_position \t measure_cnt \t

beats_in_measure \t ts_numerator \t ts_denominator

---

^ 6.13 NTRACK

Requests track count. Returns a line:

NTRACK \t value

value is 0 (no tracks, just master track) or greater.

---

^ 6.13 TRACK or TRACK/index or TRACK/start-end

Requests information about all tracks, a single track, or a range of tracks. Note that the indices used are 0 for master, 1 for first user track, etc.

Returns any number of track lines:

TRACK \t tracknumber \t trackname \t trackflags \t volume \t pan \t last_meter_peak \t last_meter_pos \t width/pan2 \t panmode \t sendcnt \t recvcnt \t hwoutcnt \t color

tracknumber is 0 for master, 1 for first track, etc.
trackname is the name of the track, or MASTER
trackflags includes various bits (test with parseInt(field)&1 etc):
1: folder
2: selected
4: has FX
8: muted
16: soloed (32: solo-in-place)
64: record armed
128: record monitoring on
256: record monitoring auto
volume is track volume, where 0 is -inf, 1 is +0dB, etc. see mkvolstr for dB conversions
pan is -1..1, where -1 is full left, 0 is centered, 1 is full right
last_meter_peak and last_meter_pos are integers that are dB*10, so -100 would be -10dB.
color is in the form of 0xaarrggbb, nonzero if a custom color set

---

^ 6.13 GET/TRACK/x/SEND/y

Gets state for track x hardware output/send y. Returns a line:

SEND \t x \t y \t flags \t volume \t pan \t other_track_index

Use y=-1 for first receive, -2 for second, etc.
other_track_index is -1 if hardware output
flags & 8 is true if send/output muted

---

^ 6.13 GET/TRACK/index/xxxxxx

Requests information for track "index", via GetSetMediaTrackInfo(). See the REAPER API documentation for which strings are acceptable, but for example you can query the track record input index for the first track via:

---

^ 6.13 GET/TRACK/1/I_RECINPUT

The returned string will be:

    GET/TRACK/1/I_RECINPUT\t

(if an error occurs you may get nothing back at all, or you might get the GET string back but with no parameter)
String will have newlines/tabs/backslashes encoded as \\n, \\t and \\.

---

^ 6.13 SET/TRACK/index/xxxxx/value

Similar to GET/TRACK/index/xxxxx, but sets the value of this item. You probably will want to follow this with a SET/UNDO command, below. This will not give any response.

---

^ 6.13 SET/TRACK/index/VOL/value

Special case of SET/TRACK/index/xxxx, sets volume for a track via control surface API (meaning it respects automation modes etc). If value starts with + or -, then adjustment is relative (in dB), otherwise adjustment is absolute (1=0dB, etc). If value ends in "g", then ganging is ignored. Does not need SET/UNDO.

---

^ 6.13 SET/TRACK/index/PAN/value

Special case of SET/TRACK/index/xxxx, sets pan for a track via control surface API. If value starts with + or -, adjustment is relative. Range is always -1..1. If value ends in "g", then ganging is ignored. Does not need SET/UNDO.

---

^ 6.13 SET/TRACK/index/WIDTH/value

Special case of SET/TRACK/index/xxxx, sets width for a track via control surface API. If value starts with + or -, adjustment is relative. Range is always -1..1. If value ends in "g", then ganging is ignored. Does not need SET/UNDO.

---

^ 6.13 **SET/TRACK/index/MUTE/value**

Special case of SET/TRACK/index/xxxx, sets mute for track. if value is <0, mute is toggled. Does not need SET/UNDO.

---

^ 6.13 **SET/TRACK/index/SOLO/value**

Special case of SET/TRACK/index/xxxx, sets solo for track. if value is <0, solo is toggled. Does not need SET/UNDO.

---

^ 6.13 **SET/TRACK/index/FX/value**

Special case of SET/TRACK/index/xxxx, sets fx enabled for track. if value is <0, fx enabled is toggled. Does not need SET/UNDO.

---

^ 6.13 **SET/TRACK/index/RECARM/value**

Special case of SET/TRACK/index/xxxx, sets record arm enabled for track. if value is <0, record arm enabled is toggled. Does not need SET/UNDO.

---

^ 6.13 **SET/TRACK/index/RECMON/value**

Special case of SET/TRACK/index/xxxx, sets record monitoring for track. if value is <0, record monitoring is cycled, otherwise 1=on, 2=auto. Does not need SET/UNDO.

---

^ 6.13 **SET/TRACK/index/SEL/value**

Special case of SET/TRACK/index/xxxx, sets selection for track. if value is <0, selection is toggled. Does not need SET/UNDO.

---

^ 6.13 **SET/UNDO/message**

Adds an undo point, useful if you have modified anything that needs it.

---

^ 6.13 **SET/UNDO_BEGIN**

Begins an undo block (should always be matched with SET/UNDO_END!)

---

^ **SET/UNDO_END/message**

Ends an undo block

---

^ **SET/REPEAT/val**

If val is -1, toggles repeat, 0 disables repeat, 1 enables repeat.

---

^ **GET/REPEAT**

Returns:

---

^ **GET/REPEAT \t val**

where val is 0 or 1.

---

^ **SET/TRACK/x/SEND/y/MUTE/value**

Sets hardware output/send mute for track x, send y. value can be -1 to toggle.
Use y=-1 for first receive, -2 for second, etc.

---

^ **SET/TRACK/x/SEND/y/VOL/value**

Sets hardware output/send volume (1.0 = +0dB) for track x, send y. append e to value to treat as "end" (capture), or "E" to treat as an instant edit.
Use y=-1 for first receive, -2 for second, etc.

---

^ **SET/TRACK/x/SEND/y/PAN/value**

Sets hardware output/send pan (0=center, -1=left) for track x, send y. append e to value to treat as "end" (capture), or "E" to treat as an instant edit.
Use y=-1 for first receive, -2 for second, etc.

---

^ **SET/POS/value_in_seconds**

Sets edit cursor position (seeking playback) to value_in_seconds

---

^ 6.13 **SET/POS_STR/value_string**

Sets edit cursor position (seeking playback) to value_string (format auto-detected)
r1 goes to region ID 1, m1 to marker 1, R1 to first timeline region, M1 to first timeline marker

---

^ 6.13 **LYRICS/trackindex**

Returns:
LYRICS \t trackindex \t beat_position \t lyric \t ...
Retrieves MIDI lyrics for trackindex.
String will have newlines/tabs/backslashes encoded as \\n, \\t and \\. Length is limited to around 16k.

---

^ 6.13 **SET/PROJEXTSTATE/section/key/value**

See SetProjExtState() API -- section, key, value should be urlencoded

---

^ 6.13 **GET/PROJEXTSTATE/section/key**

Returns:
PROJEXTSTATE \t section \t key \t string
See: GetProjExtState() API
String will have newlines/tabs/backslashes encoded as \\n, \\t and \\. Length is limited to around 16k.

---

^ 6.13 **GET/EXTSTATE/section/key**

Returns:
EXTSTATE \t section \t key \t string
See GetExtState() API
String will have newlines/tabs/backslashes encoded as \\n, \\t and \\

---

^ 6.13 **SET/EXTSTATE/section/key/value**

See SetExtState() API (persist=false) section, key, value should be urlencoded

---

^ 6.13 SET/EXTSTATEPERSIST/section/key/value

See SetExtState() API (persist=true) section, key, value should be urlencoded

---

^ 6.13 GET/command_id

command_id can be numeric or _123456789abcdef... registered ID.
Returns:
CMDSTATE \t command_id \t state

state>0 for on, 0=off, -1=no state

---

^ 6.13 OSC/oscstring[:value]

Sends an OSC message through the default processing (Default.ReaperOSC) and MIDI-learn/action mappings. oscstring:value will be urldecoded.

---

^ 6.13 MARKER

Returns a list of all markers, in the format of:

MARKER_LIST
MARKER \t name \t ID \t position [\t color]
...
MARKER_LIST_END

color is in the form of 0xaarrggbb, nonzero if a custom color set

---

^ 6.13 REGION

Returns a list of all markers or regions, in the format of:

REGION_LIST
REGION \t name \t ID \t start-position \t end-position [\t color]
...
REGION_LIST_END
