LuaJack Reference Manual

Stefano Trettel

Version 0.2, 2015-10-29

Table of Contents

[Lua logo] | *powered-by-lua.gif*

Preface

This is the reference manual of **LuaJack**, which is a **Lua** binding library for the **JACK Audio Connection Kit**. [1: This manual is written in AsciiDoc, rendered with AsciiDoctor and a CSS from the AsciiDoctor Stylesheet Factory. The PDF version is produced with AsciiDoctor-Pdf.]

It is assumed that the reader is familiar with both JACK and the Lua programming language.

For convenience of reference, this document contains external (deep) links to the Lua Reference Manual and to the JACK API documentation.

Getting and installing

The **official repository** of LuaJack is on GitHub at the following link: **https://github.com/stetre/luajack** .

LuaJack runs on GNU/Linux and requires **Lua** version 5.3 or greater, and **JACK** (aligned to JACK API v0.124.1).

By now, it has been used only on GNU/Linux Fedora 21 with JACK2 v1.9.10. It should compile and run also on any other GNU/Linux distribution and with JACK1 instead of JACK2, but this has not been tested.

To install LuaJack, download the latest release and do the following:

```
# ... download luajack-0.1.tar.gz ...
[ ]$ tar -zxpvf luajack-0.1.tar.gz
[ ]$ cd luajack-0.1
[luajack-0.1]$ make
[luajack-0.1]$ make check
[luajack-0.1]$ sudo make install
```

The *$make check* command shows you what will be installed and where (please read its output before executing *$make install*). By default, LuaJack installs its components in subdirectories of `/usr/local/` (and creates such directories, if needed). This behaviour can be changed by defining PREFIX with the desired alternative base installation directory. For example, this will install the components in `/home/joe/local`:

```
[luajack-0.1]$ make
[luajack-0.1]$ make install PREFIX=/home/joe/local
```

Module organization

The LuaJack module is loaded using Lua’s require() and returns a table containing the functions it provides (as usual with Lua modules). This manual assumes that such table is named **jack**, i.e. that it is loaded with:

```lua
jack = require("luajack")
```

but nothing forbids the use of a different name.

Examples

A few examples can be found in the **examples/** directory of the release package (some of them are LuaJack versions of the examples that come with the original JACK1 and JACK2 software).

License

LuaJack is released under the **MIT/X11 license** (same as Lua, and with the same only requirement to give proper credits to the original author). The copyright notice is in the LICENSE file in the base directory of the official repository on GitHub.

Introduction

An example application

A LuaJack application, apart from being written in Lua instead of C, looks very similar to a standard JACK application.

An example is shown below: it creates a JACK client with a couple of ports, then it registers some callbacks, activates the client and eventually sleeps while waiting for JACK to call back.

IMPORTANT LuaJack applications must implement their main loop using the jack.sleep() function.

```
jack = require("luajack")
-- This is the 'main script', executed in the 'main context'.

-- Open a JACK client:
c = jack.client_open("myclient")

-- Create two ports:
p_in = jack.input_audio_port("port_in")
p_out = jack.output_audio_port("port_out")

-- Load the process chunk:
jack.process_load(c, [[
c, p_in, p_out = table.unpack(arg)

function process(nframes)
   -- Audio processing happens here (in this example, we just
   -- copy the samples from the input port to the output port).
   jack.get_buffer(p_in)
   jack.get_buffer(p_out)
   jack.copy(p_out, p_in)
end

-- Register the (rt) process callback:
jack.process_callback(c, process)
]], c, p_in, p_out)

-- Register a non-rt callback:
jack.shutdown_callback(c, function() error("shutdown from server") end)

-- Activate the client:
jack.activate(c)

-- Sleep, waiting for JACK to call back:
jack.sleep()
```

The main deviation from an hypothetical standard implementation of the same example is in the use of the jack.process_load() function to 'load' the real-time part of the application. This difference should be (hopefully) clear after reading the next subsection.

LuaJack contexts

In this document we use the word **'context'** to refer to the combination of the pthread and the Lua state a chunk of Lua code is executed in.

Since LuaJack relies on libjack, which is a multithreaded library, it has more than one context where to execute the Lua chunks of an application. More precisely, there are (basically) three types of contexts in LuaJack:

1. The **main context**, where the main script is executed. Clients, ports, etc. are created here, and the main loop and the (Lua) non real-time callbacks are executed here, too. The main context is composed of the main pthread and the main Lua state, i.e. the state created by the Lua interpreter. There is one main context in a LuaJack application, and it is shared by all the clients the application creates.
2. The **process context**, where the real-time callbacks are executed. It is composed of the (possibly) real-time pthread created by JACK for processing audio, and a dedicated Lua state. Each client has its own process context, which it creates with the jack.process_load( ) function, passing it the Lua code to be executed there (the **'process chunk'**).
3. The **thread context**, where a client thread is executed. It is composed of the pthread created for it by JACK and a dedicated Lua state. Each client thread has its own thread context, which is created with the client thread itself by means of the jack.thread_load( ) function, passing it the Lua code to be executed there (the **'thread chunk'**).

All the above **Lua states are unrelated** (that is, independent) and thus the corresponding contexts are insulated one from each other (they have separated namespaces and don't share memory) as far as Lua code is concerned.

Communication between different contexts is achievable by means of parameter passing (when creating a context), and via the lock-free ringbuffers provided by JACK. As another mechanism of communication, client threads may also wait in their context to be signalled from other contexts (each client thread has a pthread condition variable associated with it for this purpose).

The LuaJack module is to be explicitly loaded (i.e. *jack = require("luajack")*) only in the main script. There is no need to load it in process chunks and thread chunks because they are executed in Lua states where the LuaJack module is automatically pre-loaded in a global table named **'jack'**. This pre-loaded module is actually a limited version of the full LuaJack module, in that it contains only the subset of functions that can be directly used in the process or thread context (the availability of functions in different contexts is indicated in the summary tables).

LuaJack functions are described in the following sections, whose organization somehow reflects the organization of the original JACK API documentation (on purpose).

Clients

- **jack.client_open** (*name* [, *options*])
  → *client*

  Creates a JACK client with the passed *name* and returns an opaque reference for subsequent operations. If the *options* parameter is passed, it must be a table containing zero or more of the following elements:

  - *options.use_exact_name* (boolean): if *true*, do not automatically assign a new name if *name* is already in use;
  - *options.no_start_server* (boolean): if *true*, do not start the JACK server if not already running;
  - *options.server_name* (string): select the JACK server with this name;
  - *options.session_id* (string): pass this string as SessionID Token.

- **jack.client_name_size** ( )
  → *maxsize*

  Returns the maximum allowed size for JACK clients names.

- **jack.client_close** (*client*)

  Closes a client previously opened with jack.client().

NOTE Clients are automatically closed by LuaJack when the application exits.

- **jack.client_name** (*client*)
  → *name*

  Returns the client's name.

- **jack.client_uuid** (*client*)
  → *uuid*

  Returns the client's UUID.

- **jack.client_uuid_to_name** (*client*, *uuid*)
  → *name*

  Returns the name of the client whose UUID is *uuid*, or *nil* if no such client exists.

- **jack.client_name_to_uuid** (*client*, *name*)
  → *uuid*

  Returns the UUID of the client whose name is *name*, or *nil* if no such client exists.

- **jack.activate** (*client*)

  Activates *client*.

- **jack.deactivate** (*client*)

  Deactivates *client*.

- **jack.is_realtime** (*client*)
  → *boolean*

  Returns *true* if JACK is running realtime, *false* otherwise.

- **jack.cpu_load** (*client*)
  → *cpuload*

  Returns the current CPU load estimated by JACK.

- **jack.sample_rate** (*client*)
  → *sample_rate*

  Returns the sample rate of the JACK system, as set by the user when the server was started.

- **jack.set_buffer_size** (*client*, *nframes*)

Changes the buffer size passed to the process callback to *nframes*.

- **jack.buffer_size** (*client*)
  → *nframes*

  Returns the current maximum buffer size that will ever be passed to the process callback (to be used only before the client is activated).

- **jack.freewheel** (*client*, *onoff*)

  Starts/stops JACK's '*freewheel*' mode. The *onoff* argument is a string and may be *'on'* (to start) or *'off'* (to stop).

Real-time callbacks

Real-time callbacks are executed in the process context. This context is to be created by the main script using jack.process_load() or jack.process_loadfile() before the client is activated. The loaded Lua chunk is expected to define the callbacks and register them using the functions described in this section, which are available only in the process context.

NOTE

> The process chunk is actually loaded in an hybrid context, in that its main part is executed in the Lua state of the process context while still in the non-rt pthread of the main context (the rt-pthread will be created by JACK later and does not yet exist at the time the process chunk is loaded). This means that it can perform non real-time safe operations, e.g. pre-allocate large tables for samples storage. On the other side, the rt-callbacks that the chunk registers will be executed in the pure process context, and thus they should be real-time safe.
>
> Note also that Lua is a garbage-collected language with automatic memory management. This implies that the execution of Lua callbacks potentially involves calls to functions which are not real-time safe (i.e. malloc() and friends) and that, if not adeguately controlled, the GC may kick in at any time in the real-time pthread.
>
> To mitigate the GC problem, LuaJack disables garbage collection in the Lua state of the process context, and automatically executes a basic garbage-collection step (with size=0) at the end of each rt-callback. This should suffice as long as the rt-callbacks do not allocate lots of memory (which would be a bad idea…).
>
> Calls to malloc() and friends should not be a problem as long as they do not result in page faults. This can be accomplished first of all by avoiding the creation of large temporary objects (e.g. tables) in the rt-callbacks, and by pre-allocating them in the main part of the process chunk, instead.
>
> As an additional measure one could use, via LD_PRELOAD, a custom malloc() implementation that does not return memory to the system (for example, ltalloc), and allocate+release 'enough' memory at the beginning of the program execution so that the allocator can serve subsequent memory requests without asking additional memory to the system.
>
> *(The real-time aspects of LuaJack are, however, still under study so take the above considerations with salt).*

- **jack.process_load** (*client*, *chunk*, …)

  This function is to be used in the main context to create the process context for *client*.

The *chunk* argument is a string containing the Lua code to be executed in the process context. Additional arguments in the ... variadic part, if any, are passed as arguments to the chunk and may be of the following types only: *nil*, *boolean*, *number*, and *string* (references to LuaJack objects are allowed, since their type is *number*).

A limited LuaJack module is automatically pre-loaded in the process context, with the subset of JACK functionalities that can be accessed directly by the process chunk.

- **jack.process_loadfile** (*client*, *filename*, ...)

  Same as jack.process_load(), with the only difference that it loads the chunk from the file specified by *filename*. The file is searched for using the same mechanism used by Lua's require() to search for modules.

- **jack.process_callback** (*client*, *func*)

  Registers *func* as 'process' callback (*func* must be realtime safe).

  The callback is executed as ***func(nframes)***, where *nframes* is the number of frames to process.

- **jack.buffer_size_callback** (*client*, *func*)

  Registers *func* as callback for the 'buffer size change' event.

  The callback is executed as ***func(nframes)***, where *nframes* is the new size of the JACK engine buffer.

- **jack.sync_callback** (*client*, *func*)

  Registers *client* as a slow-sync client and *func* as its *sync_callback* (*func* must be realtime safe and is invoked just before the *process* callback).

  The callback is executed as ***func(state, position)***, where the additional arguments are the current transport *state* and *position*. It is expected to return *true* if the client is ready to roll, or *false* (or *nil*) otherwise.

- **jack.timebase_callback** (*client*, *func* [, *conditional*])

Registers *client* as the timebase master and *func* as its *timebase callback* (*func* must be realtime safe because and is invoked just after the *process* callback). If another client is already registered as timebase master, this client will take over unless the optional *conditional* argument (a boolean) is passed with the value *true*.

The callback is executed as ***func(state, nframes, position, new_position)***, where the additional arguments are the current transport *state*, the current transport *position*, and a boolean *new_position* having the value *true* for a newly requested position (which the *client* should provide using jack.transport_reposition() and jack.transport_locate()).

- **jack.release_timebase** (*client*)

  Used by the client previously registered as the timebase master to withdraw that role and unregister the timebase callback.

Non real-time callbacks

The functions listed here are available only in the main context and can be used to register (Lua) callbacks for JACK events other than those that are handled in the real-time audio processing thread.

The registered callbacks are also executed by LuaJack in the main context. Notice that each callback receives the affected *client* as its first argument (this is because the main context is shared by all the clients created by the LuaJack application).

> **NOTE**
>
> All the non real-time Lua callbacks described here are executed by LuaJack in the main pthread. JACK itself may execute its C callbacks in different pthreads; this is a libjack implementation choice and may differ across different implementations as well as platforms. When JACK executes a C callback, LuaJack saves the event associated with it and promptly returns. The event is then translated in a Lua callback in the main pthread.

- **jack.shutdown_callback** (*client*, *func*)

  Registers *func* as callback for the 'shutdown' event.

  The callback is executed as ***func(client, code, reason)***, where *code* is a string version of the JackStatus flags and *reason* is a string describing the shutdown reason. The callback need not be implemented as a signal handler (it may print, release resources etc.).

- **jack.freewheel_callback** (*client*, *func*)

  Registers *func* as callback for the 'starting or stopping freewheel mode' event.

  The callback is executed as ***func(client, operation)***, where *operation* is a string that may be either *'starting'* or *'stopping'*.

- **jack.sample_rate_callback** (*client*, *func*)

  Registers *func* as callback for the 'sample rate change' event.

  The callback is executed as ***func(client, nframes)***, where *nframes* is the new sample rate.

- **jack.client_registration_callback** (*client*, *func*)

Registers *func* as callback for the 'client (de)registration' event.

The callback is executed as ***func(client, name, operation)***, where *name* is the name of the affected client and *operation* is a string that may be either *'registered'* or *'unregistered'*.

- **jack.port_registration_callback** (*client*, *func*)

  Registers *func* as callback for the 'port (de)registration' event.

  The callback is executed as ***func(client, portname, operation)***, where *portname* is the (full) name of the affected port and *operation* is a string that may be either *'registered'* or *'unregistered'*.

- **jack.port_rename_callback** (*client*, *func*)

  Registers *func* as callback for the 'port rename' event.

  The callback is executed as ***func(client, portname, newname)***, where *portname* is the old (full) name of the affected port and *newname* is its newly assigned name.

- **jack.port_connect_callback** (*client*, *func*)

  Registers *func* as callback for the 'ports connect or disconnect' event.

  The callback is executed as ***func(client, srcname, dstname, operation)***, where *srcname* and *dstname* are the (full) names of the affected ports, and *operation* is a string that may be either *'connected'* or *'disconnected'*.

- **jack.graph_order_callback** (*client*, *func*)

  Registers *func* as callback for the 'graph reorder' event.

  The callback is executed as ***func(client)***.

- **jack.xrun_callback** (*client*, *func*)

  Registers *func* as callback for the 'xrun' event.

The callback is executed as ***func(client)***.

- **jack.latency_callback** (*client*, *func*)

  Registers *func* as callback for the 'latency recomputations needed' event.

  The callback is executed as ***func(client, mode)***, where *mode* is a string that may be either *'capture'* or *'playback'*.

- **jack.session_callback** (*client*, *func*)

  Registers *func* as callback for the 'session notification' event.

  The callback is executed as ***command, flag1, flag2* = *func(client, type, path, uuid)***, where:

  - *type* (a string) is the type of this session event and may be one of *'save'*, *'save_and_quit'* or *'save_template'*;
  - *path* (a string) is the session directory path;
  - *uuid* (a string) is the client UUID which must be used when opening the client.

    The callback is expected to return the following values, which are used by LuaJack to reply to the event:
  - *command*: the command line (a string) needed to restore the client;
  - *flag1*, *flag2*: optional session flags (*'save_error'* and/or *'need_terminal'*).

Client threads

- **jack.thread_load** (*client*, *chunk*, ...)
  → *thread*

  Creates a client thread with its dedicated thread context and returns a reference for subsequent operations. The *chunk* argument is a string containing the Lua code to be executed in the thread. Additional arguments in the ... variadic part, if any, are passed as arguments to the chunk and may be of the following types only: *nil*, *boolean*, *number*, *string* (references to LuaJack objects are allowed, since their type is *number*).

  A limited LuaJack module is automatically pre-loaded in the thread context, with the subset of JACK functionalities that can be accessed directly by the thread chunk.

- **jack.thread_loadfile** (*client*, *filename*, ...)
  → *thread*

  Same as jack.thread_load(), with the only difference that it loads the chunk from the file specified by *filename*. The file is searched for using the same mechanism used by Lua's require() to search for modules.

- **jack.self** ( )
  → *client*, *thread*

  Returns the client and thread references of the calling thread. This function is available only to thread chunks (i.e. in thread contexts).

- **jack.signal** (*client*, *thread*)

  Signals the pthread condition variable associated with *thread*.

- **jack.wait** ( )

  Waits on the pthread condition variable associated with the calling thread. This function is available only to thread chunks (i.e. in thread contexts).

- **jack.real_time_priority** (*client*)

→ *priority*

If JACK is running with realtime priority, returns the priority that any thread created by JACK will run at. Otherwise it returns *nil*.

- **jack.max_real_time_priority** (*client*)
  → *priority*

  If JACK is running with realtime priority, returns the maximum priority that a realtime client thread should use. Otherwise it returns *nil*.

- **jack.acquire_real_time_scheduling** (*priority*)

  Enables realtime scheduling, with the specified *priority*, for the calling thread.

- **jack.drop_real_time_scheduling** ( )

  Drops realtime scheduling for the calling thread.

Ports

Instead of a single port_register() function, LuaJack provides a few functions to create ports depending on the desired direction (*input* or *output*) and port type (default *audio*, default *midi*, or *custom*, i.e. not built-in). These port-creating functions are listed below, followed by the functions for port handling.

Notice that most of the port-handling functions are available in two versions: the version named **jack.port_xxx** acts on a port by reference, is usually faster because it does not involve querying the server to resolve the port name, but can be used only in the application that created the port; the version named **jack.nport_xxx**, on the other side, is slower because it acts on the port by its name, but it can be used also on ports not created by the invoking application.

- **jack.input_audio_port** (*client* , *name* [, *options*])
  → *port*

  Creates an input port of the default audio type for *client* and returns a reference for subsequent operations.

  The mandatory *name* argument (a string) is the short name to be assigned to the port, while the *options* parameter, if present, must be a table with zero or more of the following elements:

  - *options.is_physical* (boolean): JackPortIsPhysical flag;
  - *options.can_monitor* (boolean): JackPortCanMonitor flag;
  - *options.is_terminal* (boolean): JackPortIsTerminal flag.

- **jack.output_audio_port** (*client* , *name* [, *options*])
  → *port*

  Creates an output port of the default audio type for *client* and returns a reference for subsequent operations.
  The meaning of the arguments is the same as for jack.input_audio_port().

- **jack.input_midi_port** (*client* , *name* [, *options*])
  → *port*

  Creates an input port of the default MIDI type for *client* and returns a reference for subsequent operations.
  The meaning of the arguments is the same as for jack.input_audio_port().

- **jack.output_midi_port** (*client* , *name* [, *options*])
  → *port*

  Creates an output port of the default MIDI type for *client* and returns a reference for subsequent operations.
  The meaning of the arguments is the same as for jack.input_audio_port().

- **jack.input_custom_port** (*client* , *name* [, *options*])
  → *port*

  Not available yet.

- **jack.output_custom_port** (*client* , *name* [, *options*])
  → *port*

  Not available yet.

- **jack.port_name_size** ( )
  → *maxsize*

  Returns the maximum allowed size for JACK port names.

- **jack.port_type_size** ( )
  → *maxsize*

  Returns the maximum allowed size for JACK port types.

- **jack.port_unregister** (*port*)

  Unregisters and deletes *port*.

NOTE A port is automatically unregistered by LuaJack when the client that created it is closed.

- **jack.port_connect** (*port*, *portname2*)
- **jack.nport_connect** (*client*, *portname1*, *portname2*) alias **jack.connect**

Connects two ports. One of the two ports must be an output port (source of the connection) and the other must be an input port (destination). The two ports can be passed in any order.

- **jack.port_disconnect** (*port* [, *portname2*])
- **jack.nport_disconnect** (*client*, *portname1* [, *portname2*]) alias **jack.disconnect**

Disconnects two ports. If *portname2* is *nil*, the first port is disconnected from all the ports it is connected to.

- **jack.port_name** (*port*)
→ *fullname*

Returns the port’s full name.

- **jack.port_short_name** (*port*)
→ *shortname*

Returns the port’s short name.

- **jack.port_set_name** (*port*, *newname*)

Changes the port’s short name to *newname*.

- **jack.port_flags** (*port*)
- **jack.nport_flags** (*client*, *portname*)
→ *flags*, *table*

Returns the port’s flags, in two equivalent formats:

  - *flags* : a string concatenating the flags that apply to the port, and
  - *table*: a table where *table.<flag>* is *true* if <*flag*> applies to the port, or *nil* otherwise.
(<*flag*> = *'input'* | *'output'* | *'audio'* | *'midi'* | *'custom'* | *'is_physical'* | *'can_monitor'* | *'is_terminal'*)

- **jack.port_uuid** (*port*)
- **jack.nport_uuid** (*client*, *portname*)

→ *uuid*

Returns the port’s UUID (a string), or *nil* if not found.

- **jack.port_type** (*port*)
- **jack.nport_type** (*client*, *portname*)
  → *type*

Returns the port’s type (a string), or *nil* if not found.

- **jack.nport_exists** (*client*, *portname*)
  → *boolean*

Returns *true* if the port with the full named *portname* exists, otherwise it returns *false*.

- **jack.port_is_mine** (*client*, *port*)
- **jack.nport_is_mine** (*client*, *portname*)
  → *boolean*

Returns *true* if the port is owned by the invoking *client*, otherwise it returns *false*.

- **jack.port_set_alias** (*port*, *alias*)
- **jack.nport_set_alias** (*client*, *portname*, *alias*)

Sets *alias* as an alias for the port.

- **jack.port_unset_alias** (*port*, *alias*)
- **jack.nport_unset_alias** (*client*, *portname*, *alias*)

Unsets *alias* as an alias for the port.

- **jack.port_aliases** (*port*)
- **jack.nport_aliases** (*client*, *portname*)
  → *alias1*, *alias2*

Returns the aliases for the port (if it has any).

- **jack.port_connections** (*port* [, *list*])
- **jack.nport_connections** (*client*, *portname* [, *list*])
  → *N*, { *portname1, …, portnameN* }

  Returns the number of connections the port is involved in. If *list* = *true*, it returns also a table containing the full names of the ports the port is connected to.

- **jack.port_connected_to** (*port*, *portname2*)
- **jack.nport_connected_to** (*client*, *portname1*, *portname2*)
  → *boolean*

  Returns *true* if the two ports are connected, *false* otherwise.

- **jack.port_monitor** (*port*, *onoff*)
- **jack.nport_monitor** (*client*, *portname*, *onoff*)

  Turns input monitoring on or off for the port (implemented with jack_port_ensure_monitor()). The *onoff* argument is a string and may be *'on'* or *'off*.

- **jack.port_monitoring** (*port*)
- **jack.nport_monitoring** (*client*, *portname*)
  → *boolean*

  Returns *true* if input monitoring has been requested for the port, *false* otherwise or if no port with this name was found.

- **jack.get_ports** (*client* [, *filter*])
  → *{ portname1, …, portnameN }*

  Returns a list (table) of full port names. If *filter* is *nil*, all the ports are listed, otherwise the ports are selected according to the *filter* parameter, which must be a table containing zero or more of the following optional elements:

  - *filter.name_pattern*: a regular expression (string) used to select ports by name;
  - *filter.type_pattern*: a regular expression (string) used to select ports by type;

- *filter.direction*: a string that may be *'input'* or *'output'*, to select input ports only or output ports only, respectively;
- *filter.is_physical* (boolean): if *true*, list only physical ports;
- *filter.can_monitor* (boolean): if *true*, list only ports that can monitor;
- *filter.is_terminal* (boolean): if *true*, list only terminal ports.

Latency

Refer to 'Managing and determining latency' in the JACK documentation for details on how latencies should be managed.

- **jack.latency_range** (*port*, *mode*)
  → *min*, *max*

  Gets the minimum and maximum latencies (in frames) defined by *mode* for the port. The *mode* parameter (a string) may be *'capture'* or *'playback'*.

- **jack.set_latency_range** (*port*, *mode*, *min*, *max*)

  Sets the minimum and maximum latencies (in frames) defined by *mode* for *port*. The *mode* parameter (a string) may be *'capture'* or *playback*. This function should only be used inside a latency callback.

- **jack.recompute_total_latencies** (*client*)

  Requests a complete recomputation of all port latencies.

Time

- **jack.time** ( )
  → *useconds*

  Returns JACK's current system time in microseconds.

- **jack.frame** (*client*) alias **jack.frame_time**
  → *frameno*

  Returns the estimated current time in frames (to be used outside the process callback).

- **jack.since** (*useconds*)
  → *elapsed*

  Returns the time, in microseconds, elapsed since JACK's system time *useconds*.

- **jack.since_frame** (*client*, *frameno*)
  → *nframes*

  Returns the number of frames elapsed since the estimated time *frameno*, in frames (to be used outside the process callback).

- **jack.frames_to_time** (*client*, *nframes*)
  → *useconds*

  Returns the estimated time in microseconds of the specified time in frames.

- **jack.time_to_frames** (*client*, *useconds*)
  → *nframes*

  Returns the estimated time in frames of the specified time in microseconds.

- **jack.frames_since_cycle_start** (*client*)
  → *nframes*

Returns the estimated time in frames that has passed since the JACK server began the current process cycle.

- **jack.last_frame_time** (*client*)
  → *nframes*

  Returns the time, in frames, at the start of the current process cycle. Available in the process callback only, to be used to interpret timestamps generated with jack.frame() in other threads.

- **jack.cycle_times** (*client*)
  → *current_frames*, *current_usecs*, *next_usecs*, *period_usecs*

  Available in the process callback only, returns internal cycle timing information (refer to jack_get_cycle_time() in the JACK documentation for more details).

Transport and timebase

Refer to 'JACK Transport Design' in the JACK documentation for details on the transport mechanism.

- **jack.current_transport_frame** (*client*)
  → *frameno*

  Returns an estimate of the current transport frame.

- **jack.transport_state** (*client*)
  → *state*

  Returns the current transport *state* (a string that may have one of the following values: *'starting'*, *'rolling'*, or *'stopping'*).

- **jack.transport_query** (*client*)
  → *state*, *position*

  Returns the current transport *state* (see jack.transport_state()) and the current transport *position*.

  The transport position is a Lua-table representation of the jack_position_t struct. Its elements have the same name and meaning as the fields of the struct, except for the *valid* field, which in the table representation is not used (optional elements which are not present are set to *nil*, i.e. not set at all).

- **jack.transport_start** (*client*)

  Starts the JACK transport rolling.

- **jack.transport_stop** (*client*)

  Stops the JACK transport rolling.

- **jack.transport_locate** (*client*, *frameno*)

  Repositions the JACK transport to the frame number *frameno* (realtime safe, may be used in the timebase callback).

- **jack.transport_reposition** (*client*, *position*)

  Request a new transport *position* (realtime safe, may be used in the timebase callback).

- **jack.set_sync_timeout** (*client*, *timeout*)

  Sets the *timeout* value (in microseconds) for slow-sync clients.

Reading and writing audio data

The functions described in this section are available only in the process callback, and allow to read and write samples from/to ports of the default audio type.

For this type of ports, a sample is represented by a Lua number.

- **jack.get_buffer** (*port*)
  → *nframes*

  Retrieves the port buffer and initializes the buffer's *current position* to the first sample. Returns the buffer size in samples (which is the same value as the *nframes* parameter passed to the process callback).

  This function must be invoked in the process callback, at each process cycle, before any other operation on the buffer.

- **jack.seek** (*port* [, *position*])
  → *position*, *available*

  Returns the current *position* in the port's buffer and the number of available samples from the current position to the end of the buffer. If *position* is passed as argument, the function sets the current position to that value.

- **jack.read** (*port* [, *count*])
  → *sample1*, ...

  Returns up to *count* samples from the (input) port buffer. Samples are read starting from the current position, which is then advanced by the number of returned samples. If *count* exceeds the available samples, this function returns the available samples only. If no samples are available, it returns *nil*. If *count* is not passed, it defaults to *'all the remaining samples'*.

- **jack.write** (*port* [, *sample1* , ...])
  → *count*

  Writes the passed samples to the (output) port's buffer and returns the *count* of written samples. Samples are written starting from the current position, which is then advanced by *count*. If the number of passed samples exceeds the available space in the buffer, the exceeding samples are discarded.

- **jack.clear** (*port* [, *count*])
  → *count*

  Same as jack.write(), with the difference that it writes up to *count* samples to zero. If *count* is not passed, it defaults to *'all the remaining samples'*.

- **jack.copy** (*dstport*, *srcport* [, *count*])
  → *count*

  Copies up to *count* samples from the buffer of the input port *srcport* to the buffer of the output port *dstport* and returns the number of copied samples. This function uses the current positions of both ports, and advances them by the number of copied samples. If *count* is not passed, it defaults to *maxcount*, which is defined as the minimum between the remaining samples in the input port's buffer and the space left in the output port's buffer. If count is passed but it is greater than *maxcount*, only *maxcount* samples are copied.

Reading and writing MIDI data

The functions described in this section are available only in the process callback, and allow to read and write MIDI events from/to ports of the default MIDI type.

A MIDI event is represented in LuaJack as a couple of values (*time*, *data*), where:

> *time* is the frame number of the event, relative to the current period, and
>
> *data* is a binary string of length >= 1, containing the MIDI message (<*status*>…).

The *data* value may be handled with Lua's native string.pack() and string.unpack() functions, or with the utilities provided in the luajack/midi.lua module.

- **jack.get_buffer** (*port*)
  → *eventcount*, *lostcount* (if port is input)
  → *space* (if port is output)

  Retrieves the port buffer. If *port* is an input port, this function returns the number of events in the buffer (*eventcount*) and the number of lost events (*lostcount*). It also initializes the current index to *0*, i.e. to the first event in the buffer. If *port* is an output port, the function returns the *space* available in the buffer for writing MIDI event data.

  This function must be invoked in the process callback, at each process cycle, before any other operation on the buffer.

- **jack.seek** (*port* [, *index*])
  → *index*, *available*

  Returns the current *index* in the (input) port buffer and the number of *available* events from the current index to the end of the buffer. If *index* is passed as argument, sets the current index to that value.

NOTE Only input MIDI ports are seekable (output MIDI ports are not).

- **jack.read** (*port* [, *index*])
  → *time*, *data*

  Returns the MIDI event from the (input) port's buffer at its current index, which it then advances by *1*. If an *index* is passed as argument, this function sets the current index to that value before reading the event. If the current index exceeds *eventcount* - *1*, the function returns *nil*.

- **jack.write** (*port*, *time*, *data*)
  → *space*

  Writes a MIDI event to the (output) port's buffer and returns the *space* available in the buffer for the next event data. If there is not enough space to write the event, this function returns *nil*.

- **jack.copy** (*dstport*, *srcport* [, *count*])
  → *count*

  Copies up to *count* MIDI events from the buffer of the input port *srcport* to the buffer of the output port *dstport*, and returns the number of copied events. Events are copied starting from the current index of *srcport*, which is then advanced by the number of copied events. If *count* is not passed, this function attempts to copy all of the remaining events in the input port's buffer that fit in the output port's buffer.

Statistics

- **jack.max_delayed_usecs** (*client*)
  → *delay*

  Returns the maximum delay reported by the backend since startup or reset.

- **jack.xrun_delayed_usecs** (*client*)
  → *delay*

  Returns the delay in microseconds due to the most recent xrun occurrence.

- **jack.reset_max_delayed_usecs** (*client*)

  Resets the maximum delay counter.

Session API

Session clients are expected to use jack.session_callback() to register a session callback in order to listen to events sent by the session manage, which instead uses the functions described below in this section.

- **jack.session_notify** (*client*, *target*, *type*, *path*)
  → *{ reply1, ..., replyN }*

  Sends an event to clients that registered a session callback. The *target* argument (a string) is the name of the target client, or *nil* for 'all'. The event *type* must be one amongst *'save'*, *'save_and_quit'* and *'save_template'*. The *path* argument is the session directory path.

  The function returns a table with the replies from the session clients. Each entry of the table is a subtable containing the following elements:

  - *reply.uuid* (a string): the client's UUID;
  - *reply.name* (a string): the client's name;
  - *reply.command* (a string): the command line needed to restore the client;
  - *reply.save_error* (*true* or *nil*): session flag;
  - *reply.need_terminal* (*true* or *nil*): session flag;

- **jack.has_session_callback** (*client*, *clientname*)
  → *boolean*

  Returns *true* if the client named *clientname* has registered a session callback, *false* otherwise.

- **jack.reserve_client_name** (*client*, *clientname*)

  Reserves the client name *clientname* and associates it with the UUID *uuid*.

Ringbuffers

LuaJack wraps the lock-free ringbuffers provided by JACK for inter-thread communication, giving them a message-oriented nature and adding means to use them in conjunction with sockets.

In LuaJack, data is written to and read from ringbuffers in form of tagged **messages**, each composed of a *tag* (an integer number) optionally followed by *data* (a Lua string). The tag can be used to represent the 'type' of the message or any other information (LuaJack by itself does not interpret it by any means).

- **jack.ringbuffer** (*client*, *size* [, *mlock* [, *usepipe*]])
  → *rbuf*

  Creates a ringbuffer of the specified *size* (in bytes) and returns a ringbuffer reference for subsequent operations. The returned reference is an integer which may be passed as argument to thread scripts. If *mlock=true*, the buffer is locked in memory. If *usepipe=true*, a pipe is associated with the ringbuffer allowing its 'read' end to be turned in an object compatible with sockect.select() and thus to be used in conjunction with sockets or other fd-based communication mechanisms (of course, you don’t want to do this in an audio processing realtime thread…).

  This function is only available in the main context and must be used before the client is activated.

- **jack.ringbuffer_write** (*rbuf*, *tag* [, *data*])
  → *ok*

  Write a message to the ringbuffer *rbuf*. Returns *true* on success, or *false* if there was not enough space for the message.

- **jack.ringbuffer_read** (*rbuf*)
  → *tag*, *data*

  Read a message from the ringbuffer *rbuf*. Returns *tag=nil* if there are no messages available. If the is message is composed of the *tag* only, then *data* is returned as an empty string (so that *data:len=0*).

- **jack.ringbuffer_reset** (*rbuf*)

  Resets the ringbuffer *rbuf*.

- **jack.ringbuffer_peek** (*rbuf*)
  → *ok*

  Returns *true* if there is a message available to be read from the ringbuffer *rbuf*, otherwise it returns *false*.

- **jack.ringbuffer_getfd** (*rbuf*)
  → *fd*

  Returns the file descriptor of the pipe associated with the ringbuffer *rbuf*, or *nil* if it was created without pipe.

Selectable ringbuffers

@@ TODO

JACK server control API

Not implemented yet.

Non-callback API

Not supported by LuaJack.

Internal clients API

Not supported by LuaJack.

Metadata API

Not implemented yet.

Additional functions

- **jack.sleep** ([*seconds*])

  Sleeps for the specified number of *seconds* (if *seconds* is negative or *nil*, sleeps forevever). This function must be used to implement the main loop in the main context, and can optionally be used in thread contexts as well.

- **jack.verbose** (*onoff*)

  If *onoff*='on', enables the LuaJack verbose mode. If *onoff*='off', it disables it. By default, the verbose mode is disabled.

MIDI utilities

The **luajack.midi** module is a LuaJack submodule that provides convenient utilities for dealing with MIDI messages in Lua.

Although it is part of the LuaJack package, **luajack.midi** is actually an independent module that can be used also in non-LuaJack applications. Since it is a plain Lua script, it can be easily customized or replaced (its source is in the *luajack/midi.lua* file in the LuaJack package).

The module is to be loaded in the usual Lua way, e.g.:

```
midi = require("luajack.midi")
```

and returns a table containing the utilities described in the subsections that follow (it is assumed here that the table is named **'midi'** as in the above example, but a different name could be used as well).

Encoding MIDI messages

Each of the following routines returns a binary string containing an encoded MIDI message (starting from the MIDI status byte) with the passed parameters.

- **midi.note_off** (*channel*, *key* [, *velocity*])
- **midi.note_on** (*channel*, *key* [, *velocity*])
  → *msg*

  Parameters: *channel* = 1-16, *key* = 0-127, *velocity* = 0-127 (default=64).

- **midi.aftertouch** (*channel*, *key*, *pressure*) a.k.a. 'polyphonic key pressure'
  → *msg*

  Parameters: *channel* = 1-16, *key* = 0-127, *pressure* = 0-127.

- **midi.control_change** (*channel*, *control* [, *value*])
  → *msg*

  Parameters: *channel* = 1-16, *control* = 0-127 or controller name, *value* = 0-127 or *'on'*|*'off'* (default=0='off').
  See the table in the following section for controller names.

- **midi.program_change** (*channel*, *number*)
  → *msg*

  Parameters: *channel* = 1-16, *number* = 1-128.

- **midi.channel_pressure** (*channel*, *pressure*)
  → *msg*

  Parameters: *channel* = 1-16, *pressure* = 0-127.

- **midi.pitch_wheel** (*channel*, *value*) a.k.a. 'pitch bend change'
  → *msg*

  Parameters: *channel* = 1-16, *value* = 0-16383 (0x3fff).

- **midi.mtc_quarter_frame** (*nnn*, *dddd*) a.k.a. 'time code quarter frame'
  → *msg*

  Parameters: *nnn* = 0-7 (MTC message type), *dddd* = 0-15 (MTC message value).

- **midi.song_position** (*value*)
  → *msg*

  Parameters: *value* = 0-16383 (0x3fff).

- **midi.song_select** (*number*)
  → *msg*

  Parameters: *number* = 1-128.

- **midi.tune_request** ( )
- **midi.end_of_exclusive** ( )
- **midi.clock** ( )
- **midi.start** ( )
- **midi.continue** ( )

- **midi.stop** ( )
- **midi.active_sensing** ( )
- **midi.reset** ( )
  → *msg*

  Parameters: none.

Decoding MIDI messages

- **midi.decode** (*msg* [, *stringize*])
  → *name*, *channel*, *parameters*

  Decodes a MIDI message. The *msg* argument is expected to be a binary string containing the MIDI message starting from the MIDI status byte. Returns the MIDI message *name* (or *nil* if not recognized), the *channel* (1-16, or *nil* if not applicable), and a table containing the decoded *parameters* (see the table below). If *stringize* is *true*, the parameters are returned as a string instead of a table (this may be useful for dumps).

- **midi.tostring** (*time*, *msg*)
  → *string*

  Returns a printable string containing the passed *time* (an integer) and the decoded *msg* (for dumps).

- **midi.tohex** (*msg*)
  → *msg*

  Returns a string containing the *msg* bytes in hexadecimal format (e.g. '84 0c 2f').

Table 1. MIDI messages

Message name	Channel	Parameters table contents
'note off'	1-16	key = 0-127, velocity = 0-127
'note on'	1-16	key = 0-127, velocity = 0-127

Message name	Channel	Parameters table contents
'aftertouch'	1-16	key = 0-127, pressure = 0-127
'control change'	1-16	number = 0-127, value = 0-127, controller = 'controller name' (see table below)
'program change'	1-16	number = 1-128
'channel pressure'	1-16	pressure = 0-127
'pitch wheel'	1-16	value = 0-16383 (0x3fff), lsb = 0-127, msb = 0-127
'system exclusive'	*nil*	*@@ TBD*
'mtc quarter frame'	*nil*	nnn = 0-7, dddd = 0-15, value = 0-127 (0nnndddd)
'song position'	*nil*	value = 0-16383 (0x3fff), lsb = 0-127, msb = 0-127
'song select'	*nil*	number = 1-128
'tune request'	*nil*	none
'end of exclusive'	*nil*	none
'clock'	*nil*	none
'start'	*nil*	none
'continue'	*nil*	none
'stop'	*nil*	none
'active sensing'	*nil*	none
'reset'	*nil*	none

Table 2. MIDI controllers

Controller number	Controller name
0 (0x00)	'bank select (coarse)'
1 (0x01)	'modulation wheel (coarse)'
2 (0x02)	'breath controller (coarse)'
4 (0x04)	'foot pedal (coarse)'
5 (0x05)	'portamento time (coarse)'

Controller number	Controller name
6 (0x06)	'data entry (coarse)'
7 (0x07)	'volume (coarse)'
8 (0x08)	'balance (coarse)'
10 (0x0a)	'pan position (coarse)'
11 (0x0b)	'expression (coarse)'
12 (0x0c)	'effect control 1 (coarse)'
13 (0x0d)	'effect control 2 (coarse)'
16 (0x10)	'general purpose slider 1'
17 (0x11)	'general purpose slider 2'
18 (0x12)	'general purpose slider 3'
19 (0x13)	'general purpose slider 4'
32 (0x20)	'bank select (fine)'
33 (0x21)	'modulation wheel (fine)'
34 (0x22)	'breath controller (fine)'
36 (0x24)	'foot pedal (fine)'
37 (0x25)	'portamento time (fine)'
38 (0x26)	'data entry (fine)'
39 (0x27)	'volume (fine)'
40 (0x28)	'balance (fine)'
42 (0x2a)	'pan position (fine)'
43 (0x2b)	'expression (fine)'
44 (0x2c)	'effect control 1 (fine)'
45 (0x2d)	'effect control 2 (fine)'
64 (0x40)	'hold pedal'
65 (0x41)	'portamento'
66 (0x42)	'sustenuto pedal'
67 (0x43)	'soft pedal'
68 (0x44)	'legato pedal'

Controller number	Controller name
69 (0x45)	'hold 2 pedal'
70 (0x46)	'sound variation'
71 (0x47)	'sound timbre'
72 (0x48)	'sound release time'
73 (0x49)	'sound attack time'
74 (0x4a)	'sound brightness'
75 (0x4b)	'sound control 6'
76 (0x4c)	'sound control 7'
77 (0x4d)	'sound control 8'
78 (0x4e)	'sound control 9'
79 (0x4f)	'sound control 10'
80 (0x50)	'general purpose button 1'
81 (0x51)	'general purpose button 2'
82 (0x52)	'general purpose button 3'
83 (0x53)	'general purpose button 4'
91 (0x5b)	'effects level'
92 (0x5c)	'tremulo level'
93 (0x5d)	'chorus level'
94 (0x5e)	'celeste level'
95 (0x5f)	'phaser level'
96 (0x60)	'data button increment'
97 (0x61)	'data button decrement'
98 (0x62)	'non-registered parameter (fine)'
99 (0x63)	'non-registered parameter (coarse)'
100 (0x64)	'registered parameter (fine)'
101 (0x65)	'registered parameter (coarse)'
120 (0x78)	'all sound off'
121 (0x79)	'all controllers off'

Controller number	Controller name
122 (0x7a)	'local keyboard'
123 (0x7b)	'all notes off'
124 (0x7c)	'omni mode off'
125 (0x7d)	'omni mode on'
126 (0x7e)	'mono operation'
127 (0x7f)	'poly operation'

Other MIDI utilities

- **midi.note_key** (*frequency*)
  → *msg*

  Returns the nearest MIDI note key corresponding to *frequency* (Hz).

- **midi.note_frequency** (*key*)
  → *msg*

  Returns the *frequency* (Hz) corresponding to the MIDI note *key*.

- **midi.tmsg** (*time*, *msg*)
  → *tmsg*

  Returns a binary string obtained by concatenating the passed *time* (an integer) and the binary MIDI *msg* (this format can be convenient if one wants to send MIDI events over ringbuffers).

- **midi.time_msg** (*tmsg*)
  → *time*, *msg*

  Extracts the *time* and *msg* from a *tmsg* binary string constructed with the midi.tmsg() function.

Summary of LuaJack functions

The tables below summarize the functions provided by LuaJack.

The availability ('Avail.') column indicates whether a function is available in the main context (M), in the process context (T) and/or in the thread context (T).

On error, unless otherwise pointed out in the manual sections, all functions call Lua's error() with a string message handler.

Table 3. Clients

Function	Return values	Description	Avail.
jack.client_open (*name* [, *options*])	*client*	Create a JACK client.	M
jack.client_name_size ( )	*maxsize*	Get the maximum allowed size for client names.	M
jack.client_close (*client*)	-	Close a client.	M
jack.client_name (*client*)	*name*	Get a client's name.	M
jack.client_uuid (*client*)	*uuid*	Get a client's UUID.	M
jack.client_uuid_to_name (*client*, *uuid*)	*name*	Translate a client's UUID into its name.	M
jack.client_name_to_uuid (*client*, *name*)	*uuid*	Translate a client's name into its UUID.	M
jack.activate (*client*)	-	Activate a client.	M
jack.deactivate (*client*)	-	Deactivate a client.	M
jack.is_realtime (*client*)	*boolean*	Check if JACK is running realtime.	M
jack.cpu_load (*client*)	*cpuload*	Get the current CPU load estimate.	MPT
jack.sample_rate (*client*)	*sample_rate*	Get the sample rate.	MPT
jack.set_buffer_size (*client*, *nframes*)	-	Change the buffer size.	MP
jack.buffer_size (*client*)	*nframes*	Get the current maximum buffer size.	MPT
jack.freewheel (*client*, *onoff*)	-	Start/stop JACK's '*freewheel*' mode.	MP

Table 4. Process context

Function	Return values	Description	Avail.
jack.process_load (*client*, *chunk*, …)	-	Create process context loading the chunk from string.	M
jack.process_loadfile (*client*, *filename*, …)	-	Create process context loading the chunk from file.	M

Table 5. Real-time callbacks

Function	Callback prototype	Avail.
jack.process_callback (*client*, *func*)	*func(nframes)*	P
jack.buffer_size_callback (*func*)	*func(nframes)*	P
jack.sync_callback (*client*, *func*)	*func(state, position)*	P
jack.timebase_callback (*client*, *func* [, *conditional*])	*func(state, nframes, position, new_position)*	P
jack.release_timebase (*client*)	-	P

Table 6. Non real-time callbacks

Function	Callback prototype	Avail.
jack.shutdown_callback (*client*, *func*)	*func(client, code, reason)*	M
jack.freewheel_callback (*client*, *func*)	*func(client, operation)*	M
jack.sample_rate_callback (*client*, *func*)	*func(client, nframes)*	M
jack.client_registration_callback (*client*, *func*)	*func(client, name, 'registered' | 'unregistered')*	M
jack.port_registration_callback (*client*, *func*)	*func(client, portname, 'registered' | 'unregistered')*	M
jack.port_rename_callback (*client*, *func*)	*func(client, portname, newname)*	M
jack.port_connect_callback (*client*, *func*)	*func(client, srcname, dstname, 'connected' | 'disconnected')*	M
jack.graph_order_callback (*client*, *func*)	*func(client)*	M
jack.xrun_callback (*client*, *func*)	*func(client)*	M
jack.latency_callback (*client*, *func*)	*func(client, 'capture' | 'playback')*	M
jack.session_callback (*client*, *func*)	*command, flag1, flag2 = func(client, type, path, uuid)*	M

Table 7. Client threads

Function	Return values	Description	Avail.
jack.thread_load (*client*, *chunk*, ...)	*thread*	Create a client thread loading the chunk from a string.	M
jack.thread_loadfile (*client*, *filename*, ...)	*thread*	Create a client thread loading the chunk from a file.	M
jack.self ( )	*client*, *thread*	Get the client and thread references.	T
jack.signal (*client*, *thread*)	-	Signal a thread's condition variable.	MPT
jack.wait ( )	-	Waits on the calling thread's condition variable.	T
jack.real_time_priority (*client*)	*priority*	Get the priority that threads created by JACK will run at.	MPT
jack.max_real_time_priority (*client*)	*priority*	Get the maximum priority that a realtime client thread should use.	MPT
jack.acquire_real_time_scheduling (*priority*)	-	Acquire realtime scheduling.	MPT
jack.drop_real_time_scheduling ( )	-	Drop realtime scheduling.	MPT

Table 8. Ports

Function	Return values	Description	Avail.
jack.input_audio_port (*client* , *name* [, *options*])	*port*	Create an input audio port.	M
jack.output_audio_port (*client* , *name* [, *options*])	*port*	Create an output audio port.	M
jack.input_midi_port (*client* , *name* [, *options*])	*port*	Create an input midi port.	M
jack.output_midi_port (*client* , *name* [, *options*])	*port*	Create an output midi port.	M
jack.input_custom_port (*client* , *name* [, *options*])	*port*	Create an input custom port.	M
jack.output_custom_port (*client* , *name* [, *options*])	*port*	Create an output custom port.	M
jack.port_name_size ( )	*maxsize*	Get the maximum allowed size for port names.	M

Function	Return values	Description	Avail.
jack.port_type_size ( )	*maxsize*	Get the maximum allowed size for port types.	M
jack.port_unregister (*port*)	-	Delete a port.	M
jack.connect (*port*, *portname2*) **jack.port_connect** (*port*, *portname2*) **jack.nport_connect** (*client*, *portname1*, *portname2*)	-	Connect two ports.	M
jack.disconnect (*port* [, *portname2*]) **jack.port_disconnect** (*port* [, *portname2*]) **jack.nport_disconnect** (*client*, *portname1* [, *portname2*])	-	Disconnects two ports.	M
jack.port_name (*port*)	*fullname*	Get the full name of a port.	M
jack.port_short_name (*port*)	*shortname*	Get the short name of a port.	M
jack.port_set_name (*port*, *newname*)	-	Changes the short name of a port.	M
jack.port_flags (*port*) **jack.nport_flags** (*client*, *portname*)	*flags*, *table*	Get the flags of a port.	M
jack.port_uuid (*port*) **jack.nport_uuid** (*client*, *portname*)	*uuid*	Get the UUID of a port.	M
jack.port_type (*port*) **jack.nport_type** (*client*, *portname*)	*type*	Get the type of a port.	M
jack.nport_exists (*client*, *portname*)	*boolean*	Check if a port exists.	M
jack.port_is_mine (*client*, *port*) **jack.nport_is_mine** (*client*, *portname*)	*boolean*	Check if a port belongs to a given client.	M
jack.port_set_alias (*port*, *alias*) **jack.nport_set_alias** (*client*, *portname*, *alias*)	-	Set an alias for a port.	M
jack.port_unset_alias (*port*, *alias*) **jack.nport_unset_alias** (*client*, *portname*, *alias*)	-	Unset an alias for a port.	M
jack.port_aliases (*port*) **jack.nport_aliases** (*client*, *portname*)	*alias1*, *alias2*	Get the aliases for a port.	M
jack.port_connections (*port* [, *list*]) **jack.nport_connections** (*client*, *portname* [, *list*])	*N*, *{ portname1, …, portnameN }*	Get the connections for a port.	M

Function	Return values	Description	Avail.
jack.port_connected_to (*port*, *portname2*) **jack.nport_connected_to** (*client*, *portname1*, *portname2*)	*boolean*	Check if two ports are connected.	M
jack.port_monitor (*port*, *onoff*) **jack.nport_monitor** (*client*, *portname*, *onoff*)	-	Enable/disable input monitoring for a port.	M
jack.port_monitoring (*port*) **jack.nport_monitoring** (*client*, *portname*)	*boolean*	Check if the input monitoring for a port is enabled.	M
jack.get_ports (*client* [, *filter*])	*{ portname1, ..., portnameN }*	List ports.	M

Table 9. Latency

Function	Return values	Description	Avail.
jack.latency_range (*port*, *mode*)	*min*, *max*	Get the minimum and maximum latencies.	M
jack.set_latency_range (*port*, *mode*, *min*, *max*)	-	Set the minimum and maximum latencies.	M
jack.recompute_total_latencies (*client*)	-	Request a complete recomputation of all port latencies.	M

Table 10. Time

Function	Return values	Description	Avail.
jack.time ( )	*useconds*	Get the current system time.	MPT
jack.frame (*client*)	*frameno*	Get the estimated current time in frames.	MPT
jack.since (*useconds*)	*elapsed*	Get the time elapsed since a point of time in the past.	MPT
jack.since_frame (*client*, *frameno*)	*nframes*	Get the time in frames elapsed since a frame number in the past.	MPT
jack.frames_to_time (*client*, *nframes*)	*useconds*	Translate time from microseconds to frames.	MPT
jack.time_to_frames (*client*, *useconds*)	*nframes*	Translate time from frames to microseconds.	MPT
jack.frames_since_cycle_start (*client*)	*nframes*	Get the no. of frames passed since the beginning of the current process cycle.	MPT
jack.last_frame_time (*client*	*nframes*	Get the time in frames at the start of the current process cycle.	P
jack.cycle_times (*client*)	*current_frames*, *current_usecs*, *next_usecs*, *period_usecs*	Get internal cycle timing information.	P

Table 11. Transport and timebase

Function	Return values	Description	Avail.
jack.current_transport_frame (*client*)	*frameno*	Get an estimate of the current transport frame.	MPT
jack.transport_state (*client*)	*state*	Get the current transport state.	MPT
jack.transport_query (*client*)	*state, position*	Get the current transport state and position.	MPT
jack.transport_start (*client*)	-	Start the transport rolling.	MPT
jack.transport_stop (*client*)	-	Stop the transport rolling.	MPT
jack.transport_locate (*client, frameno*)	-	Reposition the transport frame.	MPT
jack.transport_reposition (*client, position*)	-	Request a new transport position.	MPT

Table 12. Reading and writing audio data

Function	Return values	Description	Avail.
jack.get_buffer (*port*)	*nframes*	Get the port buffer.	P
jack.seek (*port* [, *position*])	*position, available*	Get/set the port buffer’s current position.	P
jack.read (*port* [, *count*])	*sample1, …*	Read samples from the port buffer.	P
jack.write (*port* [, *sample1* , …])	*count*	Write samples to the port buffer.	P
jack.clear (*port* [, *count*])	*count*	Clear buffer.	P
jack.copy (*dstport, srcport* [, *count*])	*count*	Copy samples between ports buffers.	P

Table 13. Reading and writing MIDI data

Function	Return values	Description	Avail.
jack.get_buffer (*port*)	*eventcount, lostcount*	Get the port buffer (input port).	P
jack.get_buffer (*port*)	*space*	Get the port buffer (output port).	P
jack.seek (*port* [, *index*])	*index, available*	Get/set the port buffer’s current index.	P

Function	Return values	Description	Avail.
jack.read (*port* [, *index*])	*time, data*	Read a MIDI event from the port buffer.	P
jack.write (*port, time, data*)	*space*	Write a MIDI event to the port buffer.	P
jack.copy (*dstport, srcport* [, *count*])	*count*	Copy MIDI events between port buffers.	P

Table 14. Statistics

Function	Return values	Description	Avail.
jack.max_delayed_usecs (*client*)	*delay*	Get the the maximum reported delay.	M
jack.xrun_delayed_usecs (*client*)	*delay*	Get the delay due to the most recent xrun.	M
jack.reset_max_delayed_usecs (*client*)	-	Reset the maximum delay counter.	M

Table 15. Session API

Function	Return values	Description	Avail.
jack.session_notify (*client, target, type, path*)	*{ reply1, ..., replyN }*	Send an event to session clients.	M
jack.has_session_callback (*client, clientname*)	*boolean*	Check if client is listening to session notifications.	M
jack.reserve_client_name (*client, clientname*)	-	Reserve a client name.	M

Table 16. Ringbuffers

Function	Return values	Description	Avail.
jack.ringbuffer (*client, size* [, *mlock* [, *usepipe*]])	*rbuf*	Create a ringbuffer.	M
jack.ringbuffer_write (*rbuf, tag* [, *data*])	*ok*	Write a message to a ringbuffer.	MPT
jack.ringbuffer_read (*rbuf*)	*tag, data*	Read a message from a ringbuffer.	MPT
jack.ringbuffer_reset (*rbuf*)	-	Reset a ringbuffer.	MPT

Function	Return values	Description	Avail.
jack.ringbuffer_peek (*rbuf*)	*ok*	Check if there is a message to read from a ringbuffer.	MPT
jack.ringbuffer_getfd (*rbuf*)	*fd*	Get the read file descriptor of a ringbuffer’s pipe.	MPT