The cqueues User Guide

for composing

Socket, Signal, Thread, & File Change Messaging

on

Linux, OS X, Solaris, FreeBSD, NetBSD, & OpenBSD

with

William Ahern

July 21, 2019

Contents

1 Dependencies

1.1 Operating Systems

`cqueues` heavily relies on a modern POSIX environment. But the fundamental premise is to build on the new but non-standard polling facilities provided by contemporary Unix environments. Specifically, BSD `kqueue`, Linux `epoll`, and Solaris Event Ports.

`cqueues` should work on recent versions of Linux, OS X, Solaris, NetBSD, FreeBSD, OpenBSD, and derivatives. The only other possible candidate is AIX, if and when support for AIX's `pollset` interface is added to the embedded "kpoll" library.

1.1.1 ¬ Microsoft Windows

Microsoft Windows support is basically out of the question[1], for far too many reasons to put here. Aside from the more technical reasons, Windows I/O and networking programming interfaces have a fundamentally different character than on Unix. Unix historically relies on readiness polling, while Windows uses event completion callbacks. There are strengths and weaknesses to each approach. Trying to paper over the chasm between the two approaches invariably results in a framework with the strengths of neither and the weaknesses of both. The purpose of `cqueues` is to leverage the strengths of polling as well as address the weaknesses.

1.2 Libraries

1.2.1 LuaJIT, Lua 5.2, Lua 5.3

`cqueues` principally targets Lua 5.2 and above. Some semantics are not fully portable to Lua 5.1 due to 5.1 not supporting yielding from metamethods and iterators. LuaJIT does not have this handicap.

1.2.2 OpenSSL

The `cqueues socket` module provides seamless SSL/TLS support using OpenSSL.

Comprehensive bindings for certificate and key management are provided in the companion `openssl` module, `luaossl`.

[1] I have been toying with the idea of using an fd_set in-place of a pollable descriptor on Windows, and taking the union of all fd_sets when polling.

1.2.3 pthreads

`cqueues` provides an optional threading module, using POSIX threads.[2] Internally it consistently uses thread-safe routines when built with either the _REENTRANT or _THREAD_SAFE feature macros, such as `pthread_sigmask` instead of `sigprocmask`. Thread support is enabled by default.

Linking Note that on some systems, such as NetBSD and FreeBSD, the loading application must be linked against pthreads (using -lpthread or -pthread). It is not enough for the `cqueues` module to pull in the dependency at load time. In particular, if using the stock Lua interpreter, it must have been linked against pthreads at build time. Add the appropriate linker flag to MYLIBS in lua-5.2.x/src/Makefile.

OpenBSD OpenBSD 5.1 threading is completely *fubar*, especially with regard to signals, because of OpenBSD's transition to kernel threading. If using OpenBSD, be sure to compile *without* the thread-safe macros predefined, especially if using `cqueues.signal`.

1.3 Compilers

The source code is mostly ISO C99 compliant, and even more so with regards to ISO C11. But regardless of standards conformance, it aims to build cleanly with the native compiler for each targeted platform. It currently builds with recent versions of GCC, clang, and SunPro.

Patches are welcome to silence compiler diagnostics.

1.4 GNU Make

The Makefile requires GNU Make, usually installed as gmake on platforms other than Linux or OS X. The actual `Makefile` proxies to `GNUmakefile`. As long as `gmake` is installed on non-GNU systems you can invoke your system's `make`.

[2] Building without threading enabled is not well tested.

2 Installation

Refer to the included `README.md` for up-to-date build and installation instructions.

3 Usage

3.1 Conventions

3.1.1 Polling

`cqueues` works through a simple protocol. When a coroutine yields to its parent `cqueues` controller, it can pass one or more objects. These objects are introspected for three methods: `:pollfd`, `:events`, and `:timeout`. These methods generate the parameters for installing descriptor and timeout events. When one of these events fires, `cqueues` will resume the coroutine, passing the relevant objects which were interested in the triggered event. It's analogous to calling Unix `poll`, and in fact the routine `cqueues.poll` is provided as a wrapper for `coroutine.yield`.[1]

`:pollfd()`

The `:pollfd` method should return a descriptor integer or nil. This descriptor must remain in existence until the owner object is garbage collected, `cqueues.cancel` is used, the coroutine executes one additional yield/resume cycle (so the old descriptor is expired from the descriptor queue), or until after the coroutine exits. If the descriptor is closed prematurely, the kernel will remove it from the internal descriptor queue, bringing it out of sync with the controller, and probably causing `cqueues:step` to return EBADF or ENOENT errors.

Alternatively, `:pollfd` may return a condition variable object, or the member field may itself be a condition variable instead of a function. Similarly, the `.pollfd` member field may be an integer descriptor. This permits user code to create *ad hoc* pollable objects.

`:events()`

The `:events` method should return a string or nil. `cqueues` searches the string for the flags 'r' and 'w', which describe the events to associate with the descriptor—respectively, `POLLIN` and `POLLOUT`.

The flag 'p' may also be specified, describing `POLLPRI`. However, `POLLPRI` is not supported for kqueue-based environments.[2]

[1] This wrapper can also detect if the current coroutine was resumed by a controller, and if not chain yield calls—with the cooperation of a `cqueues.resume`—until a controller is reached.

[2] OS X's `EV_OOBAND` is only useable as an output flag. DragonflyBSD's `EVFILT_EXCEPT` maps well and will be supported in a future release.

Alternatively, the events may be a literal integer value of the logical-OR of the system event values `POLLIN`, `POLLOUT`, `POLLPRI`, etc. However, specifying any events beyond the three discussed is not currently supported and may lead to unexpected behavior.

`:timeout()`

The `:timeout` should return a number or nil. This schedules an independent timeout event. To effect a simple one second timeout, you can do

```
cqueues.poll({ timeout = function() return 1.0 end })
```

which is equivalent to the shortcut

```
cqueues.poll(1.0)
```

Instantiated `cqueues` objects implement all three methods.[3] In particular, this means that you can stack `cqueues`, or poll on a `cqueues` object using some other event loop library. Each `cqueues` object is entirely self-contained, without any global state.

3.1.2 ¬ Globals

Like the core controller module, other `cqueues` modules adhere to a *no global side effects* discipline. In particular, this means

- no global process variables;
- no signal handling gimmicks—like the pipe trick—which could conflict with other components of your application[4];
- consistent use of thread-safe function variants; and
- consistent use of O_CLOEXEC and similar flags to eliminate or reduce `fork` + `exec` races in threaded applications.

3.1.3 Errors

The usual behavior is for errors to be returned directly. But see `socket.onerror`. If a routine is specified to return an object or string, nil is returned; if a boolean, false is returned. In both cases, these are usually followed by a numeric error code. Thus, if a routine is specified to return two values on success, then on error three values are returned, the first two nil or false, and the third an error code.

`cqueues` is a relatively low-level component library. In almost all cases errors will be system errors, returned as numeric error codes for easy and efficient comparison. For example, attempting to create a UNIX domain socket with `socket.listen` in a directory without sufficient permissions might return 'nil, `EACCES`'.

[3] `:pollfd` returns the internal `kqueue`, `epoll`, or Ports descriptor; `:events` returns "r"; and `:timeout` returns the time to the next internal timeout event.

[4] The `cqueues.thread` module ensures threads are started with a filled signal mask.

`EAGAIN`

`cqueues` modules are implemented in both C and Lua. The C routines never yield, and always return recoverable errors directly. Most C routines are wrapped—and methods interposed—with Lua functions. These Lua functions usually poll when `EAGAIN` is encountered and retry the C routine on resumption. Few methods will return `EAGAIN` directly.

`ETIMEDOUT`

This error value is usually seen when a timeout is specified by the caller of a logically synchronous method. The method will normally yield and poll if the operation cannot be completed immediately, but if the timeout expires then it will return a failure with `ETIMEDOUT`.

`EPIPE`

In Unix `EPIPE` is only encountered when attempting to write to a closed pipe or socket. In `cqueues` `EPIPE` is used to signal both EOF and a closed output stream.[5] The low-level I/O method `socket:recv`, for example, returns `EPIPE` on EOF. In other cases, as with `socket:read`, EOF is not an error condition.

`EBADF`

This error commonly occurs in asynchronous applications, which are especially prone to bugs related to their complex state management. With Lua code using the `cqueues` APIs, `EBADF` should never be encountered. When it does occur, it's a sure sign of a bug somewhere in the parent application or an extension module and—hopefully—not `cqueues`.

The Future

The idiomatic protocol for returning errors in Lua is a string representation followed by the integer errno number. This is how Lua's `io` and `file` modules behave. The original concern was that this would be too wasteful for a networking library, where "errors" like EAGAIN, ETIMEDOUT, and EPIPE are common and not very exceptional. Copying even small strings into the Lua VM is somewhat costly. However, in the future the API may be configurable to use the Lua-idiomatic protocol by default, using upvalue memoization to minimize the cost of returning string representations.

In the meantime, the auxiliary routines `auxlib.assert` and `auxlib.fileresult` can be used to explicitly achieve the idiomatic behavior.

3.2 Modules

3.2.1 `cqueues`

`cqueues.VENDOR`

String describing the vendor, e.g. william@25thandClement.com. If you fork this project please change this string so I don't receive unwarranted scorn or praise.

[5] In some situations, such as with SSL/TLS, a read attempt might require a write, anyhow. Expanding the scope of EPIPE simplifies the logic required to handle various I/O failures.

`cqueues.VERSION`

Number describing the running version, formatted as YYYYMMDD. Official releases are tagged in the git repo as rel-YYYYMMDD.

`cqueues.COMMIT`

Git commit hash string of HEAD.

`cqueues.type(obj)`

Return the string "controller" if *obj* is a controller object, or *nil* otherwise.

`cqueues.interpose(name, function)`

Add or interpose a `cqueues` controller class method. Returns the previous method, if any.

`cqueues.monotime()`

Return the system's monotonic clock time, usually clock_gettime(CLOCK_MONOTONIC).

`cqueues.cancel(fd)`

Cancels the specified descriptor, fd, for all controllers. If fd is an object, the descriptor is obtained by calling the `:pollfd` method. Any coroutine polling on the canceled descriptor is placed on its controller's pending queue.

To simplify error and exit paths in application code, canceling a descriptor that isn't installed is a no-op. Similarly, fd may be -1. However, if fd was installed with any controller but the descriptor has already been closed, then this is an error.

Cancellation must be done before closing a descriptor[6], otherwise controller state becomes corrupted. Closing a descriptor automatically removes the descriptor from the kernel's internal polling data structures, but not the user-land data structures. When the process then attempts to modify or remove the descriptor, the operation will fail with `EBADF`. Some event loops silently suppress such errors because it's very common for applications to close a descriptor before destroying an event handle. But such ordering issues aren't always so benign. If a new socket object was created between the close and cancel operations which happens to have the same descriptor number, then the controller erroneously believes the descriptor is already installed, or if not previously installed than the cancel stalls some other thread. Such bugs can be extremely difficult to track down; they're much easier to discover if the library bubbles up `EBADF` when canceling an already closed descriptor.

`cqueues` objects—controller, notify, resolver, signal, socket, etc—automatically call cancel before closing any descriptor. Normally only extension libraries and modules need to explicitly cancel descriptors.

[6] Unless the application knows the descriptor isn't currently installed. But note that `cqueues` persists descriptor events for at least one yield/resume cycle. When `cqueues.poll` returns, for example, the descriptor is still installed. It won't be uninstalled until the coroutine yields again without requesting any events for the descriptor.

cqueues.poll(...)

Takes a series of objects obeying the polling protocol and yields control to the parent `cqueues` controller. On an event the coroutine is resumed and `.poll` returns the objects which polled ready. A number value is interpreted as a simple timeout, *not* a file descriptor.

This routine is intended to be behave much like POSIX `poll(2)`. Think of each object as a `struct pollfd` object. Then

```
local ready = { assert(cqueues.poll(socket1, socket2, 10)) }
for i=1,#ready do
    ...
end
```

looks and behaves like

```
struct pollfd pollset[2];
pollset[0].fd = fd1;
pollset[0].events = POLLIN;
pollset[1].fd = fd2;
pollset[1].events = POLLIN;
int n = poll(pollset, 2, 10 * 1000);
assert(n != -1);
for (i = 0; i < n; i++) {
    ...
}
```

cqueues.sleep(number)

Yields to the parent `cqueues` controller and schedules a wakeup for 'number' seconds in the future.

cqueues.running()

Returns two values: the immediate controller currently executing, if any, or nil; and a boolean—true if the caller's coroutine is the same coroutine resumed by the controller.

cqueues.resume(co)

See `auxlib.resume`.

cqueues.wrap(f)

See `auxlib.wrap`.

cqueues.new()

Create a new cqueues object.

cqueue:attach(coroutine)

Attach and manage the specified coroutine. Returns the controller.

`cqueue:wrap(function)`

Execute function inside a new coroutine managed by the controller. Returns the controller.

`cqueue:step([timeout])`

Step once through the event queue. Unless the timeout is explicitly specified as `0`, or unless the current thread of execution is a `cqueues` managed coroutine, *it suspends the process indefinitely or for the specified timeout* until a descriptor event or timeout fires.

Returns true on success. Otherwise returns false, an error message, and additional context: a numeric error code (possibly *nil*), a Lua thread object (possibly *nil*), an object that was polled (possibly *nil*), and an integer file descriptor (possibly *nil*). :step can be called again after errors.

If embedding `cqueues` within an existing application, the top-level :step invocation should always specify a 0 timeout. A controller is a pollable object, and the descriptor returned by the :pollfd method can be used with third-party event libraries, whether written in Lua, C, or some other language. Don't forget to also schedule a timeout using the value from :timeout.

`cqueue:loop([timeout])`

Invoke `cqueues:step` in a loop, exiting on error, timeout, or if the event queue is empty. Returns same values as `cqueues:step`.

`cqueue:errors([timeout])`

Returns an iterator function over errors returned from `cqueues:loop`. If `cqueues:loop` returns successfully because of an empty event queue, or if the timeout expires, returns nothing, which terminates any for-loop. 'timeout' is cumulative over the entire iteration, not simply passed as-is to each invocation of `cqueues:loop`.

`cqueue:empty()`

Returns true if there are no more descriptor or timeout events queued, false otherwise.

`cqueue:count()`

Returns a count of managed coroutines.

`cqueue:cancel(fd)`

Cancel the specified descriptor for that controller. See cqueues.cancel.

`cqueue:pause(signal [, signal ... ])`

A wrapper around `pselect` which *suspends execution of the process* until the controller polls ready or a signal is delivered. This interface is provided as a very basic least common denominator for simple slave process controller loops and similar scenarios, where immediate response to signal delivery is required on platforms like Solaris without a proper signal polling primitive. (`signal.listen` on Solaris merely periodically queries the pending set.)

Much better alternatives are possible for Solaris, but require global process state or an LWP thread helper.

3.2.2 `cqueues.socket`

The socket bindings provide built-in DNS, SSL/TLS, buffering, and line translation. DNS happens transparently, and SSL/TLS can be initiated with the `socket:starttls` method.

The default I/O mode is "tl"—text translation and line buffering. This makes sockets work intuitively with the most common protocols on the Internet, like SMTP and HTTP, which require CRLF and use line delimited framing.

`socket[]`

A table mapping socket related system identifier names to number codes, including AF_UNSPEC, AF_INET, AF_INET6, AF_UNIX, SOCK_STREAM, and SOCK_DGRAM.

`socket.type(obj)`

Return the string "socket" if *obj* is a socket object, or *nil* otherwise.

`socket.interpose(name, function)`

Add or interpose a socket class method. Returns the previous method, if any.

`socket.connect(host, port [, family] [, type])`

Return a new socket immediately ready for reading or writing. DNS lookup and TCP connection handling are handled transparently.

`socket.connect{ ... }`

Like `socket.connect` with list arguments, but takes a table of named arguments:

field	type:default	description
.host	string:nil	IP address or host domain name
.port	string:nil	host port
.path	string:nil	UNIX domain socket path
.family	number	protocol family—AF_INET (default), AF_INET6, AF_UNIX (default if .path specified)
.type	number	protocol type—SOCK_STREAM (default) or SOCK_DGRAM
.mode	string:nil	fchmod or chmod socket after creating UNIX domain socket
.mask	string:nil	set and restore umask when binding UNIX domain sockets
.unlink	boolean:false	unlink socket path before binding
.reuseaddr	boolean:true	SO_REUSEADDR socket option
.reuseport	boolean:false	SO_REUSEPORT socket option

.nodelay	boolean:false	TCP_NODELAY IP option
.nopush	boolean:false	TCP_NOPUSH, TCP_CORK, or equivalent IP option
.v6only	boolean:nil	enables or disables IPV6_V6ONLY IPv6 option, otherwise the system default is left as-is
.oobinline	boolean:false	SO_OOBLINE socket option
.nonblock	boolean:true	O_NONBLOCK descriptor flag
.cloexec	boolean:true	O_CLOEXEC descriptor flag
.nosigpipe	boolean:true	O_NOSIGPIPE, SO_NOSIGPIPE, MSG_NOSIGNAL, or equivalent descriptor flag
.verify	boolean:false	require SSL certificate verification
.sendname	boolean:true	send connect host as TLS SNI host name
	string:nil	send specified string as TLS SNI host name
.time	boolean:true	track elapsed time for statistics

`socket.listen(host, port)`

Return a new socket immediately ready for accepting connections.

`socket.listen{ ... }`

Like `socket.listen` with list arguments, but takes a table of named arguments. See also `socket.connect{}`.

`socket.dup(fd)`

Return a new socket using a duplicate of an existing file-descriptor (integer).

`socket.fdopen(fd)`

Like `socket.dup(fd)`, but takes ownership of the file-descriptor itself instead of duplicating it. This is useful e.g. for listening on sockets passed from systemd.

`socket.pair([type])`

Returns two bound sockets. Type should be the system type number, e.g. `socket.SOCK_STREAM` or `socket.SOCK_DGRAM`.

`socket.pair{ ... }`

Like single argument form of `socket.listen`, but takes a table of named arguments. See also `socket.connect{}`.

`socket.setvbuf(mode [, size])`

Set the default output buffering mode for all new sockets. See `socket:setvbuf`.

`socket.setmode([input] [, output])`

Set the default I/O modes for all new sockets. See `socket:setmode`.

`socket.setbufsiz([input] [, output])`

Set the default I/O buffer sizes for all new sockets. See `socket:setbufsiz`.

`socket.setmaxline([input] [, output])`

Set the default I/O line-buffering limits for all new sockets. See `socket:setmaxline`.

`socket.settimeout([timeout])`

Set the default timeout for all new sockets. See `socket:settimeout`.

`socket.setmaxerrs([which,][limit])`

Set the default error limit for all new sockets. See `socket:setmaxerrs`.

`socket.onerror([function])`

Set the default error handler for all new sockets. See `socket:onerror`.

`socket:connect([timeout])`

Wait for connection establishment to succeed. You do not need to wait before proceeding to perform read or write calls, but waiting may ease diagnosing connection problems in your code and allows you to separate connect phase from I/O phase timeouts.

`socket:listen([timeout])`

Wait for socket binding to succeed. You do not need to wait before proceeding to call `:accept`, but waiting may ease diagnosing binding problems in your code and allows you to separate listen phase from accept phase timeouts.

Socket binding may not occur immediately if you provided a host address that required DNS resolution over the network. This is uncommon for listening sockets but supported nonetheless; the symmetry simplifies internal code. Also, socket object instantiation with `socket.listen` and `socket.connect` only return errors regarding user data object construction; address lookup and binding errors are detected later, when initiated by subsequent method calls.

`socket:accept([options] [, timeout])`

Wait for and return an incoming client socket on a listening object.

Optionally takes a table of named arguments. See also `socket.connect{}`.

`socket:clients([options] [, timeout])`

Iterator over `socket:accept`: `for con in srv:clients() do ... end`.

`socket:starttls([context][, timeout])`

Place socket into TLS mode, optionally using the `openssl.ssl.context` object as the configuration prototype, and wait for the handshake to complete.[7] Returns true on success, false and an error code on failure.

`socket:checktls()`

If in TLS mode, returns an `openssl.ssl` object, otherwise nil. If the openssl module cannot be loaded, returns nil and an error string.

`socket:setvbuf(mode [, size])`

Same as Lua `file:setvbuf`. Analogous to "n", "l", and "f" mode flags. Returns the previous output mode and output buffer size.

`socket:setmode([input] [, output])`

Sets the the input and output buffering and translation modes, which mirror C's stdio semantics. Either mode can be nil or none, in which case the mode is left unchanged.

A mode is specified as a string containing one or more of the following flags

flag	default	description
t	default	text mode—input or output undergoes LF/CRLF translation
b		binary mode—no LF/CRLF translation
n		no output buffering—output buffer is always flushed completely after every write operation
l	default	line buffered output—output buffer is flushed up to and including the last LF after every write operation
f		fully buffered output—all buffer-sized blocks are flushed after every write operation (see `socket:setbufsiz` and `socket:setvbuf`)
a	default	enable autoflush—every read operation attempts to flush the output buffer as-if an explicit unbuffered flush were performed with a 0-second timeout; when initiating SSL/TLS, all pending data is fully flushed before proceeding with negotiation
A		disable autoflush
p	default	enable pushback—when initiating SSL/TLS, any data in the input buffer is virtually pushed back to the socket so that it will be processed as part of the SSL/TLS handshake
P		disable pushback

Returns the previous input and output modes as fixed-sized strings. At present the first character is one of "t" or "b", and the second character one of "n", "l", "f", or "-" (for in the input mode).

[7] Prior to 2014-04-30, if no timeout was specified then the routine returned immediately.

`socket:setbufsiz([input] [, output])`

Sets the input and output buffer size. Either size can be nil or none, in which case the size is left unchanged.

These are not hard limits for SOCK_STREAM sockets. The input buffer argument simply sets a minimum for input buffering, to reduce syscalls. The output buffer argument is the same as provided to `:setvbuf`, and effectively changes when flushing occurs for full- or line-buffered output modes.

For SOCK_DGRAM sockets, the input buffer sets a hard limit on the size of datagram messages. Any message over this size will be truncated, unless a previous block- or line-buffered read operation forced the buffer to be reallocated to a larger size.

Returns the previous input and output buffer sizes, or throws an error if the buffers could not be reallocated.

`socket:setmaxline([input] [, output])`

Sets the maximum input and output length for line-buffered operations. Either size can be nil or none, in which case the size is left unchanged.

These are hard limits. For line-buffered input operations, if a \n character is not found within this limit then the data is processed as-if EOF was reached at this boundary. For line-buffered output, a chunk is always flushed at this boundary.

Returns the previous input and output sizes.

`socket:settimeout([timeout])`

Sets the default timeout period for I/O. If nil or none, then clears any default timeout. If a timeout is cleared, any operation which polls will wait indefinitely until completion or an error occurs.

Sockets are instantiated without a default timeout.

`socket:setmaxerrs([which,] limit)`

Set the maximum number of times an error will be returned to a caller before throwing an error, instead. *which* specifies which I/O channel limit to set—"r" for the input channel, "w" for the output channel, or "rw" for both. *which* defaults to "rw". *limit* is an integer limit. The initial *limit* is 100.

Returns the previous channel limits in the order specified.

Note that `socket:clearerr` will clear the error counters as well as any errors.

Unchecked error loops The default error handler will throw on most errors. However, EPIPE and ETIMEDOUT are returned directly as they're common errors that normally need to be handled explicitly in correct applications. Furthermore, errors will be repeated until cleared. If errors were not repeated then unchecked transient errors could lead to difficult to detect loss of data bugs by giving the illusion of successful forward progress.[8] Code which loops and fails to check the success

[8] This is especially true of Lua's for-loop iterator pattern.

of I/O calls could enter an infinite loop which never yields to the controller and stalls the process. This is a fail-safe mechanism to catch such code.

`socket:onerror([function])`

Set the error handler. The error handler is passed four arguments: socket object, method name, error number, and stack level of caller. The handler is expected to either throw an error or return an error code—to be returned to the caller as part of the documented return interface.

The default error handler returns `EPIPE` and `ETIMEDOUT` directly, and throws everything else. `EAGAIN` is handled internally for logically synchronous calls.

Returns the previous error handler, if any.

`socket:error([which])`

Returns the saved error conditions for the input and output channels. *which* is a string containing one or more of the characters 'r' and 'w', which return the input and output channel errors respectively and in the order specified. *which* defaults to the string "rw".

`socket:clearerr([which])`

Clears the error conditions and counters for the specified I/O channels and returns any previous errors. *which* is a string containing one or more of the characters "r" and "w", which clears the input and output channel errors respectively, and returns the previous error numbers (or nil) in the order specified. *which* defaults to the string "rw".

`socket:read(...)`

Similar to Lua's `file:read`, with additional formats.

format	description
*n	unsupported
*a	read until EOF
*l	read the next line, trimming the EOL marker
*L	read the next line, keeping the EOL marker
*h	read and unfold MIME compliant header
*H	read MIME compliant header, keeping EOL markers
`--`*marker*	read multipart MIME entity chunk delineated by MIME boundary *marker*
number	read *number* bytes or until EOF
−*number*	read 1 to *number* bytes, immediately returning if possible

For SOCK_DGRAM sockets, each message is treated as-if EOF was reached. The slurp operation returns a single datagram, and line-buffered operations will return the remaining text in a message even without a terminating \n. Datagrams will be truncated if the message is larger than the input buffer size.

The MIME entity reader allows efficient reading of large MIME-encoded bodies, such as with HTTP POST file uploads. The format will return chunks until the boundary is reached. The last chunk will have any trailing EOL marker removed, regardless of input mode, as this is part of the boundary token. In binary mode chunks are sized according to the current input channel buffer size, except that the last chunk will probably be short. In text mode chunks will not exceed the maximum of the current input channel buffer size or maximum line size; and in addition to EOL translation, chunks are broken along line boundaries with multiple lines aggregated into a single chunk.

Both the MIME header and MIME entity reader require a proper terminating condition. In particular, *EOF is not a terminating condition*. Applications must be careful to handle truncation if the stream was prematurely closed. When looping over one of these input formats, the application should read the next line of input after the loop terminates. If the next next line does not match the terminating condition, then the stream is invalid and the application should abort processing the stream.

For MIME headers the next line should be non-*nil* and should not appear to be a prefix of a header.

```
local function isbreak(ln) -- requires *L, not *l
    return find(ln, "\n", #ln, true) and not match(ln, "[%w%-_]+%s*:")
end
```

For MIME entities the next line should begin with the boundary text.

```
local function isboundary(marker, ln)
    local p, pe = find(ln, marker, 1, true)

    if p == 1 then
        if find(ln, "^\r?\n?$", pe + 1) then
            return "begin"
        elseif find(ln, "^--\r?\n?$", pe + 1) then
            return "end"
        end
    end

    return false
end
```

`socket:xread(format[, mode][, timeout])`

Like `socket:read`, but only takes a single format instead of a list of formats, and permits specifying an input mode and timeout. *mode* should be in the format described at `socket:setmode`. *mode* and *timeout* are used only for the current read operation; they do not change the default mode and timeout for the socket.

`socket:lines(...)`

Returns an iterator function over `socket:read`.

socket:xlines([format][, mode][, timeout])

Returns an iterator function over `socket:xread`.

socket:write(...)

Same as Lua `file:write`.

socket:xwrite(string[, mode][, timeout])

Like `socket:write`, but only takes a single string, and permits specifying an output mode and timeout. *mode* should be in the format described at `socket:setmode`. *mode* and *timeout* are used only for the current write operation; they do not change the default mode and timeout for the socket.

socket:flush([mode][, timeout])

Flushes the output buffer. Mode is one of the "nlf" flags described in `socket:setmode`. A nil mode implies "n", i.e. no buffering and effecting a full flush. An empty string mode resolves to the configured output buffering mode.

socket:fill(size[, timeout])

Fills the input buffer with 'size' bytes. Returns true on success, false and an error code on failure.

socket:unget(string)

Writes 'string' to the head of the socket input buffer.

socket:pending()

Returns two numbers—the counts of buffered bytes in the input and output streams. This does not include the bytes in the kernel's buffer.

socket:uncork()

Disables TCP_NOPUSH, TCP_CORK, or equivalent socket option.

socket:recv(format [, mode])

Similar to `socket:read`, except takes only a single format and returns immediately without polling. On success returns the string or number. On failure returns nil and a numeric error code–usually EAGAIN or EPIPE. Does not use error handler.

'mode' is as described in `socket.connect`, and defaults to the configured input mode.

socket:send(string, i, j [, mode])

Write out the slice 'string'[i,j]. Similar to passing `string:sub(i, j)`, but without instantiating a new string object. Immediately returns two values: count of bytes written (0 to j-i+1), and numerical error code, if any (usually EAGAIN or EPIPE).

socket:recvfd([prepbufsiz][, timeout])

Receive an ancillary socket message with accompanying descriptor. 'prepbufsiz' specifies the maximum message size to expect.

This routine bypasses I/O buffering.

Returns message-string, socket-object on success; nil, nil, error-integer on failure. On success socket-object may still be nil. Message truncation is treated as an error condition.

socket:sendfd(msg, socket[, timeout])

Send an ancillary socket message with accompanying descriptor. 'msg' should be a non-zero-length string, which some platforms require. 'socket' should be a Lua file handle, `cqueues` socket, integer descriptor, or nil.

This routine bypasses I/O buffering.

Returns true on success; false and an error code on failure.

socket:shutdown(how)

Simple binding to `shutdown(2)`. 'how' is a string containing one or both of the flags "r" or "w".

flag	description
r	analagous to `shutdown(SHUT_RD)`
w	analagous to `shutdown(SHUT_WR)`

socket:eof([which])

Returns boolean values representing whether EOF has been received on the input channel, and whether the output channel has signaled closure (e.g. `EPIPE`). *which* is a string containing one or more of the characters "r" and "w", which return the state of the input or output channel, respectively, in the order specified. *which* defaults to "rw".

Note that `socket:shutdown` does not change the state of these values. They are set only upon receiving the condition after I/O is attempted.

socket:peername()

Returns one, two, or three values. On success, returns three values for AF_INET and AF_INET6 sockets—the address family number, IP address string, and IP port. For AF_UNIX sockets, returns the address family and either the file path or nil (e.g. for an unnamed socket). If the socket is not yet connected, returns the address family AF_UNSPEC, usually numeric 0.

On failure returns nil and a numeric error code.

`socket:peereid()`

Queries the effective UID and effective GID of an AF_UNIX, SOCK_STREAM peer as cached by the kernel when the stream initially connected.

Returns two numbers representing the UID and GID, respectively, on success, otherwise nil and a numeric error code.

`socket:peerpid()`

Queries the PID of a AF_UNIX, SOCK_STREAM peer as cached by the kernel when the stream initially connected. This capability is unsupported on OS X and FreeBSD; they only provide `getpeereid`, which cannot provide the PID.

Returns a number representing the PID on success, otherwise nil and a numeric error code.

`socket:localname()`

Identical to `socket:peername`, but returns the local address of the socket.

`socket:stat()`

Returns a table containing two subtables, 'sent' and 'rcvd', which each have three fields—.count for the number of bytes sent or received, a boolean .eof signaling whether input or output has been shutdown, and .time logging the last send or receive operation.

`socket:close()`

Explicitly and immediately close all internal descriptors. This routine ensures all descriptors are properly cancelled.

3.2.3 `cqueues.errno`

`errno[]`

A table mapping all system error string macros to numerical error codes, and all numerical error codes to system error string macros. Thus, `errno.EAGAIN` evaluates to a numeric error code, and `errno[errno.EAGAIN]` evaluates to the string "EAGAIN".

`errno.strerror(code)`

Returns string returned by strerror(3).

3.2.4 `cqueues.signal`

`signal[]`

A table mapping signal string macros to numerical signal codes. In all likelihood, `signal.SIGKILL` evaluates to the number 9.

`signal.strsignal(code)`

Returns string returned by strsignal(3).

`signal.ignore(signal [, signal ... ])`

Set the signal handler to SIG_IGN for the specified signals.

`signal.default(signal [, signal ... ])`

Set the signal handler to SIG_DFL for the specified signals.

`signal.discard(signal [, signal ... ])`

Set the signal handler to a builtin "noop" handler for the specified signals. Use this is you want signals to interrupt syscalls.

`signal.block(signal [, signal ... ])`

Block the specified signals.

`signal.unblock(signal [, signal ... ])`

Unblock the specified signals.

`signal.raise(signal [, signal ... ])`

raise(3) the specified signals.

`signal.type(obj)`

Return the string "signal listener" if *obj* is a signal listener object, or *nil* otherwise.

`signal.interpose(name, function)`

Add or interpose a signal listener class method. Returns the previous method, if any.

`signal.listen(signal [, signal ... ])`

Returns a signal listener object for the specified signals. Semantics differ between platforms:

kqueue BSD `kqueue` provides the most intuitive behavior. All listeners will detect a signal sent to the process irrespective of whether the signal is ignored, blocked, or delivered. However, EVFILT_SIGNAL is edge-triggered, which means no notification of delivery of a pending signal upon being unblocked.

signalfd Linux `signalfd` will not detect ignored or delivered signals, and only one signalfd object will poll ready per signal.

sigtimedwait Solaris provides no signal polling kernel primitive. Instead, the pending set is periodically queried using `sigtimedwait`. See `signal:settimeout`. Like Linux, only one listener can notify per interrupt.

To be portable the application must block the relevant signals. See `signal.block`. Otherwise, neither Linux nor Solaris will be able to detect the interrupt. Any signal should be assigned to one listener only, although any listener may query multiple signals.

Alternatively, applications may start a dedicated thread to field incoming signals, and send notifications over a socket. In the future this may be provided as an optional listener implementation.

See also `cqueue:pause` for another, if crude, alternative.

`signal:wait([timeout])`

Polls for the signal set passed to the constructor. Returns the signal number, or nil on timeout.

`signal:settimeout(timeout)`

Set the polling interval for implementations such as Solaris which lack a signal polling kernel primitive. On such systems signal:wait merely queries the pending set every 'timeout' seconds.

3.2.5 `cqueues.thread`

`thread.type(obj)`

Return the string "thread" if *obj* is a thread object, or *nil* otherwise.

`thread.self()`

Returns the LWP thread object for the running Lua instances. Threads not started via thread.start return nil.

`thread.start(function [, string [, string ... ]])`

Generates a socket pair, starts a POSIX LWP thread, initializes a new Lua VM instance, preloads the `cqueues` library, and loads and executes the specified function from the new LWP thread and Lua instance. The function receives as the first parameter one end of the socket pair—instantiated as a `cqueues.socket` object—followed by the string parameters passed to thread.start.

The new LWP thread starts with all signals blocked.

Returns a thread object and a socket object—the other end of the socket pair. The thread object is pollable, and readiness signals that the LWP thread has exited, or is imminently about to exit.

On error returns two nils and an error code.

`thread.join([timeout])`

Wait for the thread to terminate. Calling the equivalent of thread.self():join() is disallowed.

Returns a boolean and error value. If false, error value is an error code describing a local error, usually `EAGAIN` or `ETIMEDOUT`. If true, error value is 1) an error code describing a system error which the thread encountered, 2) an error message string returned by the new Lua instance, or 3) nil if completed successfully.

3.2.6 `cqueues.notify`

`notify[]`

A table mapping bitwise flags to names, and vice-versa.

name	description
CREATE	file creation event
ATTRIB	metadata change event
MODIFY	modification to file contents or directory entries
REVOKE	permission revoked
DELETE	file deletion event
ALL	bitwise-or of CREATE, DELETE, ATTRIB, MODIFY, and REVOKE

`notify.flags(bitset[, bitset ... ])`

Returns an iterator over the flags in the specified bitwise change sets. Thus, `notify.flags(bit32.xor(notify.CRE`
`notify.DELETE), notify.MODIFY)` returns an iterator returning all three flags.

`notify.type(obj)`

Return the string "file notifier" if *obj* is a notification object, or *nil* otherwise.

`notify.opendir(path[, changes ])`

Returns a notification object associated with the specified directory. Directory change events are limited to the set, 'changes', or to notify.ALL if nil.

`notify:add(name[, changes ])`

Track the specified file name within the notification directory. 'changes' defaults to notify.ALL if nil.

`notify:get([timeout])`

Returns a bitwise change set and a filename on success.

`notify:changes([timeout])`

Returns an iterator over the `notify:get` method.

3.2.7 `cqueues.dns`

As the internal DNS implementation has no global state, `cqueues.dns` is mostly a convenience wrapper around other facilities.

`dns.version()`

Returns the release, ABI, and API version numbers of the internal DNS implementation as three numbers.

`dns.query(name[, type][, class][, timeout])`

Proxies the `resolvers:query` method of the internal resolver pool. If no resolver pool has been set with `dns:setpool`, creates a new stub resolver pool.

`dns.setpool(pool)`

Sets the internal resolver pool for use by subsequent calls to `dns.query` to *pool*.

`dns.getpool()`

Returns the internal resolver pool. This routine should never return nil, as it will automatically create a new resolver pool if none has been set yet.

3.2.8 `cqueues.dns.record`

DNS resource record objects are implemented within `cqueues.dns.record`. The global tables and shared methods are documented below. The type-specific accessory methods are quite numerous. Until documented please confer with cqueues/src/dns.c. Also, the accessory method names are usually equivalent to the structure member names in cqueues/src/lib/dns.h, which in return usually reflect the member names in the relevant RFC.

The `__tostring` metamethod returns a representation of the record data only, excluding the name, type, ttl, etc. For an A record, it's equivalent to string.format("%s", `rr:addr()`). For MX—which has multiple members—it's string.format("%d %s", rr:preference(), rr:host()).

`record.type[]`

A table mapping DNS record type string identifiers to number values, and vice-versa. So, `record.type.A` evaluates to 1, the IANA numeric record type. String identifiers are only provided for record types which are directly parseable and composable by the library. Currently supported types include A, NS, CNAME, SOA, PTR, MX, TXT, AAAA, SRV, OPT, SSHFP, and SPF. Other record types can be instantiated, but the numeric type must be used and the only methods available operate on the raw rdata.

`record.class[]`

A table mapping DNS record class string identifiers to number values, and vice-versa. At present the only class included is IN.

`record.sshfp[]`

A table mapping DNS SSHFP record string identifiers to the number values—RSA, DSA, and SHA1.

`record.type(obj)`

Return the string "dns record" if *obj* is a record object, or *nil* otherwise.

`record:section()`

Returns the section identifier from whence the record came, if derived from a packet. Specifically, QUESTION, ANSWER, AUTHORITY, or ADDITIONAL. See `cqueues.dns.packet.section[]`.

`record:name()`

Returns the uncompressed record domain name as a string.

`record:type()`

Returns the numeric record type. If 'rr' holds an AAAA record, then the return value of rr:type() will compare equal to `record.type.AAAA`.

`record:class()`

Returns the numeric record class. See `record.class[]`.

`record:ttl()`

Returns the record TTL.

3.2.9 `cqueues.dns.packet`

DNS packets are stored in a simple structure encapsulating the raw packet data. One consequence is that packets are append only. Because a packet is composed of four adjacent sections, when building a packet all the information necessary should be at-hand so that records can be appended in order.

The `__tostring` metamethod composes a string similar to the output of the venerable dig utility.

`packet.section[]`

A table mapping packet section string identifiers to number values, and vice-versa. A packet is composed of only four sections: QUESTION, ANSWER, AUTHORITY, and ADDITIONAL.

`packet.opcode[]`

A table mapping packet opcode string identifiers to number values, and vice-versa. The currently mapped opcodes are QUERY, IQUERY, STATUS, NOTIFY, and UPDATE.

`packet.rcode[]`

A table mapping packet rcode string identifiers to number values, and vice-versa. The currently mapped rcodes are NOERROR, FORMERR, SERVFAIL, NXDOMAIN, NOTIMP, REFUSED, YXDOMAIN, YXRRSET, NXRRSET, NOTAUTH, and NOTZONE.

`packet.type(obj)`

Return the string "dns packet" if *obj* is a packet object, or *nil* otherwise.

`packet.interpose`

Add or interpose a packet class method. Returns the previous method, if any.

`packet.new([prepbufsiz])`

Instantiate a new packet object. 'prepbufsiz' is the maximum space available for appending compressed records. For constructing a packet with a single question, the most space possibly necessary is 260—256 bytes for the name, and 2 bytes each for the type and class (a QUESTION record has no TTL or rdata section).

`packet:qid()`

Returns the 16-bit QID value.

`packet:flags()`

Returns a table of packet header flags.

field	type	description
.qr	integer	specifies whether the packet is a query (0) or response (1)
.opcode	number	specifies the query type
.aa	boolean	signals an authoritative answer
.tc	boolean	signals packet truncation
.rd	boolean	signals "recursion desired"
.ra	boolean	signals "recursion available"
.z	boolean	reserved by RFC 1035 and used by other RFCs
.rcode	integer	specifies the response disposition

`packet:count([sections])`

Returns a count of records in the sections specified by the bitwise parameter 'sections'. Defaults to `packet.section.ALL`, which is the XOR of all four sections.

`packet:grep{ ... }`

Returns a record iterator over the packet according to all the criteria specified by the optional table parameter.

field	description
.section	select records by bitwise AND with the specified sections
.type	select records of this type (not bitwise)
.class	selects records of this class (not bitwise)
.name	select records with this name

3.2.10 `cqueues.dns.config`

The traditional BSD /etc/resolv.conf file is the prototype for this module, although it's also capable of parsing /etc/nsswitch.conf. `cqueues.dns.config` objects are used when instantiating new resolver objects, and provide the general options controlling a resolver.

The `__tostring` metamethod composes a string adhering to /etc/resolv.conf syntax, with /etc/nsswitch.conf alternatives as comments.

`config[]`

A table mapping flag identifiers to number values.

field	description
TCP_ENABLE	fall back to TCP when truncation detected (default)
TCP_ONLY	only use TCP when querying
TCP_DISABLE	do not fall back to TCP
RESOLV_CONF	specifies BSD /etc/resolv.conf input syntax
NSSWITCH_CONF	specifies Solaris /etc/nsswitch.conf input syntax

`config.type(obj)`

Return the string "dns config" if *obj* is a config object, or *nil* otherwise.

`config.interpose(name, function)`

Add or interpose a config class method. Returns the previous method, if any.

`config.new{ ... }`

Returns a new config object, optionally initialized according to the specified table values.

field	type	description
.nameserver	table	list of IP address strings to use for stub resolvers
.search	table	list of domain suffixes to append to query names
.lookup	table	order of lookup methods—"file" and "bind"
.options	table	canonical location for .edns0, .ndots, .timeout, .attempts, .rotate, .recurse, .smart, and .tcp options
..edns0	boolean	enable EDNS0 support
..ndots	number	if query name has fewer labels than this, reverse suffix search order

..timeout	number	timeout between query retries
..attempts	number	maximum number of attempts per nameserver
..rotate	boolean	randomize nameserver selection
..recurse	boolean	query recursively instead of as a simple stub resolver
..smart	boolean	for NS, MX, SRV and similar record queries, resolve the A record if not included as glue in the initial answer
..tcp	number	see TCP_ENABLE, TCP_ONLY, TCP_DISABLE in `config[]`
.interface	string	IP address to bind to when querying (e.g. [192.168.1.1]:1234)

`config.stub{ ... }`

Returns a config object initialized for a stub resolver by loading the relevant system files; e.g. /etc/resolv.conf and /etc/nsswitch.conf. Takes optional initialization values like `config.new`.

`config.root{ ... }`

Returns a config object initialized for a recursive resolver. Takes optional initialization values like `config.new`.

`config:loadfile(file[, syntax])`

Parse the Lua file object 'file'. 'syntax' describes the format, which should be RESOLV_CONF (default), or NSSWITCH_CONF.

`config:loadpath(path[, syntax])`

Like :loadfile, but takes a file path.

`config:get()`

Returns the configuration as a Lua table structure. See `config.new` for a description of the values.

`config:set{ ... }`

Apply the defined configuration values. The table should have the same structure as described for `config.new`.

3.2.11 `cqueues.dns.hosts`

The traditional BSD /etc/hosts file is the prototype for this module, and provides resolvers the data source for the "file" lookup method.

The `__tostring` metamethod composes a string adhering to /etc/hosts syntax.

hosts.type(obj)

Return the string "dns hosts" if *obj* is a hosts object, or *nil* otherwise.

hosts.interpose(name, function)

Add or interpose a hosts class method. Returns the previous method, if any.

hosts.new()

Returns a new hosts object.

hosts.stub()

Returns a host object initialized for a stub resolver by loading the relevant system files; e.g. /etc/hosts.

hosts.root()

Returns a hosts object initialized for a recursive resolver.

hosts:loadfile(file)

Parse the Lua file object 'file' for host entries.

hosts:loadpath(path)

Like :loadfile, but takes a file path.

hosts:insert(address, name[, alias])

Inserts a new hosts entry. 'address' should be an IPv4 or IPv6 address string, 'name' the domain name, and 'alias' a boolean—true if 'name' is canonical and a valid response for a reverse address lookup.

3.2.12 cqueues.dns.hints

The internal DNS library is implemented as a recursive resolver. No matter whether configured as a stub or recursive resolver, when a query is submitted it consults a "hints" database for the initial name servers to contact. In stub mode these would usually be the local recursive, caching name servers, derived from the `cqueues.dns.config` object; in recursive mode, the root IANA name servers.

The `__tostring` metamethod composes a multi-line string indexing SOA zone names and addresses.

hints.type(obj)

Return the string "dns hints" if *obj* is a hints object, or *nil* otherwise.

`hints.interpose(name, function)`

Add or interpose a hints class method. Returns the previous method, if any.

`hints.new([resconf])`

Returns a new hints object. 'resconf' is an optional `cqueues.dns.config` object which in the future may be used to initialize database behavior. Currently it's unused, and *does not* pre-load the name server list.

`hints.stub([resconf])`

Returns a hints object initialized for a stub resolver. If provided, the initial hints are taken from the `cqueues.dns.config` object, 'resconf'. Otherwise, the hints are derived from a temporary "stub" config object internally.

`hints.root([resconf])`

Returns a hints object initialized for a recursive resolver. The root name servers are initialized from an internal database compiled into the module. See `hints.new` for the function of the optional 'resconf'.

`hints:insert(zone, address|resconf[, priority])`

Inserts a new hints entry. 'zone' is the domain name which anchors the SOA (e.g. ".", or "com."), and 'address' the IPv4 or IPv6 of the nameserver. Alternatively, in lieu of a string address a `cqueues.dns.config` object can be specified, and the addresses taken from the nameserver list property. 'priority' is used for ordering nameservers in each zone.

IPv4 and IPv6 addresses can optionally contain a port component, e.g. "[2001:503:ba3e::2:30]:123" or "[198.41.0.4]:53".

3.2.13 `cqueues.dns.resolver`

This module implements a comprehensive DNS resolution algorithm, capable of working in both stub and recursive modes, and automatically querying for missing glue records.

The resolver implementation only supports one outstanding query per resolver, with a 1:1 mapping between resolvers and sockets. This is intended to promote both simplicity and security—it maximizes port number and QID entropy to mitigate spoofing. An additional module, `cqueues.dns.resolvers`, implements a resolver pool to assist with bulk querying.

`resolver.type(obj)`

Return the string "dns resolver" if *obj* is a resolver object, or *nil* otherwise.

`resolver.interpose(name, function)`

Add or interpose a resolver class method. Returns the previous method, if any.

`resolver.new([resconf][,hosts][,hints])`

Returns a new resolver object, configured according to the specified config, hosts, and hints objects. 'resconf' can be either an object, or a table suitable for passing to `config.new`. 'hosts' and 'hints', if nil, are instantiated according to the mode—recursive or stub—of the config object.

`resolver.stub{ ... }`

Returns a stub resolver, optionally initialized to the defined config parameters, which should have a structure suitable for passing to `cqueues.dns.config.new`.

`resolver.root{ ... }`

Returns a recursive resolver, optionally initialized to the defined config parameters, which should have a structure suitable for passing to `cqueues.dns.config.new`.

`resolver:query(name[, type][, class][, timeout])`

Query for the DNS resource record with the specified type and class. *name* is the fully-qualified or prefix domain name string. *type* and *class* corresponding to the IANA-assigned numeric or string identifier for the type of answer desired, and default to A (0x01) and IN (0x01), respectively. *timeout* is the total elapsed time for resolution, irrespective of the *.attempts* and *.timeout* configuration values.[9]

Returns a `cqueues.dns.packet` answer packet on success, or nil and a numeric error code on failure. The answer may not actually have anything in the ANSWERS section; e.g. if the RCODE is NXDOMAIN.

This routine is a simple wrapper around `resolver:submit` and `resolver:fetch`.

`resolver:submit(name[, type][, class])`

Resets the query state and submits a new query. Returns true on success, or false and an error number on failure. This routine does not poll.

`resolver:fetch()`

Process a previously submitted query. Returns a `dns.packet` object on success, or nil and an error number on failure—usually `EAGAIN`. This routine does not poll.

`resolver:stat()`

Returns a table of statistics for the resolver instance.

field	description
.queries	number of queries submitted

[9]The `resolv.conf` *.timeout* controls the time to wait on each query to a nameserver, while *.attempts* controls how many times to query each nameserver in the nameserver list. Thus in the absence of an overall timeout, the effective timeout is *.timeout* x *.attempts* x number of nameservers.

.udp.sent.count	number of UDP packets sent
.udp.sent.bytes	number of UDP bytes sent
.udp.rcvd.count	number of UDP packets received
.udp.rcvd.bytes	number of UDP bytes received
.tcp.sent.count	number of TCP packets sent
.tcp.sent.bytes	number of TCP bytes sent
.tcp.rcvd.count	number of TCP packets received
.tcp.rcvd.bytes	number of TCP bytes received

`resolver:close()`

Explicitly destroy the resolver object, immediately closing all internal descriptors. This routine ensures all descriptors are properly cancelled.

3.2.14 `cqueues.dns.resolvers`

A resolver pool is both a factory and container for resolver objects. When a resolver is requested it attempts to pull one from the internal queue. If none is available and the *.hiwat* mark has not been reached, a new resolver is created, otherwise the calling coroutine waits on a conditional variable until a resolver becomes available, or the request times-out. When a resolver is placed back into the queue it is cached if the number of cached resolvers is below *.lowat*, otherwise it is closed and discarded.

`resolvers.type(obj)`

Return the string "dns resolver pool" if *obj* is a resolver pool object, or *nil* otherwise.

`resolvers.new([resconf][,hosts][,hints])`

Behaves similar to `resolver:new`. Returns a new resolver pool object.

`resolvers.stub{ ... }`

Returns a stub resolver pool, with each resolver optionally initialized to the defined config parameters, which should have a structure suitable for passing to `cqueues.dns.config.new`.

`resolvers.root{ ... }`

Returns a recursive resolver pool, with each resolver optionally initialized to the defined config parameters, which should have a structure suitable for passing to `cqueues.dns.config.new`.

`resolvers:query(name[, type][, class][, timeout])`

Behaves similar to `resolver:query`, except that *timeout* is inclusive of the time spent waiting for a resolver to become available in the pool.

`resolvers:get([timeout])`

Return a resolver from the pool. If *timeout* is expires, returns nil and ETIMEDOUT.

`resolvers:put(resolver)`

Returns *resolver* back to the pool. Any waiting coroutines are woken.

3.2.15 `cqueues.condition`

This module implements a condition variable. A condition variable can be used to queue multiple Lua threads to await a user-defined event. Unlike some condition variable implementations, this one does not implement the monitor pattern directly. A monitor uses both a mutex and a condition variable. However, a full monitor will usually be unnecessary as coroutines do not run in parallel. Monitors are more a necessity in pre-emptive threading environments.

The condition variable primitive can be used to implement mutexes, semaphores, and monitors.

`condition.type(obj)`

Returns the string "condition" if *obj* is a condition variable, or *nil* otherwise.

`condition.interpose(name, function)`

Add or interpose a condition class method. Returns the previous method, if any.

`condition.new([lifo])`

Returns a new condition variable object. If 'lifo' is `true`, waiting threads are woken in LIFO order, otherwise in FIFO order.

Note that the `cqueues` scheduler might schedule execution of multiple woken threads in a different order. The LIFO/FIFO behavior is most useful when implementing a mutex and for whatever reason you wish to select the thread which has waited either the longest or shortest amount of time.

`condition:wait([...])`

Wait on the condition variable. Additional arguments are yielded to the `cqueues` controller for polling. Passing an integer, for example, allows you to effect a timeout. Passing a socket allows you to wait on both the condition variable and the socket.

Returns true if the thread was woken by the condition variable, and false otherwise. Additional values are returned if they polled as ready. It's possible that both the condition variable and, e.g., a socket object poll ready simultaneously, in which case two values are returned—true and the socket object.

You can also directly yield a condition variable, along with other condition variables, timeouts, or pollable objects, to the `cqueues` controller with `cqueues.poll`.

`condition:signal([n])`

Signal a condition, wakening one or more waiting threads. If specified, a maximum of 'n' threads are woken, otherwise all threads are woken.

3.2.16 `cqueues.promise`

This module implements the promise/future pattern. It most closely resembles the C++11 std::promise and std::future APIs rather than the JavaScript Promise API. JavaScript lacks coroutines, so JavaScript Promises are overloaded with complex functionality intended to mitigate the problems with lacking such a primitive. The typical usage of promises/futures with C++11's threading model mirrors how they would be typically used in `cqueues`' thread–like model.

The promise object uses a condition variable to wakeup any coroutines waiting inside `promise:wait` or `promise:get`.

`promise.type(obj)`

Returns the string "promise" if *obj* is a promise, or *nil* otherwise.

`promise.new([`*f*`[, ...]])`

Returns a new promise object. *f* is an optional function to run asynchronously, to which any subsequent arguments are passed. *f* is called using `pcall`, and the return values of `pcall` are passed directly to `promise:set`.

`promise:status()`

Returns "pending" if the promise is yet unresolved, "fulfilled" if the promise has been resolved (`promise:get` will return the values), or "rejected" if the promise failed (`promise:get` will throw an error).

`promise:set(`*ok*`[, ...])`

Resolves the state of the promise object. If *ok* is `true` then any subsequent arguments will be returned to `promise:get` callers. If *ok* is `false` then an error will be thrown to `promise:get` callers, with the error value taken from the first subsequent argument, if any.

`promise:set` can only be called once. Subsequent invocations will throw an error.

`promise:get([`*timeout*`])`

Wait for resolution of the promise object (if unresolved) and either return the resolved values directly or, if the promise was "rejected", throw an error. If *timeout* is specified, returns nothing if the promise is not resolved within the timeout.

`promise:wait([`*timeout*`])`

Wait for resolution of the promise object or until *timeout* expires. Returns promise object if the status is no longer pending (i.e. "fulfilled" or "rejected"), otherwise `nil`.

`promise:pollfd()`

Returns a condition variable suitable for polling which is used to signal resolution of the promise to any waiting threads.[10]

3.2.17 `cqueues.auxlib`

The auxiliary module exposes some convenience interfaces, including some interfaces to help with application integration or for dealing with quirky behavior that hasn't yet been changed because of API stability concerns.

`auxlib.assert(`*v* `[...])`

Similar to Lua's built-in `assert`, except that when *v* is false searches the argument list for the first non-nil, non-false value to use as the message. If the message is an integer, applies `errno.strerror` to derive a human readable string.

This routine can be explicitly monkey patched to be the global `assert`.

Most `cqueues` interfaces return a single integer error rather than the Lua-idiomatic string followed by an integer error. The original concern was that most "errors" would be EAGAIN, ETIMEDOUT, or EPIPE, which occur very often and would be costly to continually copy onto the stack as strings, especially given that they'd normally be discarded. In the future the plan is to revert to the idiomatic return protocol used by Lua's `file` API, but memoize the more common errno string representations using upvalues so they can be efficiently returned.

`auxlib.fileresult(`*v* `[...])`

Serves a similar purpose as `auxlib.assert`, except on error returns *v* (`nil` or `false`) followed by the string message and any integer error. For example, in

```
1    local v, why, syserr = fileresult(false, nil, EPERM)
```

v is `false`, *why* is "Operation not permitted", and *syserr* is EPERM. Whereas with

```
1    local v, why, syserr = fileresult(nil, ``No such file or directory'')
```

v is `nil`, *why* is "No such file or directory", and *syserr* is `nil`.

`auxlib.resume(`*co* `[...])`

Similar to Lua's built-in `coroutine.resume`, except that when coroutines yield using `cqueues.poll` recursively yields up the stack until the controller is reached, and then silently restart the coroutine when the poll operation completes. This permits creating iterators which can transparently yield. The application must be careful to ensure that this wrapper is used at every point in a yield/resume chain to get the automatic behavior.

This routine can be explicitly monkey patched to be `coroutine.resume`.

[10] To improve performance of the scheduler the pollfd member is itself the condition variable, but it can be called as a function because condition variables support the ___call metamethod.

`auxlib.tostring(`v`)`

Similar to Lua's built-in `tostring`, except supports yielding of ___tostring metamethods.

This routine can be explicitly monkey patched to be the global `tostring`.

`auxlib.wrap(`f`)`

Similar to Lua's built-in `coroutine.wrap`, except uses `auxlib.resume` when resuming coroutines.

This routine can be explicitly monkey patched to be `coroutine.wrap`.

Note that unlike `cqueues:wrap`, the created coroutine is not attached to a controller.

4 Examples

4.1 HTTP SSL Request

```
local cqueues = require"cqueues"
local socket = require"cqueues.socket"

local http = socket.connect("google.com", 443)

local cq = cqueues.new()

cq:wrap(function()
    http:starttls()

    http:write("GET / HTTP/1.0\n")
    http:write("Host: google.com:443\n\n")

    local status = http:read()
    print("!", status)

    for ln in http:lines"*h" do
        print("|", ln)
    end

    local empty = http:read"*L"
    print"~"

    for ln in http:lines"*L" do
        io.stdout:write(ln)
    end

    http:close()
end)

assert(cq:loop())
```

4.2 Multiplexing Echo Server

```
local cqueues = require"cqueues"
local socket = require"cqueues.socket"
local bind, port, wait = ...

local srv = socket.listen(bind or "127.0.0.1", tonumber(port or 8000))

local cq = cqueues.new()

cq:wrap(function()
    for con in srv:clients(wait) do
        cq:wrap(function()
            for ln in con:lines("*L") do
                con:write(ln)
            end

            con:shutdown("w")
        end)
    end
end)

assert(cq:loop())
```

4.3 Thread Messaging

```
local cqueues = require"cqueues"
local thread = require"cqueues.thread"

-- we start a thread and pass two parameters--`0' and '9'
local thr, con = thread.start(function(con, i, j)
    -- the `cqueues' upvalue defined above is gone
    local cqueues = require"cqueues"
    local cq = cqueues.new()

    cq:wrap(function()
        for n = tonumber(i), tonumber(j) do
            io.stdout:write("sent ", n, "\n")
            con:write(n, "\n")
             -- sleep so our stdout writes don't mix
            cqueues.sleep(0.1)
        end
    end)

    assert(cq:loop())
end, 0, 9)


local cq = cqueues.new()

cq:wrap(function()
    for ln in con:lines() do
        io.stdout:write(ln, " rcvd", "\n")
    end

    local ok, why = thr:join()

    if ok then
        print(why or "OK")
    else
        error(require"cqueues.errno".strerror(why))
    end
end)

assert(cq:loop())
```