The Lua Module Reference

Marc Balmer marc@msys.ch

Table of Contents

Open Source Lua Modules

This is the documentation for selected open source Lua modules maintained by micro systems on the public github repository at https://github.com/arcapos/.

Lua Modules for General Use

Proxying Lua States

The **proxy** module allows for new Lua states to be created from an existing Lua state and to access data in the other Lua state. Once a proxy object has been created, it can be accessed like a regular Lua table to set and get variable from the proxied state.

Creating a Lua state

```
proxy.new()
```

Create a new Lua state and return a handle to it.

Accessing a Lua state

```
proxy:dostring(chunk)
```

Execute the Lua chunk in the proxy state.

```
proxy.variable = value
```

Set *variable* in the proxy state to *value*.

Unix Functions

The **unix** module is used to access Unix specific functionality that is not found elsewhere, e.g. forking a process, accessing the system log etc. The unix module does not aim at being a complete set of all Unix functions and system calls, it merely contains those functions that where needed at some point of the arcapos development.

Process related functions

```
unix.chdir(path)
```

Change the current working directory to path.

```
unix.dup2(oldfd, newfd)
```

`dup2()` makes *newfd* be the copy of *oldfd*, closing *newfd* first if necessary.

```
unix.fork()
```

Fork the current process. Returns the PID in the parent process, 0 in the child process, or, -1 in case of an error (no child process is created in this case).

```
unix.kill(pid, signal)
```

Send the signal *signal* to the process with process id *pid*.

```
unix.getcwd()
```

Returns the current working directory.

```
unix.getpid()
```

Returns the process id of the process calling the function.

```
unix.setpgid(pid, pgid)
```

Set the process group id of process pid to *pgid*.

```
unix.getuid()
```

Returns the user id of the process calling the function.

```
unix.getgid()
```

Returns the group id of the process calling the function.

File related functions

```
unix.chown(path, uid, gid)
```

Change file ownership of the file at *path* to the (numerical) user id *uid* and (numerical) group id *gid*.

```
unix.chmod(path, mode)
```

Change the file access mode of the file at *path* to *mode*.

```
unix.rename(old, new)
```

Rename the file at old to new.

```
unix.stat(path)
```

`stat()` stats the file pointed to by path and returns a table containing the following elments:

st_dev	ID of device containing file
st_ino	inode number
st_mode	protection
st_nlink	number of hard links
st_uid	user ID of owner
st_gid	group ID of owner
st_rdev	device ID (if special file)
st_size	total size, in bytes
st_blksize	blocksize for filesystem I/O
st_blocks	number of 512B blocks allocated
st_atime	time of last access
st_mtime	time of last modification
st_ctime	time of last status change

```
unix.mkdir(path, mode)
```

Create directory *path* with mode *mode*.

```
unix.unlink(path)
```

Unlink (delete) the file at *path*.

Accessing User Information

```
unix.setpwent()
```

Start accessing the user database.

```
unix.endpwent()
```

Stop accessing the user database.

```
unix.getpwent()
```

Get the next password entry. Returns a table with the following fields:

- pw_name
- pw_passpwd
- pw_uid
- pw_gid
- pw_gecos
- pw_dir
- pw_shell

```
unix.getpwnam(username)
```

Return the password entry for user username.Returns a table with the following fields:

- pw_name
- pw_passpwd
- pw_uid
- pw_gid
- pw_gecos
- pw_dir
- pw_shell

```
unix.getpwuid(uid)
```

Return the password entry for the user with the given user id uid. Returns a table with the following fields:

- pw_name
- pw_passpwd
- pw_uid
- pw_gid
- pw_gecos
- pw_dir
- pw_shell

```
unix.getgrnam(name)
```

Return the group entry for the group name. Returns a table with the following fields:

- gr_name
- gr_passwd
- gr_gid
- gr_mem

The `gr_mem` field is itself a table containing all members of this group.

```
unix.getgrgid(gid)
```

Get the group entry for the group with the numerical id *gid*. The result is the same table as for the `getgrnam()` function.

Getting and setting the system hostname

```
unix.gethostname()
```

Return the hostname or nil if an error occurs.

```
unix.sethostname(hostname)
```

Set the hostname, returns true on success, nil on error.

Using the system log

```
unix.openlog(ident, option, facility)
```

Open the system log with the given *ident*, *option*, and, *facility*.

```
unix.syslog(level, message)
```

Log *message* at the given *level*.

```
unix.closelog()
```

Close the system log.

```
unix.setlogmask(mask)
```

Sets the log mask to *mask* and returns the old value.

Select

```
unix.fd_set()
```

Obtains a file descriptor set for later selecting on it. The set is initially zeroed.

```
fdset:clr(fd)
```

Clear *fd* in the file descriptor set.

```
fdset:isset(fd)
```

Set *fd* in the file descriptor set.

```
fdset:set(fd)
```

Check if *fd* is set in the filedescriptor set. Returns true if *fd* is set, false otherwise.

```
fdset:zero()
```

Zero (clear) the file descriptor set.

```
unix.select(nfds, readfds, writefds, errorfds [, timeout])
```

Perform `select()` on the specified file descriptor sets. Pass nil to omit one or more of the file descriptor sets. *nfds* is highest file descriptor number passed in the sets plus one. Timeout is either a single value representing milliseconds or two comma separated integers representing seconds and milliseconds. `select()` returns the number of file descriptors ready or -1 if an error occurs. If no *timeout* is specified, `select()` effectively becomes a poll and returns 0 if no file descriptors are ready.

Miscellaneous functions

```
unix.arc4random()
```

The `arc4random()` function uses the key stream generator employed by the arc4 cipher, which uses

8*8 8 bit S-Boxes. The S-Boxes can be in about (2^1700) states. The `arc4random()` function returns pseudo-random numbers in the range of 0 to (2^32)-1.

```
unix.errno()
```

Returns the last error code.

```
signal(sigcode, action)
```

Set the action for a signal. The following values for action are valid:

unix.SIG_DFL	Install the default signal handler.
unix.SIG_IGN	Ignore the signal.
unix.SIG_REAPER	Use with *sigcode* `SIGCHLD` only. Installs a signal handler that calls `wait(2)` to prevent zombie processes when a child process terminates.

```
unix.sleep(seconds)
```

Sleep for the number of seconds passed.

```
unix.getpass(prompt)
```

Display the prompt and get a password on the console.

UUIDs

The **uuid** module is used to generate and parse uuids. It uses the libuuid by Theodore Y. Ts'o. libuuid is part of the util-linux package since version 2.15.1 and is available from ftp://ftp.kernel.org/pub/linux/utils/util-linux/. This documentation is therefore based on the libuuid documentation.

The UUID is internally 16 bytes (128 bits) long, which gives approximately 3.4x10^38 unique values (there are approximately 10^80 elementary particles in the universe according to Carl Sagan's Cosmos). The new UUID can reasonably be considered unique among all UUIDs created on the local system, and among UUIDs created on other systems in the past and in the future.

UUID Generating Functions

```
uuid.generate([format])
```

The `generate()` function creates a new uuid. The uuid will be generated based on high-quality

randomness from `/dev/urandom`, if available. If it is not available, then `generate()` will use an alternative algorithm which uses the current time, the local ethernet MAC address (if available), and random data generated using a pseudo-random generator.

If the optional parameter *format* is the string `t`, then uuid will be returned as a string, otherwise it will be returned as a uuid object with a proper metatable.

```
uuid.generate_random([format])
```

The `generate_random()` function forces the use of the all-random UUID format, even if a high-quality random number generator (i.e., `/dev/urandom`) is not available, in which case a pseudo-random generator will be substituted. Note that the use of a pseudo-random generator may compromise the uniqueness of UUIDs generated in this fashion.

```
uuid.generate_time([format])
```

The `generate_time()` function forces the use of the alternative algorithm which uses the current time and the local ethernet MAC address (if available). This algorithm used to be the default one used to generate UUID, but because of the use of the ethernet MAC address, it can leak information about when and where the UUID was generated. This can cause privacy problems in some applications, so the `generate()` function only uses this algorithm if a high-quality source of randomness is not available. To guarantee uniqueness of UUIDs generated by concurrently running processes, the underlying uuid library uses a global clock state counter (if the process has permissions to gain exclusive access to this file) and/or the uuidd daemon, if it is running already or can be spawned by the process (if installed and the process has enough permissions to run it). If neither of these two synchronization mechanisms can be used, it is theoretically possible that two concurrently running processes obtain the same UUID(s). To tell whether the UUID has been generated in a safe manner, use `generate_time_safe()`.

```
uuid.generate_time_safe([format])
```

The `generate_time_safe()` is similar to `generate_time()`, except that it returns a value which denotes whether any of the synchronization mechanisms (see above) has been used.

```
uuid.parse(string)
```

The `parse()` function converts the UUID string into a uuid. The input UUID is a string of the form `967c2ed8-7903-4ace-8a27-97daf7f63097` (in `string.format()` format `%08x-%04x-%04x-%04x-%012x`, 36 bytes.)

UUID functions

```
uuid:clear()
```

Clear the memory used by a uuid object. This is usuall called by the Lua garbage collector.

```
uuid:compare(uuid2)
```

Compare *uuid* to *uuid2*. Returns an integer less than, equal to, or greater than zero if *uuid* lexicographically less than, equal, or greater than *uuid2*.

```
uuid.is_null(uuid)
```

The `is_null()` function compares the value of the supplied UUID object uuid to the NULL value. If the value is equal to the NULL UUID, true is returned, otherwise false is returned.

```
uuid.time(uuid)
```

The `time()` function extracts the time at which the supplied timebased UUID uuid was created. Note that the UUID creation time is only encoded within certain types of UUIDs. This function can only reasonably be expected to extract the creation time for UUIDs created with the `generate_time()` and `generate_time_safe()` functions. It may or may not work with UUIDs created by other mechanisms.

The `time()` functions returns two integers, the seconds value and the microseconds value.

```
uuid.unparse(uuid)
```

The `unparse()` function converts the supplied UUID uuid from the binary representation into a 36-byte string of the form `967c2ed8-7903-4ace-8a27-97daf7f63097`. The case of the hex digits returned by `unparse()` may be upper or lower case, and is dependent on the system-dependent local default.

Metamethods

The following metamethods are defined for uuids.

__eq
: The `==` operation. Test if two uuids are equal.

__lt
: The `<` operation. Test if one uuid is less than another.

__le
: The `<=` operator. Test if one uuid is less or equal than another.

__tostring
: Convert a uuid to its string representation.

__concat
: Concatenate a uuid.

__len

The length operator #. This returns the length in bytes of the textual representation of the uuid.

Data Formats

JSON

The **json** module is used to encode and decode data in JSON format (Javascript Object Notation).

Decoding JSON data

```
json.decode(data [, nullHandling])
```

Decode JSON encoded data. The optional string argument *nullHandling* specifies how JSON null values are mapped to Lua values:

'json-null'

Maps to a Lua table with a special `JSON null` Metatable that can be detected using the `isnull()` function described below. This is the default.

'empty-string'

Maps JSON null values to an empty string, which can be useful in web-based applications where e.g. PostgreSQL is used to generate JSON data which is then handed to a browser over a WebSocket or similar mechanism.

'nil'

Maps JSON null values to Lua `nil`.

`json.decode()` returns the decoded values as Lua values or `nil` if an error occurs. If an error occured, a second return value contains the error message.

Encoding Lua values into JSON format

```
json.encode(data)
```

Encode data into JSON format.

Handling of JSON-null values

JSON has a special datatype to denote no value: the JSON null value. To insert a JSON null value, assign `json.null`. Use the following function to test for JSON null values.

```
json.isnull(var)
```

Returns true if *var* is JSON null.

YAML

The **yaml** module is used to parse files in YAML format ("YAML Ain't Markup Language").

```
yaml.parse(text [, env])
```

Parse YAML data from a string and return a table containing the data. The optional parameter *env* is the environment to be used for Lua code in the YAML data.

See the section "A Note on YAML Tags" for details on embedding Lua code in YAML data.

```
yaml.parsefile(path [, env])
```

Parse YAML data from the file *path* and return a table containing the data.

```
level = yaml.verbosity([level])
```

Set the verbosity level to *level* and returns the old verbosity level. If no *level* parameter is given, returns the current verbosity level.

Set this to 1 to have events printed to the console while parsing.

For this to work, the yaml Lua module must have been compiled with the -DDEBUG option. Otherwise the function will always return nil.

A Note on YAML Tags

Values in YAML data can be annotated with tags. Default tags start with !! whereas local tags start with a single ! character.

The following YAML data uses some default tags to make sure the right types are selected:

```
boolean_value: !!bool True

string_value: !!str True
```

The YAML Lua module introduces five local tags: **!Lua/load**, **!Lua/call**, **!Lua/loadfile**, **!Lua/callfile**, and, **!file**.

!Lua/load will load a chunk and assign it to a value, but does not execute it.

!Lua/call will load a chunk and execute it and assign to the value whatever the chunk returns.

!Lua/loadfile will assign the Lua code in a file as a chunk to a value, but does not execute it.

!Lua/callfile will load and call a file and assign to the value whatever the code returns.

!file will assign the content of a file to a value.

An optional environment can be specified when parsing YAML data, this environment will then be used for all use cases.

```
myFunction: !Lua/load
  a = 40 + 2
  return 'The answer is ' .. a

myResult: !Lua/call return os.date()

myFunctionFromFile: !Lua/loadfile myfunction.lua

myValueFromFile: !Lua/callfile myvalue.lua

myContentFromFile: !file logo.png
```

Networking

Web/Internet Transactions

The **curl** module is used to create and perform web transactions like HTTP requests, FTP transfers etc.

curl implements the cURL easy interface described at http://curl.haxx.se/libcurl/c/libcurl-easy.html and the cURL multi interface. Constants defined by the C interface are mapped to Lua in the following way:

`CURLE_NAMED` becomes `curl.NAMED` etc.

The typical use is to create a curl session, set the options (i.e. what type of transfer it is, the URL, any data etc.) and then to perform the session.

Creating and performing requests

```
curl.easy()
```

Create a new curl session using the cURL easy interface.

```
req:setopt(option, value)
```

Set a curl option.

```
req:getinfo(code)
```

Request internal information from the curl session.

```
req:perform()
```

Perform the request.

```
req:close()
```

Close the curl session.

Escaping and unescaping of URL strings

```
curl.escape(string)
```

Escape *string* for use in a URL.

```
curl.unescape(sting)
```

Unescape the URL-escaped *string* parameter.

Network Clients and Servers

The **net** module is used to implement network clients or servers. It supports IPv4, IPv6, and local sockets.

Creating network servers

```
net.bind(hostname [, port] [, backlog])
```

Bind a socket on the specified *hostname* and *port*. This also does the listen system call. If the *hostname* argument starts with a slash or dot character, a local socket (AF_UNIX) is assumed, an IP socket otherwise.

```
sock:accept()
```

Accept a new connection and return a new socket for the new connection.

```
sock:close()
```

Close a socket.

Creating network clients

```
net.connect(hostname, port)
```

Connect to *hostname* at the specified *port* and return a new socket.

Transferring data

```
sock:write(data)
```

Write *data* to the socket.

```
sock:print(string)
```

Write string to the socket and append a newline character.

```
sock:read([timeout])
```

Read data from a socket with an optional timeout in milliseconds. Returns the data read or nil if the timeout expires or an error occured.

```
sock:readln([timeout])
```

Read data up to the first newline character from a socket with an optional timeout in milliseconds. Returns the data read or nil if the timeout expires or an error occured.

Filedescriptor passing

Open filedescriptors can be passed only over AF_UNIX sockets.

```
sock:sendfd(fd)
```

Send a filedescriptor.

```
sock:recvfd(fd)
```

Receive a filedescriptor.

Miscellaneous functions

```
sock:socket()
```

Return as an integer the underlying socket.

```
sock:close()
```

Close a socket.

Web Services

FastCGI Servers

The **fcgi** module is used to implement FastCGI servers, i.e. servers that run as daemon processes and that are contacted by a webserver using the FastCGI protocol to deliver dynamic content.

Creating FastCGI servers

Before a webserver can connect to a FastCGI server, such a server must be created by opening a socket and waiting for connections on it.

```
local socket = fcgi.openSocket(path, backlog)
```

Create a FastCGI listen socket. *path* is the Unix domain socket, or a colon followed by a port number. e.g. `/var/run/.s`, `:5000`. *backlog* is the listen queue depth used in the `listen()` system call. Returns the sockets file descriptor or -1 on error.

Handling FastCGI connections

Firs a FastCGI request object must be created:

```
local request = fcgi.initRequest(socket)
```

```
request:accept()
```

Accept a new request.

```
request:finish()
```

Finish the current request.

```
request:fflush()
```

Flushes any buffered output. Server-push is a legitimate application of flush. Otherwise, `flush()` is not very useful, since accept does it implicitly. Calling `flush()` in non-push applications results in extra writes and therefore reduces performance.

```
request:getLine(count)
```

Reads up to *count*-1 consecutive bytes from the input stream. Stops before *count*-1 bytes have been

read if `\n` or EOF is read. The terminating `\n` is copied into the result. After copying the last byte. Returns nil if EOF is the first thing read from the input stream, the string otherwise.

```
request:getStr(count)
```

Reads up to *count* consecutive bytes from the input stream. Performs no interpretation of the input bytes. Returns the number of bytes read. If the result is smaller than *count*, the end of input has been reached.

```
request:getParam(name)
```

Obtain the value of an FCGI parameter in the environment.

```
request:getEnv()
```

Obtain a table containing the environment.

```
request:putStr(data)
```

Writes consecutive bytes from the character array *data* into the output stream. Performs no interpretation of the output bytes. Returns the number of bytes written (n) for normal return, -1 if an error occurred.

Convenience Functions

```
request:parse()
```

Parse the current query (i.e. the contents of the `QUERY_STRING` variable) and return a table containing all variables such that the variable name is the key.

WebSocket Servers

The **websocket** module is used to implement WebSocket network servers. The websocket module supports both encrypted (wss://) and unencrypted (ws://) WebSockets.

Creating WebSockets servers

```
websocket.bind(address, port [, pem-file])
```

Create a new WebSocket server, bind it to the address and the port specified. If the optional parameter *pem-file* is passed, it must be the path name of a valid PEM-file containing the server secret key and certificate in PEM-format. A secure WebSocket is created and all communication is

encrypted using SSL/TLS. If *pem-file* is omitted, an unencrypted WebSocket is created.

Accepting and closing connections

```
websock:accept()
```

Accept a new connection and return a new WebSocket object.

```
websock:handshake(request)
```

Perform the WebSocket handshake on a websocket. The handshake only succeeds if the client request matches the *request* parameter.

```
websock:close()
```

Close a WebSocket. This does not perform an SSL/TLS shutdown if *websock* is a secure WebSocket.

```
websock:shutdown()
```

Close a WebSocket, if *websock* is a secure WebSocket, a proper SSL/TLS shutdown is performed.

Transferring data

```
websock:send(data)
```

Send *data* over the socket.

```
websock:recv()
```

Receive data from a socket. Returns the data received or nil if client closed the connection or an error occured.

```
websock:socket()
```

Return the underlying socket as an integer, e.g. to perform select() on it.

Database Access

PostgreSQL

The **pgsql** is module is used to access PostgreSQL databases from Lua code. It is a Lua binding to libpq, the PostgreSQL C language interface and offers more or less the same functionality.

Database connection control functions

The following functions deal with making a connection to a PostgreSQL backend server. An application program can have several backend connections open at one time. (One reason to do that is to access more than one database.) Each connection is represented by a connection object, which is obtained from the function connectdb. The status function should be called to check the return value for a successful connection before queries are sent via the connection object.

```
pgsql.connectdb(conninfo)
```

Makes a new connection to the database server. This function opens a new database connection using the parameters taken from the string conninfo. The passed string can be empty to use all default parameters, or it can contain one or more parameter settings separated by whitespace, or it can contain a URI.

```
pgsql.connectStart(conninfo)
```

Make a connection to the database server in a nonblocking manner. With connectStart, the database connection is made using the parameters taken from the string conninfo as described above for connectdb.

```
pgsql.ping(conninfo)
```

ping reports the status of the server. It accepts connection parameters identical to those of connectdb, described above. It is not necessary to supply correct user name, password, or database name values to obtain the server status; however, if incorrect values are provided, the server will log a failed connection attempt.

```
conn:connectPoll()
```

If connectStart succeeds, the next stage is to poll libpq so that it can proceed with the connection sequence. Use conn:socket to obtain the descriptor of the socket underlying the database connection. Loop thus: If conn:connectPoll() last returned PGRES_POLLING_READING, wait until the socket is ready to read (as indicated by select(), poll(), or similar system function). Then call conn:connectPoll() again. Conversely, if conn:connectPoll() last returned PGRES_POLLING_WRITING, wait until the socket is ready to write, then call conn:connectPoll() again. If you have yet to call connectPoll, i.e., just after the call to connectStart, behave as if it last returned PGRES_POLLING_WRITING. Continue this loop until conn:connectPoll() returns PGRES_POLLING_FAILED, indicating the connection procedure has failed, or PGRES_POLLING_OK, indicating the connection has been successfully made.

```
conn:finish()
```

Closes the connection to the server. Also frees memory used by the underlying connection object. Note that even if the server connection attempt fails (as indicated by status), the application should call finish to free the memory used by the underlying connection object. The connection object must not be used again after finish has been called.

```
conn:reset()
```

Resets the communication channel to the server. This function will close the connection to the server and attempt to reestablish a new connection to the same server, using all the same parameters previously used. This might be useful for error recovery if a working connection is lost.

```
conn:resetStart()
```

Reset the communication channel to the server, in a nonblocking manner.

```
conn:resetPoll()
```

Connection status functions

```
conn:db()
```

Returns the database name of the connection.

```
conn:user()
```

Returns the user name of the connection.

```
conn:pass()
```

Returns the password of the connection.

```
conn:host()
```

Returns the server host name of the connection.

```
conn:port()
```

Returns the port of the connection.

```
conn:tty()
```

Returns the debug TTY of the connection. (This is obsolete, since the server no longer pays attention to the TTY setting, but the function remains for backward compatibility.)

```
conn:options()
```

Returns the command-line options passed in the connection request.

```
conn:status()
```

Returns the status of the connection.

The status can be one of a number of values. However, only two of these are seen outside of an asynchronous connection procedure: CONNECTION_OK and CONNECTION_BAD. A good connection to the database has the status CONNECTION_OK. A failed connection attempt is signaled by status CONNECTION_BAD. Ordinarily, an OK status will remain so until PQfinish, but a communications failure might result in the status changing to CONNECTION_BAD prematurely. In that case the application could try to recover by calling reset.

```
conn:transactionStatus()
```

Returns the current in-transaction status of the server.

The status can be PQTRANS_IDLE (currently idle), PQTRANS_ACTIVE (a command is in progress), PQTRANS_INTRANS (idle, in a valid transaction block), or PQTRANS_INERROR (idle, in a failed transaction block). PQTRANS_UNKNOWN is reported if the connection is bad. PQTRANS_ACTIVE is reported only when a query has been sent to the server and not yet completed.

```
conn:parameterStatus(paramName)
```

Looks up a current parameter setting of the server.

Certain parameter values are reported by the server automatically at connection startup or whenever their values change. parameterStatus can be used to interrogate these settings. It returns the current value of a parameter if known, or nil if the parameter is not known.

Parameters reported as of the current release include server_version, server_encoding, client_encoding, application_name, is_superuser, session_authorization, DateStyle, IntervalStyle, TimeZone, integer_datetimes, and standard_conforming_strings. (server_encoding, TimeZone, and integer_datetimes were not reported by releases before 8.0; standard_conforming_strings was not reported by releases before 8.1; IntervalStyle was not reported by releases before 8.4; application_name was not reported by releases before 9.0.) Note that server_version, server_encoding and integer_datetimes cannot change after startup.

Pre-3.0-protocol servers do not report parameter settings, but pgsql includes logic to obtain values for server_version and client_encoding anyway. Applications are encouraged to use parameterStatus rather than ad hoc code to determine these values. (Beware however that on a pre-3.0 connection, changing client_encoding via SET after connection startup will not be reflected by parameterStatus.) For server_version, see also serverVersion, which returns the information in a numeric form that is much easier to compare against.

If no value for standard_conforming_strings is reported, applications can assume it is off, that is, backslashes are treated as escapes in string literals. Also, the presence of this parameter can be taken as an indication that the escape string syntax (E'...') is accepted.

```
conn:protocolVersion()
```

Interrogates the frontend/backend protocol being used.

Applications might wish to use this function to determine whether certain features are supported. Currently, the possible values are 2 (2.0 protocol), 3 (3.0 protocol), or zero (connection bad). The protocol version will not change after connection startup is complete, but it could theoretically change during a connection reset. The 3.0 protocol will normally be used when communicating with PostgreSQL 7.4 or later servers; pre-7.4 servers support only protocol 2.0. (Protocol 1.0 is obsolete and not supported by pgsql.)

```
conn:serverVersion()
```

Returns an integer representing the backend version.

Applications might use this function to determine the version of the database server they are connected to. The number is formed by converting the major, minor, and revision numbers into two-decimal-digit numbers and appending them together. For example, version 8.1.5 will be returned as 80105, and version 8.2 will be returned as 80200 (leading zeroes are not shown). Zero is returned if the connection is bad.

```
conn:errorMessage()
```

Returns the error message most recently generated by an operation on the connection.

Nearly all pgsql functions will set a message for errorMessage if they fail. Note that by pgsql convention, a nonempty errorMessage result can consist of multiple lines, and will include a trailing newline.

```
conn:socket()
```

Obtains the file descriptor number of the connection socket to the server. A valid descriptor will be greater than or equal to 0; a result of nil indicates that no server connection is currently open. (This will not change during normal operation, but could change during connection setup or reset.)

```
conn:backendPID()
```

Returns the process ID (PID) of the backend process handling this connection.

The backend PID is useful for debugging purposes and for comparison to NOTIFY messages (which include the PID of the notifying backend process). Note that the PID belongs to a process executing on the database server host, not the local host!

```
conn:connectionNeedsPassword()
```

Returns true (1) if the connection authentication method required a password, but none was

available. Returns false (0) if not.

This function can be applied after a failed connection attempt to decide whether to prompt the user for a password.

```
conn:connectionUsedPassword()
```

Returns true if the connection authentication method used a password. Returns false if not.

This function can be applied after either a failed or successful connection attempt to detect whether the server demanded a password.

Command execution functions

```
conn:exec(command)
```

Submits a command to the server and waits for the result.

The command string can include multiple SQL commands (separated by semicolons). Multiple queries sent in a single exec call are processed in a single transaction, unless there are explicit BEGIN/COMMIT commands included in the query string to divide it into multiple transactions. Note however that the returned result object describes only the result of the last command executed from the string. Should one of the commands fail, processing of the string stops with it and the returned result describes the error condition.

```
conn:execParams(command [[, param] ···])
```

Submits a command to the server and waits for the result, with the ability to pass parameters separately from the SQL command text.

The primary advantage of execParams over exec is that parameter values can be separated from the command string, thus avoiding the need for tedious and error-prone quoting and escaping.

Unlike exec, execParams allows at most one SQL command in the given string. (There can be semicolons in it, but not more than one nonempty command.) This is a limitation of the underlying protocol, but has some usefulness as an extra defense against SQL-injection attacks.

```
conn:prepare()
```

Submits a request to create a prepared statement with the given parameters, and waits for completion.

prepare creates a prepared statement for later execution with execPrepared. This feature allows commands that will be used repeatedly to be parsed and planned just once, rather than each time they are executed. prepare is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

The function creates a prepared statement named stmtName from the query string, which must contain a single SQL command. stmtName can be to create an unnamed statement, in which case any pre-existing unnamed statement is automatically replaced; otherwise it is an error if the statement name is already defined in the current session. If any parameters are used, they are referred to in the query as $1, $2, etc.

As with exec, the result is normally a result object whose contents indicate server-side success or failure. A null result indicates out-of-memory or inability to send the command at all. Use errorMessage to get more information about such errors.

```
conn:execPrepared()
```

Sends a request to execute a prepared statement with given parameters, and waits for the result.

execPrepared is like execParams, but the command to be executed is specified by naming a previously-prepared statement, instead of giving a query string. This feature allows commands that will be used repeatedly to be parsed and planned just once, rather than each time they are executed. The statement must have been prepared previously in the current session. PQexecPrepared is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

The parameters are identical to execParams, except that the name of a prepared statement is given instead of a query string, and the paramTypes[] parameter is not present (it is not needed since the prepared statement's parameter types were determined when it was created).

```
conn:describePrepared()
```

Submits a request to obtain information about the specified prepared statement, and waits for completion.

describePrepared allows an application to obtain information about a previously prepared statement. describePrepared is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

stmtName can be or NULL to reference the unnamed statement, otherwise it must be the name of an existing prepared statement. On success, a result with status PGRES_COMMAND_OK is returned. The functions nparams and paramtype can be applied to this result to obtain information about the parameters of the prepared statement, and the functions nfields, fname, ftype, etc provide information about the result columns (if any) of the statement.

```
conn:describePortal(portalName)
```

Submits a request to obtain information about the specified portal, and waits for completion.

describePortal allows an application to obtain information about a previously created portal. (libpq does not provide any direct access to portals, but you can use this function to inspect the properties

of a cursor created with a DECLARE CURSOR SQL command.) PQdescribePortal is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

portalName can be or NULL to reference the unnamed portal, otherwise it must be the name of an existing portal. On success, a result with status PGRES_COMMAND_OK is returned. The functions nfields, fname, ftype, etc can be applied to the result to obtain information about the result columns (if any) of the portal.

Result functions

```
res:status()
```

Returns the result status of the command.

PQresultStatus can return one of the following values:

PGRES_EMPTY_QUERY	**The string sent to the server was empty.**
PGRES_COMMAND_OK	Successful completion of a command returning no data.
PGRES_TUPLES_OK	Successful completion of a command returning data (such as a SELECT or SHOW).
PGRES_COPY_OUT	Copy Out (from server) data transfer started.
PGRES_COPY_IN	Copy In (to server) data transfer started.
PGRES_BAD_RESPONSE	The server's response was not understood.
PGRES_NONFATAL_ERROR	A nonfatal error (a notice or warning) occurred.
PGRES_FATAL_ERROR	A fatal error occurred.
PGRES_COPY_BOTH	Copy In/Out (to and from server) data transfer started. This feature is currently used only for streaming replication, so this status should not occur in ordinary applications.
PGRES_SINGLE_TUPLE	The result contains a single result tuple from the current command. This status occurs only when single-row mode has been selected for the query.

If the result status is PGRES_TUPLES_OK or PGRES_SINGLE_TUPLE, then the functions described below can be used to retrieve the rows returned by the query. Note that a SELECT command that happens to retrieve zero rows still shows PGRES_TUPLES_OK.

PGRES_COMMAND_OK is for commands that can never return rows (INSERT or UPDATE without a RETURNING clause, etc.). A response of PGRES_EMPTY_QUERY might indicate a bug in the client software.

A result of status PGRES_NONFATAL_ERROR will never be returned directly by exec or other query execution functions; results of this kind are instead passed to the notice processor.

```
res:resStatus(status)
```

Converts the enumerated type returned by PQresultStatus into a string constant describing the status code.

```
res:errorMessage()
```

Returns the error message associated with the command, or an empty string if there was no error.

If there was an error, the returned string will include a trailing newline.

Immediately following an exec or getResult call, errorMessage (on the connection) will return the same string as resultErrorMessage (on the result). However, a result will retain its error message until destroyed, whereas the connection's error message will change when subsequent operations are done. Use resultErrorMessage when you want to know the status associated with a particular result; use errorMessage when you want to know the status from the latest operation on the connection.

```
res:errorField(fieldcode)
```

Returns an individual field of an error report.

fieldcode is an error field identifier; see the symbols listed below. NULL is returned if the result is not an error or warning result, or does not include the specified field. Field values will normally not include a trailing newline.

The following field codes are available:

pgsql.PG_DIAG_SEVERITY

The severity; the field contents are ERROR, FATAL, or PANIC (in an error message), or WARNING, NOTICE, DEBUG, INFO, or LOG (in a notice message), or a localized translation of one of these. Always present.

pgsql.PG_DIAG_SQLSTATE

The SQLSTATE code for the error. The SQLSTATE code identifies the type of error that has occurred; it can be used by front-end applications to perform specific operations (such as error handling) in response to a particular database error. For a list of the possible SQLSTATE codes, see Appendix A. This field is not localizable, and is always present.

pgsql.PG_DIAG_MESSAGE_PRIMARY

The primary human-readable error message (typically one line). Always present.

pgsql.PG_DIAG_MESSAGE_DETAIL

Detail: an optional secondary error message carrying more detail about the problem. Might run to multiple lines.

pgsql.PG_DIAG_MESSAGE_HINT

Hint: an optional suggestion what to do about the problem. This is intended to differ from detail in that it offers advice (potentially inappropriate) rather than hard facts. Might run to multiple lines.

pgsql.PG_DIAG_STATEMENT_POSITION

A string containing a decimal integer indicating an error cursor position as an index into the original statement string. The first character has index 1, and positions are measured in characters not bytes.

pgsql.PG_DIAG_INTERNAL_POSITION

This is defined the same as the PG_DIAG_STATEMENT_POSITION field, but it is used when the cursor position refers to an internally generated command rather than the one submitted by the client. The pgsql.PG_DIAG_INTERNAL_QUERY field will always appear when this field appears.

pgsql.PG_DIAG_INTERNAL_QUERY

The text of a failed internally-generated command. This could be, for example, a SQL query issued by a PL/pgSQL function.

pgsql.PG_DIAG_CONTEXT

An indication of the context in which the error occurred. Presently this includes a call stack traceback of active procedural language functions and internally-generated queries. The trace is one entry per line, most recent first.

pgsql.PG_DIAG_SCHEMA_NAME

If the error was associated with a specific database object, the name of the schema containing that object, if any.

pgsql.PG_DIAG_TABLE_NAME

If the error was associated with a specific table, the name of the table. (Refer to the schema name field for the name of the table’s schema.)

pgsql.PG_DIAG_COLUMN_NAME

If the error was associated with a specific table column, the name of the column. (Refer to the schema and table name fields to identify the table.)

pgsql.PG_DIAG_DATATYPE_NAME

If the error was associated with a specific data type, the name of the data type. (Refer to the schema name field for the name of the data type’s schema.)

pgsql.PG_DIAG_CONSTRAINT_NAME

If the error was associated with a specific constraint, the name of the constraint. Refer to fields listed above for the associated table or domain. (For this purpose, indexes are treated as constraints, even if they weren’t created with constraint syntax.)

pgsql.PG_DIAG_SOURCE_FILE

The file name of the source-code location where the error was reported.

pgsql.PG_DIAG_SOURCE_LINE

The line number of the source-code location where the error was reported.

pgsql.PG_DIAG_SOURCE_FUNCTION

The name of the source-code function reporting the error.

The client is responsible for formatting displayed information to meet its needs; in particular it should break long lines as needed. Newline characters appearing in the error message fields should be treated as paragraph breaks, not line breaks.

Errors generated internally by pgsql will have severity and primary message, but typically no other fields. Errors returned by a pre-3.0-protocol server will include severity and primary message, and sometimes a detail message, but no other fields.

Note that error fields are only available from result objects, not conn objects; there is no errorField function.

Retrieving query result information

These functions are used to extract information from a result object that represents a successful query result (that is, one that has status PGRES_TUPLES_OK or PGRES_SINGLE_TUPLE). They can also be used to extract information from a successful Describe operation: a Describe's result has all the same column information that actual execution of the query would provide, but it has zero rows. For objects with other status values, these functions will act as though the result has zero rows and zero columns.

```
res:ntuples()
```

Returns the number of rows (tuples) in the query result. Because it returns an integer result, large result sets might overflow the return value on 32-bit operating systems.

```
res:nfields()
```

Returns the number of columns (fields) in each row of the query result.

```
res:fname(columnNumber)
```

Returns the column name associated with the given column number. Column numbers start at 1.

```
res:fnumber(columnName)
```

Returns the column number associated with the given column name.

-1 is returned if the given name does not match any column.

The given name is treated like an identifier in an SQL command, that is, it is downcased unless double-quoted.

```
res:ftable(columnNumber)
```

Returns the OID of the table from which the given column was fetched. Column numbers start at 1.

```
res:ftablecol(columnNumber)
```

Returns the column number (within its table) of the column making up the specified query result column. Query-result column numbers start at 1.

```
res:fformat(columnNumber)
```

Returns the format code indicating the format of the given column. Column numbers start at 1.

Format code zero indicates textual data representation, while format code one indicates binary representation. (Other codes are reserved for future definition.)

```
res:ftype(columnNumber)
```

Returns the data type associated with the given column number. The integer returned is the internal OID number of the type. Column numbers start at 1.

You can query the system table pg_type to obtain the names and properties of the various data types. The OIDs of the built-in data types are defined in the file src/include/catalog/pg_type.h in the PostgreSQL source tree.

```
res:fmod(columnNumber)
```

Returns the type modifier of the column associated with the given column number. Column numbers start at 1.

The interpretation of modifier values is type-specific; they typically indicate precision or size limits. The value -1 is used to indicate no information available. Most data types do not use modifiers, in which case the value is always -1.

```
res:fsize(columnNumber)
```

Returns the size in bytes of the column associated with the given column number. Column numbers start at 1.

fsize returns the space allocated for this column in a database row, in other words the size of the

server's internal representation of the data type. (Accordingly, it is not really very useful to clients.) A negative value indicates the data type is variable-length.

```
res:binaryTuples()
```

Returns true if the result contains binary data and false if it contains text data.

This function is deprecated (except for its use in connection with COPY), because it is possible for a single result to contain text data in some columns and binary data in others. fformat is preferred. binaryTuples returns true only if all columns of the result are binary (format 1).

```
res:getvalue(rowNumber, columNumber)
```

Returns a single field value of one row of a result. Row and column numbers start at 1.

For data in text format, the value returned by getvalue is a string representation of the field value. For data in binary format, the value is in the binary representation determined by the data type's typsend and typreceive functions. (The value is actually followed by a zero byte in this case too, but that is not ordinarily useful, since the value is likely to contain embedded nulls.)

An empty string is returned if the field value is null. See getisnull to distinguish null values from empty-string values.

```
res:getisnull(rowNumber, columnNumber)
```

Tests a field for a null value. Row and column numbers start at 1.

This function returns true if the field is null and false if it contains a non-null value. (Note that getvalue will return an empty string, not nil, for a null field.)

```
res:getlength(rowNumber, columnNumber)
```

Returns the actual length of a field value in bytes. Row and column numbers start at 1.

This is the actual data length for the particular data value, that is, the size of the object pointed to by getvalue. For text data format this is the same as strlen(). For binary format this is essential information. Note that one should not rely on fsize to obtain the actual data length.

```
res:nparams()
```

Returns the number of parameters of a prepared statement.

```
res:paramtype(paramNumber)
```

Returns the data type of the indicated statement parameter. Parameter numbers start at 1.

This function is only useful when inspecting the result of describePrepared. For other types of queries it will return zero.

Retrieving other result information

These functions are used to extract other information from result objects.

```
res:cmdStatus()
```

Returns the command status tag from the SQL command that generated the result.

Commonly this is just the name of the command, but it might include additional data such as the number of rows processed.

```
res:cmdTuples()
```

Returns the number of rows affected by the SQL command.

This function returns a string containing the number of rows affected by the SQL statement that generated the result. This function can only be used following the execution of a SELECT, CREATE TABLE AS, INSERT, UPDATE, DELETE, MOVE, FETCH, or COPY statement, or an EXECUTE of a prepared query that contains an INSERT, UPDATE, or DELETE statement. If the command that generated the result was anything else, cmdTuples returns an empty string.

```
res:oidValue()
```

Returns the OID of the inserted row, if the SQL command was an INSERT that inserted exactly one row into a table that has OIDs, or a EXECUTE of a prepared query containing a suitable INSERT statement. Otherwise, this function returns InvalidOid. This function will also return InvalidOid if the table affected by the INSERT statement does not contain OIDs.

```
res:oidStatus()
```

This function is deprecated in favor of oidValue and is not thread-safe. It returns a string with the OID of the inserted row, while oidValue returns the OID value.

Escaping strings for inclusion in SQL commands

```
conn:escapeLiteral(str)
```

escapeLiteral escapes a string for use within an SQL command. This is useful when inserting data values as literal constants in SQL commands. Certain characters (such as quotes and backslashes)

must be escaped to prevent them from being interpreted specially by the SQL parser. escapeLiteral performs this operation.

escapeLiteral returns an escaped version of the str parameter. The return string has all special characters replaced so that they can be properly processed by the PostgreSQL string literal parser. A terminating zero byte is also added. The single quotes that must surround PostgreSQL string literals are included in the result string.

On error, escapeLiteral returns nil and a suitable message is stored in the conn object.

Note that it is not necessary nor correct to do escaping when a data value is passed as a separate parameter in execParams or its sibling routines.

```
conn:escapeString(str)
```

Escape escapes string literals, much like escapeLiteral.

```
conn:escapeIdentifier(str)
```

escapeIdentifier escapes a string for use as an SQL identifier, such as a table, column, or function name. This is useful when a user-supplied identifier might contain special characters that would otherwise not be interpreted as part of the identifier by the SQL parser, or when the identifier might contain upper case characters whose case should be preserved.

escapeIdentifier returns a version of the str parameter escaped as an SQL identifier. The return string has all special characters replaced so that it will be properly processed as an SQL identifier. A terminating zero byte is also added. The return string will also be surrounded by double quotes.

On error, escapeIdentifier returns nil and a suitable message is stored in the conn object.

```
conn:escapeBytea(str)
```

Escapes binary data for use within an SQL command with the type `bytea`. As with escapeString, this is only used when inserting data directly into an SQL command string.

Certain byte values must be escaped when used as part of a `bytea` literal in an SQL statement. escapeBytea escapes bytes using either hex encoding or backslash escaping.

On error, nil is returned, and a suitable error message is stored in the `conn` object. Currently, the only possible error is insufficient memory for the result string.

```
unescapeBytea(str)
```

Converts a string representation of binary data into binary data — the reverse of escapeBytea. This is needed when retrieving bytea data in text format, but not when retrieving it in binary format.

Asynchronous command processing

The exec function is adequate for submitting commands in normal, synchronous applications. It has a few deficiencies, however, that can be of importance to some users:

- exec waits for the command to be completed. The application might have other work to do (such as maintaining a user interface), in which case it won't want to block waiting for the response.
- Since the execution of the client application is suspended while it waits for the result, it is hard for the application to decide that it would like to try to cancel the ongoing command. (It can be done from a signal handler, but not otherwise.)
- exec can return only one result object. If the submitted command string contains multiple SQL commands, all but the last result are discarded by exec.
- exec always collects the command's entire result, buffering it in a single result. While this simplifies error-handling logic for the application, it can be impractical for results containing many rows.

Applications that do not like these limitations can instead use the underlying functions that exec is built from: sendQuery and getResult. There are also sendQueryParams, sendPrepare, sendQueryPrepared, sendDescribePrepared, and sendDescribePortal, which can be used with getResult to duplicate the functionality of execParams, prepare, execPrepared, describePrepared, and describePortal respectively.

```
conn:sendQuery(command)
```

Submits a command to the server without waiting for the result(s). true is returned if the command was successfully dispatched and false if not (in which case, use errorMessage to get more information about the failure).

After successfully calling sendQuery, call getResult one or more times to obtain the results. sendQuery cannot be called again (on the same connection) until getResult has returned a null pointer, indicating that the command is done.

```
conn:sendQueryParams(command [[, param] ..])
```

Submits a command and separate parameters to the server without waiting for the result(s).

This is equivalent to sendQuery except that query parameters can be specified separately from the query string. The function's parameters are handled identically to execParams. Like execParams, it will not work on 2.0-protocol connections, and it allows only one command in the query string.

```
conn:sendPrepare(stmtName, query [[, param] ..])
```

Sends a request to create a prepared statement with the given parameters, without waiting for completion.

This is an asynchronous version of prepare: it returns true if it was able to dispatch the request, and false if not. After a successful call, call PQgetResult to determine whether the server successfully created the prepared statement. The function's parameters are handled identically to prepare. Like prepare, it will not work on 2.0-protocol connections.

```
conn:sendQueryPrepared(stmtName [[, param] ..])
```

Sends a request to execute a prepared statement with given parameters, without waiting for the result(s).

This is similar to sendQueryParams, but the command to be executed is specified by naming a previously-prepared statement, instead of giving a query string. The function's parameters are handled identically to execPrepared. Like execPrepared, it will not work on 2.0-protocol connections.

```
conn:sendDescribePrepared(stmtName)
```

Submits a request to obtain information about the specified prepared statement, without waiting for completion.

This is an asynchronous version of describePrepared: it returns true if it was able to dispatch the request, and false if not. After a successful call, call getResult to obtain the results. The function's parameters are handled identically to describePrepared. Like describePrepared, it will not work on 2.0-protocol connections.

```
conn:sendDescribePortal(portalName)
```

Submits a request to obtain information about the specified portal, without waiting for completion.

This is an asynchronous version of describePortal: it returns true if it was able to dispatch the request, and false if not. After a successful call, call getResult to obtain the results. The function's parameters are handled identically to describePortal. Like describePortal, it will not work on 2.0-protocol connections.

```
conn:getResult()
```

Waits for the next result from a prior sendQuery, sendQueryParams, sendPrepare, sendQueryPrepared, sendDescribePrepared, or sendDescribePortal call, and returns it. nil is returned when the command is complete and there will be no more results.

getResult must be called repeatedly until it returns nil, indicating that the command is done. (If called when no command is active, getResult will just return nil at once.) Each non-nil result from getResult should be processed using the same result accessor functions previously described. Note that getResult will block only if a command is active and the necessary response data has not yet been read by consumeInput.

Note: Even when resultStatus indicates a fatal error, getResult should be called until it returns a nil, to allow pgsql to process the error information completely.

Using sendQuery and getResult solves one of exec's problems: If a command string contains multiple SQL commands, the results of those commands can be obtained individually. (This allows a simple form of overlapped processing, by the way: the client can be handling the results of one command while the server is still working on later queries in the same command string.)

By itself, calling getResult will still cause the client to block until the server completes the next SQL command. This can be avoided by proper use of two more functions:

```
conn:consumeInput()
```

If input is available from the server, consume it.

consumeInput normally returns true indicating no error, but returns false if there was some kind of trouble (in which case errorMessage can be consulted). Note that the result does not say whether any input data was actually collected. After calling consumeInput, the application can check isBusy and/or notifies to see if their state has changed.

consumeInput can be called even if the application is not prepared to deal with a result or notification just yet. The function will read available data and save it in a buffer, thereby causing a select() read-ready indication to go away. The application can thus use consumeInput to clear the select() condition immediately, and then examine the results at leisure.

```
conn:isBusy()
```

Returns true if a command is busy, that is, getResult would block waiting for input. A false return indicates that getResult can be called with assurance of not blocking.

isBusy will not itself attempt to read data from the server; therefore PQconsumeInput must be invoked first, or the busy state will never end.

A typical application using these functions will have a main loop that uses select() or poll() to wait for all the conditions that it must respond to. One of the conditions will be input available from the server, which in terms of select() means readable data on the file descriptor identified by socket. When the main loop detects input ready, it should call consumeInput to read the input. It can then call isBusy, followed by getResult if isBusy returns false. It can also call notifies to detect NOTIFY messages.

A client that uses sendQuery/getResult can also attempt to cancel a command that is still being processed by the server. But regardless of the return value of cancel, the application must continue with the normal result-reading sequence using getResult. A successful cancellation will simply cause the command to terminate sooner than it would have otherwise.

By using the functions described above, it is possible to avoid blocking while waiting for input from the database server. However, it is still possible that the application will block waiting to send output to the server. This is relatively uncommon but can happen if very long SQL commands or

data values are sent. (It is much more probable if the application sends data via COPY IN, however.) To prevent this possibility and achieve completely nonblocking database operation, the following additional functions can be used.

```
conn:setnonblocking(arg)
```

Sets the nonblocking status of the connection.

Sets the state of the connection to nonblocking if arg is true, or blocking if arg is false. Returns true if OK, false if error.

In the nonblocking state, calls to sendQuery, putline, putnbytes, and endcopy will not block but instead return an error if they need to be called again.

Note that exec does not honor nonblocking mode; if it is called, it will act in blocking fashion anyway.

```
conn:isnonblocking()
```

Returns the blocking status of the database connection.

Returns true if the connection is set to nonblocking mode and false if blocking.

```
conn:flush()
```

Attempts to flush any queued output data to the server. Returns true if successful (or if the send queue is empty), nil if it failed for some reason, or false if it was unable to send all the data in the send queue yet (this case can only occur if the connection is nonblocking).

After sending any command or data on a nonblocking connection, call PQflush. If it returns false, wait for the socket to be write-ready and call it again; repeat until it returns true. Once PQflush returns true wait for the socket to be read-ready and then read the response as described above.

Retrieving Query Results Row-By-Row

Ordinarily, pgsql collects a SQL command's entire result and returns it to the application as a single `result`. This can be unworkable for commands that return a large number of rows. For such cases, applications can use sendQuery and getResult in single-row mode. In this mode, the result row(s) are returned to the application one at a time, as they are received from the server.

To enter single-row mode, call setSingleRowMode immediately after a successful call of sendQuery (or a sibling function). This mode selection is effective only for the currently executing query. Then call getResult repeatedly, until it returns nil. If the query returns any rows, they are returned as individual `result` objects, which look like normal query results except for having status code `PGRES_SINGLE_TUPLE` instead of `PGRES_TUPLES_OK`. After the last row, or immediately if the query returns zero rows, a zero-row object with status `PGRES_TUPLES_OK` is returned; this is the signal that no more rows will arrive. (But note that it is still necessary to continue calling getResult until it

returns nil.) All of these `result` objects will contain the same row description data (column names, types, etc) that an ordinary `result` object for the query would have.

```
conn:setSingleRowMode()
```

Select single-row mode for the currently-executing query.

This function can only be called immediately after sendQuery or one of its sibling functions, before any other operation on the connection such as consumeInput or getResult. If called at the correct time, the function activates single-row mode for the current query and returns true. Otherwise the mode stays unchanged and the function returns false. In any case, the mode reverts to normal after completion of the current query.

Canceling queries in progress

```
conn:cancel()
```

Requests that the server abandon processing of the current command.

Asynchronous notification functions

PostgreSQL offers asynchronous notification via the LISTEN and NOTIFY commands. A client session registers its interest in a particular notification channel with the LISTEN command (and can stop listening with the UNLISTEN command). All sessions listening on a particular channel will be notified asynchronously when a NOTIFY command with that channel name is executed by any session. A payload string can be passed to communicate additional data to the listeners.

pgsql applications submit LISTEN, UNLISTEN, and NOTIFY commands as ordinary SQL commands. The arrival of NOTIFY messages can subsequently be detected by calling notifies.

```
conn:notifies()
```

The function notifies returns the next notification from a list of unhandled notification messages received from the server. It returns nil if there are no pending notifications. Once a notification is returned from notifies, it is considered handled and will be removed from the list of notifications.

notifies does not actually read data from the server; it just returns messages previously absorbed by another pgsql function.

A good way to check for NOTIFY messages when you have no useful commands to execute is to call consumeInput, then check notifies. You can use select() to wait for data to arrive from the server, thereby using no CPU power unless there is something to do. (See socket to obtain the file descriptor number to use with select().) Note that this will work OK whether you submit commands with sendQuery/getResult or simply use exec. You should, however, remember to check notifies after each getResult or exec, to see if any notifications came in during the processing of the command.

Functions associated with the COPY command

The COPY command in PostgreSQL has options to read from or write to the network connection used by pgsql. The functions described in this section allow applications to take advantage of this capability by supplying or consuming copied data.

The overall process is that the application first issues the SQL COPY command via exec or one of the equivalent functions. The response to this (if there is no error in the command) will be a result object bearing a status code of PGRES_COPY_OUT or PGRES_COPY_IN (depending on the specified copy direction). The application should then use the functions of this section to receive or transmit data rows. When the data transfer is complete, another result object is returned to indicate success or failure of the transfer. Its status will be PGRES_COMMAND_OK for success or PGRES_FATAL_ERROR if some problem was encountered. At this point further SQL commands can be issued via exec. (It is not possible to execute other SQL commands using the same connection while the COPY operation is in progress.)

If a COPY command is issued via exec in a string that could contain additional commands, the application must continue fetching results via getResult after completing the COPY sequence. Only when PQgetResult returns NULL is it certain that the PQexec command string is done and it is safe to issue more commands.

The functions of this section should be executed only after obtaining a result status of PGRES_COPY_OUT or PGRES_COPY_IN from exec or getResult.

A result object bearing one of these status values carries some additional data about the COPY operation that is starting. This additional data is available using functions that are also used in connection with query results:

```
res:nfields()
```

Returns the number of columns (fields) to be copied.

```
res:binaryTuples()
```

false indicates the overall copy format is textual (rows separated by newlines, columns separated by separator characters, etc). true indicates the overall copy format is binary. See COPY for more information.

```
res:fformat()
```

Returns the format code (0 for text, 1 for binary) associated with each column of the copy operation. The per-column format codes will always be zero when the overall copy format is textual, but the binary format can support both text and binary columns. (However, as of the current implementation of COPY, only binary columns appear in a binary copy; so the per-column formats always match the overall format at present.)

Functions for sending COPY data

These functions are used to send data during COPY FROM STDIN. They will fail if called when the connection is not in COPY_IN state.

```
conn:putCopyData(buffer)
```

Sends data to the server during COPY_IN state.

Transmits the COPY data in the specified buffer, to the server. The result is true if the data was sent, false if it was not sent because the attempt would block (this case is only possible if the connection is in nonblocking mode), or nil if an error occurred. (Use errorMessage to retrieve details if the return value is nil. If the value is zero, wait for write-ready and try again.)

The application can divide the COPY data stream into buffer loads of any convenient size. Buffer-load boundaries have no semantic significance when sending. The contents of the data stream must match the data format expected by the COPY command.

```
conn:putCopyEnd(errormsg)
```

Sends end-of-data indication to the server during COPY_IN state.

Ends the COPY_IN operation successfully if errormsg is nil. If errormsg is not nil then the COPY is forced to fail, with the string pointed to by errormsg used as the error message. (One should not assume that this exact error message will come back from the server, however, as the server might have already failed the COPY for its own reasons. Also note that the option to force failure does not work when using pre-3.0-protocol connections.)

The result is true if the termination data was sent, false if it was not sent because the attempt would block (this case is only possible if the connection is in nonblocking mode), or nil if an error occurred. (Use PQerrorMessage to retrieve details if the return value is nil. If the value is zero, wait for write-ready and try again.)

After successfully calling putCopyEnd, call getResult to obtain the final result status of the COPY command. One can wait for this result to be available in the usual way. Then return to normal operation.

Functions for receiving COPY data

These functions are used to receive data during COPY TO STDOUT. They will fail if called when the connection is not in COPY_OUT state.

```
conn:getCopyData(async)
```

Receives data from the server during COPY_OUT state.

Attempts to obtain another row of data from the server during a COPY. Data is always returned one

data row at a time; if only a partial row is available, it is not returned.

When a row is successfully returned, the return value is the data in the row as a string. A result of false indicates that the COPY is still in progress, but no row is yet available (this is only possible when async is true). A result of true indicates that the COPY is done. A result of nil indicates that an error occurred (consult errorMessage for the reason).

When async is true, getCopyData will not block waiting for input; it will return false if the COPY is still in progress but no complete row is available. (In this case wait for read-ready and then call consumeInput before calling getCopyData again.) When async is false, getCopyData will block until data is available or the operation completes.

After getCopyData returns true, call getResult to obtain the final result status of the COPY command. One can wait for this result to be available in the usual way. Then return to normal operation.

Control functions

```
conn:clientEncoding()
```

Returns the client encoding.

```
conn:setClientEncoding(encoding)
```

Sets the client encoding.

```
conn:setErrorVerbosity()
```

Determines the verbosity of messages returned by errorMessage and resultErrorMessage.

setErrorVerbosity sets the verbosity mode, returning the connection's previous setting. In TERSE mode, returned messages include severity, primary text, and position only; this will normally fit on a single line. The default mode produces messages that include the above plus any detail, hint, or context fields (these might span multiple lines). The VERBOSE mode includes all available fields. Changing the verbosity does not affect the messages available from already-existing result objects, only subsequently-created ones.

```
conn:trace(file)
```

Enables tracing of the client/server communication to a debugging file stream obtaining via io.open().

```
conn:untrace()
```

Disables tracing started by conn:trace().

Miscellaneous functions

```
conn:encryptPassword(passwd, user [, algorithm])
```

Prepares the encrypted form of a PostgreSQL password.

This function is intended to be used by client applications that wish to send commands like ALTER USER joe PASSWORD 'pwd'. It is good practice not to send the original cleartext password in such a command, because it might be exposed in command logs, activity displays, and so on. Instead, use this function to convert the password to encrypted form before it is sent.

The passwd and user arguments are the cleartext password, and the SQL name of the user it is for. algorithm specifies the encryption algorithm to use to encrypt the password. Currently supported algorithms are md5 and scram-sha-256 (on and off are also accepted as aliases for md5, for compatibility with older server versions). Note that support for scram-sha-256 was introduced in PostgreSQL version 10, and will not work correctly with older server versions. If algorithm is nil or absent, this function will query the server for the current value of the password_encryption setting. That can block, and will fail if the current transaction is aborted, or if the connection is busy executing another query. If you wish to use the default algorithm for the server but want to avoid blocking, query password_encryption yourself before calling conn:encryptPassword(), and pass that value as the algorithm.

The return value is a string. The caller can assume the string doesn't contain any special characters that would require escaping. On error, conn:encryptPassword() returns nil, and a suitable message is stored in the connection object.

```
pgsql.encryptPassword()
```

Prepares the md5-encrypted form of a PostgreSQL password.

pgsql.encryptPassword() is an older, deprecated version of conn:encryptPasswod(). The difference is that encryptPassword() does not require a connection object, and md5 is always used as the encryption algorithm.

```
pgsql.libVersion()
```

Return the version of the underlying libpq that is being used.

The result of this function can be used to determine, at run time, if specific functionality is available in the currently loaded version of libpq. The function can be used, for example, to determine which connection options are available for connectdb or if the hex bytea output added in PostgreSQL 9.0 is supported.

The number is formed by converting the major, minor, and revision numbers into two-decimal-digit numbers and appending them together. For example, version 9.1 will be returned as 90100, and version 9.1.2 will be returned as 90102 (leading zeroes are not shown).

Notice processing

Notice and warning messages generated by the server are not returned by the query execution functions, since they do not imply failure of the query. Instead they are passed to a notice handling function, and execution continues normally after the handler returns. The default notice handling function prints the message on stderr, but the application can override this behavior by supplying its own handling function.

For historical reasons, there are two levels of notice handling, called the notice receiver and notice processor. The default behavior is for the notice receiver to format the notice and pass a string to the notice processor for printing. However, an application that chooses to provide its own notice receiver will typically ignore the notice processor layer and just do all the work in the notice receiver.

```
conn:setNoticeReceiver()
```

See below.

```
conn:setNoticeProcessor()
```

The function setNoticeReceiver sets or examines the current notice receiver for a connection object. Similarly, setNoticeProcessor sets or examines the current notice processor.

Each of these functions returns the previous notice receiver or processor function pointer, and sets the new value. If you supply a null function pointer, no action is taken, but the current pointer is returned.

When a notice or warning message is received from the server, or generated internally by libpq, the notice receiver function is called. It is passed the message in the form of a PGRES_NONFATAL_ERROR result. (This allows the receiver to extract individual fields using resultErrorField, or the complete preformatted message using resultErrorMessage.) The same void pointer passed to setNoticeReceiver is also passed. (This pointer can be used to access application-specific state if needed.)

The default notice receiver simply extracts the message (using resultErrorMessage) and passes it to the notice processor.

The notice processor is responsible for handling a notice or warning message given in text form. It is passed the string text of the message (including a trailing newline), plus a void pointer that is the same one passed to setNoticeProcessor. (This pointer can be used to access application-specific state if needed.)

Once you have set a notice receiver or processor, you should expect that that function could be called as long as either the conn object or result objects made from it exist. At creation of a result, the conn's current notice handling pointers are copied into the result for possible use by functions like getvalue.

SSL Support

```
pgsql.initOpenSSL(do_ssl, do_crypt)
```

Allows applications to select which security libraries to initialize.

When do_ssl is true, luapgsql will initialize the OpenSSL library before first opening a database connection. When do_crypto is true, the libcrypto library will be initialized. By default (if initOpenSSL is not called), both libraries are initialized. When SSL support is not compiled in, this function is present but does nothing.

If your application uses and initializes either OpenSSL or its underlying libcrypto library, you must call this function with false for the appropriate parameter(s) before first opening a database connection. Also be sure that you have done that initialization before opening a database connection.

Large objects

```
conn:lo_create(lobjId)
```

Creates a new large object. The OID to be assigned can be specified by lobjId; if so, failure occurs if that OID is already in use for some large object. If lobjId is InvalidOid (zero) then lo_create assigns an unused OID (this is the same behavior as lo_creat). The return value is the OID that was assigned to the new large object, or InvalidOid (zero) on failure.

lo_create is new as of PostgreSQL 8.1; if this function is run against an older server version, it will fail and return InvalidOid.

To import an operating system file as a large object, call

```
conn:lo_import(filename)
```

filename specifies the operating system name of the file to be imported as a large object. The return value is the OID that was assigned to the new large object, or InvalidOid (zero) on failure. Note that the file is read by the client interface library, not by the server; so it must exist in the client file system and be readable by the client application.

The function

```
conn:lo_import_with_oid(filename, lobjId)
```

also imports a new large object. The OID to be assigned can be specified by lobjId; if so, failure occurs if that OID is already in use for some large object. If lobjId is InvalidOid (zero) then lo_import_with_oid assigns an unused OID (this is the same behavior as lo_import). The return value is the OID that was assigned to the new large object, or InvalidOid (zero) on failure.

lo_import_with_oid is new as of PostgreSQL 8.4 and uses lo_create internally which is new in 8.1; if this function is run against 8.0 or before, it will fail and return InvalidOid.

To export a large object into an operating system file, call

```
conn:lo_export(lobjId, filename)
```

The lobjId argument specifies the OID of the large object to export and the filename argument specifies the operating system name of the file. Note that the file is written by the client interface library, not by the server. Returns true on success, false on failure.

To open an existing large object for reading or writing, call

```
fd = conn:lo_open(lobjId, mode)
```

The lobjId argument specifies the OID of the large object to open. The mode bits control whether the object is opened for reading (INV_READ), writing (INV_WRITE), or both. (These symbolic constants are defined in the PostgreSQL header file libpq/libpq-fs.h.) lo_open returns a (non-negative) large object descriptor for later use in lo:read, lo:write, lo:lseek, lo:lseek64, lo:tell, lo:tell64, lo:truncate, lo:truncate64, and lo:close. The descriptor is only valid for the duration of the current transaction. On failure, nil is returned.

The server currently does not distinguish between modes INV_WRITE and INV_READ INV_WRITE: you are allowed to read from the descriptor in either case. However there is a significant difference between these modes and INV_READ alone: with INV_READ you cannot write on the descriptor, and the data read from it will reflect the contents of the large object at the time of the transaction snapshot that was active when lo_open was executed, regardless of later writes by this or other transactions. Reading from a descriptor opened with INV_WRITE returns data that reflects all writes of other committed transactions as well as writes of the current transaction. This is similar to the behavior of REPEATABLE READ versus READ COMMITTED transaction modes for ordinary SQL SELECT commands.

The function

```
conn:lo_write(fd, buf)
```

writes all bytes from buf to a large object. The number of bytes actually written is returned (in the current implementation, this will always equal #buf unless there is an error). In the event of an error, the return value is -1.

Although the len parameter is declared as size_t, this function will reject length values larger than INT_MAX. In practice, it’s best to transfer data in chunks of at most a few megabytes anyway.

The function

```
conn:lo_read(fd, len)
```

reads up to len bytes from large object descriptor fd into buf (which must be of size len). The fd argument must have been returned by a previous lo_open. The number of bytes actually read is returned; this will be less than len if the end of the large object is reached first. In the event of an error, the return value is -1.

Although the len parameter is declared as size_t, this function will reject length values larger than INT_MAX. In practice, it's best to transfer data in chunks of at most a few megabytes anyway.

To change the current read or write location associated with a large object descriptor, call

```
conn:lo_lseek(fd, offset, whence)
```

This function moves the current location pointer for the large object descriptor identified by fd to the new location specified by offset. The valid values for whence are SEEK_SET (seek from object start), SEEK_CUR (seek from current position), and SEEK_END (seek from object end). The return value is the new location pointer, or -1 on error.

When dealing with large objects that might exceed 2GB in size, instead use

```
conn:lo_lseek64(fd, offset, whence)
```

This function has the same behavior as lo:lseek, but it can accept an offset larger than 2GB and/or deliver a result larger than 2GB. Note that l:lseek will fail if the new location pointer would be greater than 2GB.

conn:lo_lseek64 is new as of PostgreSQL 9.3. If this function is run against an older server version, it will fail and return -1.

To obtain the current read or write location of a large object descriptor, call

```
conn:lo_tell(fd)
```

If there is an error, the return value is -1.

When dealing with large objects that might exceed 2GB in size, instead use

```
conn:lo_tell64(fd)
```

This function has the same behavior as lo_tell, but it can deliver a result larger than 2GB. Note that lo_tell will fail if the current read/write location is greater than 2GB.

conn:lo_tell64 is new as of PostgreSQL 9.3. If this function is run against an older server version, it will fail and return -1.

To truncate a large object to a given length, call

```
conn:lo_truncate(fd, len)
```

This function truncates the large object to length len. If len is greater than the large object's current length, the large object is extended to the specified length with null bytes ('\0'). On success, lo:truncate returns zero. On error, the return value is -1.

The read/write location associated with the descriptor fd is not changed.

Although the len parameter is declared as size_t, lo_truncate will reject length values larger than INT_MAX.

When dealing with large objects that might exceed 2GB in size, instead use

```
conn:lo_truncate64(fd, len)
```

This function has the same behavior as lo_truncate, but it can accept a len value exceeding 2GB.

conn:lo_truncate64 is new as of PostgreSQL 8.3; if this function is run against an older server version, it will fail and return -1.

conn:lo_truncate64 is new as of PostgreSQL 9.3; if this function is run against an older server version, it will fail and return -1.

A large object descriptor can be closed by calling

```
conn:lo_close(fd)
```

To remove a large object from the database, call

```
conn:lo_unlink(lobjid)
```

The lobjId argument specifies the OID of the large object to remove. Returns 1 if successful, -1 on failure.

Notify functions

```
notify:relname()
```

Return the relname field of a notification.

```
notify:pid()
```

Return the pid field of a notification.

```
notify:extra()
```

Return the extra data field of a notification.

Miscellaneous

mqLua

mqLua is a "different" way to execute Lua code. It combines the Lua language with POSIX threads, and, 0MQ (zeromq.orq). mqLua comes as a binary, called "mqlua", which takes a filename of a Lua program as argument. This Lua program is meant to "orchestrate" a network of so called nodes: independent Lua states each running in its own thread with the ability to communicate over message queues. By means of POSIX threads, the Lua states run truly in parallel, using all available CPU cores. By using 0MQ message queues, the Lua states can communicate with other Lua states (or, in fact, any program supporting 0MQ) running in a different thread in the same process, with other Lua states running in a different process on the same machine, or even with Lua states running on different machines.

Since 0MQ itself is language agnostic, this mechanism can be used to communicate with software written in different languages as well. 0MQ bindings exist for almost any programming language.

Using mqLua

Besides running Lua programs and providing the same standard libraries as the "lua" binary does, mqLua offers two non-standard modules: **node** and **zmq** for the creation and management of nodes and for communicating over message queues.

A new Node (i.e. a Lua state running in its own thread) is created using node.create():

```
local n = node.create('worker.lua', 'bee', 42)
```

This will create a new thread with a new Lua state, running the chunk found in the file "worker.lua", passing the arguments "bee" and 42 in ... to worker.lua.

Nodes can use the "zmq" module to communicate with each other. So it is possible to run multiple independent Lua threads in one process and have them communicate with each other, or with Nodes in differents processes on the same machine, or with Nodes running on a remote machines.

Integrating mqLua

The mqLua source code has been organized in a way that faciliates the integration with software written in the C or C++ language. Basically one has to compile (and link) the node.c and zmq.c files to the software that is to use mqLua and link the software with pthreads and libzmq. The file main.c can serve as an example how to glue things together.

In the future the author might provide mqLua as a simple library instead of a binary.

References

ZeroMQ, http://zeromq.org/