PsychOS API Documentation

Contents

1 lib/doc.lua

1.1 `doc.docs(topic`$^{\texttt{string}}$`): boolean`

Displays the documentation for *topic*, returning true, or errors. Also callable as just doc().

1.2 `doc.format(fdoc`$^{\texttt{table}}$`): string`

returns VT100 formatted documentation from documentation table *fdoc*

1.3 `doc.searchers.lib(name`$^{\texttt{string}}$`): string, string`

Tries to find a documentation from a library with *name*. Returns either a string of documentation, or false and a reason.

1.4 `doc.parsefile(path`$^{\texttt{string}}$`): table`

parses file from *path* to return a documentation table

2 lib/event.lua

2.1 `event.ignore(e`$^{\texttt{string}}$`, f`$^{\texttt{function}}$`)`

stop function f running for every occurance of event e

2.2 `event.listen(e`$^{\texttt{string}}$`, f`$^{\texttt{function}}$`)`

run function f for every occurance of event e

2.3 `event.pull(t`$^{\texttt{number}}$`, ...)`

return an event, optionally with timeout t and filter $\ldots$.

3 lib/minitel.lua

3.1 `net.flisten(port`$^{\texttt{number}}$`, listener`$^{\texttt{function}}$`): function`

run function *listener* on a connection to *port*

3.2 `net.usend(to`$^{\texttt{string}}$`, port`$^{\texttt{number}}$`, data`$^{\texttt{string}}$`, npID`$^{\texttt{string}}$`)`

send an unreliable packet to host *to* on port *port* with data *data*, optionally with the packet ID *npID*

3.3 `net.genPacketID(): string`

generate a random 16-character string, for use in packet IDs

3.4 `net.open(to`$^{\texttt{string}}$`, port`$^{\texttt{number}}$`): buffer`

open a socket to host *to* on port *port*

3.5 `net.rsend(to`$^{\texttt{string}}$`, port`$^{\texttt{number}}$`, data`$^{\texttt{string}}$`, block`$^{\texttt{boolean}}$`): boolean`

send a reliable packet to host *to* on port *port* with data *data*, with *block* set to true to disable blocking

3.6 `net.send(to`$^{\texttt{string}}$`, port`$^{\texttt{number}}$`, ldata`$^{\texttt{string}}$`): boolean`

send arbitrary data *ldata* reliably to host *to* on port *port*

3.7 `net.listen(port`$^{\texttt{number}}$`): buffer`

listen for connections on port *port* in a blocking manner

4 lib/netutil.lua

4.1 `netutil.nc(host`$^{\texttt{string}}$`, port`$^{\texttt{number}}$`): boolean`

Starts an interactive Minitel socket connection to *host* on *port*, primarily for remote login. Returns whether the attempt was successful.

4.2 `netutil.exportfs(path`$^{\texttt{string}}$`): boolean`

Export the directory *path* over RPC.

4.3 `netutil.importfs(host`$^{\texttt{string}}$`, rpath`$^{\texttt{string}}$`, lpath`$^{\texttt{string}}$`): boolean`

Import filesystem *rpath* from *host* and attach it to *lpath*.

4.4 `netutil.ping(addr`$^{\texttt{string}}$`, times`$^{\texttt{number}}$`, timeout`$^{\texttt{number}}$`, silent`$^{\texttt{boolean}}$`): boolean, number, number, number`

Request acknowledgment from *addr*, waiting *timeout* seconds each try, and try *times* times. If *silent* is true, don't print status. Returns true if there was at least one successful ping, the number of successes, the number of failures, and the average round trip time.

5 lib/rc.lua

5.1 rc.start(name[string], ...): boolean, string

Stops service *name*, supplying ... to the stop function. Returns false and a reason if this fails.

5.2 rc.load(name[string], force[boolean]): table

Attempts to load service *name*, and if *force* is true, replaces the current instance.

5.3 rc.disable(name[string])

Disables service *name* being started on startup.

5.4 rc.restart(name[string])

Restarts service *name* using rc.stop and rc.start.

5.5 rc.enable(name[string])

Enables service *name* being started on startup.

5.6 rc.stop(name[string], ...): boolean, string

Stops service *name*, supplying ... to the stop function. Returns false and a reason if this fails.

6 lib/rpc.lua

6.1 `rpc.register(name`$^{\texttt{string}}$`, fn`$^{\texttt{function}}$`)`

Registers a function to be exported by the RPC library.

6.2 `rpc.proxy(hostname`$^{\texttt{string}}$`, filter`$^{\texttt{string}}$`): table`

Returns a component.proxy()-like table from the functions on *hostname* with names matching *filter*.

6.3 `rpc.call(hostname`$^{\texttt{string}}$`, fn`$^{\texttt{string}}$`, ...): boolean`

Calls exported function *fn* on host *hostname*, with parameters ..., returning whatever the function returns, or false.

7 lib/serialization.lua

7.1 `serial.serialize(value`[boolean]`, af): string`

serialize *value* into a string. If *af* is true, allow functions. This breaks unserialization.

7.2 `serial.unserialize(data`[string]`)`

return *data*, but unserialized

8 lib/vtansi.lua

8.1 `vtansi.vtemu(gpu)`

takes GPU component proxy *gpu* and returns a function to write to it in a manner like an ANSI terminal

8.2 `vtansi.vtsession(gpua, scra)`

creates a process to handle the GPU and screen address combination *gpua*/*scra*. Returns read, write and "close" functions.

9 module/buffer.lua

9.1 `buffer.new(mode, stream)`

create a new buffer in mode *mode* backed by stream object *stream*

10 module/devfs.lua

10.1 `devfs.register(fname, fopen)`

Register a new devfs node with the name *fname* that will run the function *fopen* when opened. This function should return a function for read, a function for write, function for close, and optionally, a function for seek, in that order.

11 module/fs.lua

11.1 `fs.mount(path, proxy)`

mounts the filesystem *proxy* to the mount point *path* if it is a directory. BYO proxy.

11.2 `fs.open(path, mode)`

opens file *path* with mode *mode*

11.3 `fs.segments(path)`

splits *path* on each /

11.4 `fs.type(path)`

returns the component type of the filesystem at a given path, if applicable

11.5 `fs.mounts()`

returns a table containing the mount points of all mounted filesystems

11.6 `fs.resolve(path)`

resolves *path* to a specific filesystem mount and path

11.7 `fs.rename(from, to)`

moves file *from* to *to*

11.8 `fs.address(path)`

returns the address of the filesystem at a given path, if applicable; do not expect a sensical response

11.9 `fs.copy(from, to)`

copies a file from *from* to *to*

12 module/io.lua

12.1 `io.input(fd)`

Sets the default input stream to *fd* if provided, either as a buffer as a path. Returns the default input stream.

12.2 `io.write(...)`

Writes its arguments to the default output stream.

12.3 `print(...)`

Writes each argument to the default output stream, separated by newlines.

12.4 `io.read(...)`

Reads from the default input stream.

12.5 `io.open(path, mode)`

Open file *path* in *mode*. Returns a buffer object.

12.6 `io.output(fd)`

Sets the default output stream to *fd* if provided, either as a buffer as a path. Returns the default output stream.

13 module/loadfile.lua

13.1 `loadfile(p)`

reads file *p* and returns a function if possible

13.2 `require(f, force)`

searches for a library with name *f* and returns what the library returns, if possible. if *force* is set, loads the library even if it is cached

13.3 `os.spawnfile(p, n, ...)`

spawns a new process from file *p* with name *n*, with arguments following *n*.

13.4 `runfile(p, ...)`

runs file *p* with arbitrary arguments in the current thread

14 module/osutil.lua

14.1 `os.chdir(p)`

changes the current working directory of the calling process to the directory specified in *p*, returning true or false, error

15 module/sched.lua

15.1 `os.tasks()`

returns a table of process IDs

15.2 `os.getenv(k)`

gets a process' k environment variable

15.3 `os.setenv(k, v)`

set's the current process' environment variable k to v, which is passed to children

15.4 `os.sched()`

the actual scheduler function

15.5 `os.kill(pid)`

removes process *pid* from the task list

15.6 `os.pid()`

returns the current process' PID

15.7 `os.taskInfo(pid)`

returns info on process *pid* as a table with name and parent values

15.8 `os.spawn(f, n)`

creates a process from function f with name n

16 module/syslog.lua

16.1 syslog(msg[string], level[number], service[string])

Output *msg* to the system log, with severity *level*, from *service*.