

EGGDROP

TCL 1.6

This is an exhaustive list of all the Tcl commands added to Eggdrop. All of the normal Tcl built-in commands are still there, of course. But you can also use these to manipulate features of the bot. They are listed according to category.

NOTICE: This list is accurate for Eggdrop v1.6.8! Scripts written for v1.3/v1.4 series of Eggdrop should probably work with a few minor modifications depending on the script.

Scripts which were written for v0.9, v1.0, v1.1 or v1.2 will probably not work without modification. Commands which have been changed in the v1.6 series (or are just new commands) are marked with vertical bars on the left.

This entire document is the tcl-commands.doc that comes with eggdrop 1.6.8, with the exception of “(E) TEXTFILE SUBSTITUTION” which is text-substitutions.doc that comes with the program.

Copyright © 1999-2002 by Eggheads Development Team <eggdev@eggheads.org>
Layout by Nils Østbjerg <shorty@business.auc.dk>

CORE EGGDROP COMMANDS

These are commands provided in the core part of Eggdrop, for module specific commands: see later.

OUTPUT COMMANDS

putserv <text> [options]

sends text to the server, like 'dump' (intended for direct server commands); output is queued so that you won't flood yourself off the server. options are:

- next push messages to the front of the queue

returns: nothing

puthelp <text> [options]

sends text to the server like 'putserv', but uses a different queue (intended for sending messages to channels or people). options are:

- next push messages to the front of the queue

returns: nothing

putquick <text> [options]

sends text to the server, like 'dump' (intended for direct server commands); output is queued so that you won't flood yourself off the server, using the MODE queue (should be a lot faster). options are:

- next push messages to the front of the queue

returns: nothing

putkick <channel> <nick,nick,...> [reason]

sends kicks to the server and tries to put as many nicks into one kick command as possible.

returns: nothing

putlog <text>

sends text to the log for any channel, marked as 'misc' (o)

returns: nothing

putcmdlog <text>

sends text to the log for any channel, marked as 'command' (c)

returns: nothing

putxferlog <text>

sends text to the log for any channel, marked as 'file-area' (x)

returns: nothing

putloglev <level(s)> <channel> <text>

sends text to the log, tagged with all of the valid levels given (use "*" to indicate all log levels)

returns: nothing

dumpfile <nick> <filename>

dumps out a file from the help/text directory to a user on IRC via msg (one line per msg); the user has no flags, so the flag bindings wont work within the file.

queuesize [queue]

returns: the number of msgs in all queues. If a queue is specified, only the size of this queue is returned. valid queues are: mode, server, help.

clearqueue <queue>

valid arguments for queue are: mode, server, help, or all.

returns: number of deleted lines from the specified queue

USER RECORD MANIPULATION COMMANDS

countusers

returns: number of users in the bot's database

validuser <handle>

returns: "1" if a user by that name exists; "0" otherwise

finduser <nick!user@host>

finds the user record which most closely matches the given user@host

returns: the handle found, or "*" if none

userlist [flags]

you can use the new flag matching system here, usage: [global]{&/}[chan]{&/}[bot] matches the flags relevantly, (chan matches vs anywhere), & specifies and when match, | specifies or. only the first of these is relevant, the default is or.

returns: a list of the handles of users on the bot

passwdok <handle> <pass>

checks the password given against the user's password check against the password "" (a blank string) or '-' to find out if a user has no password set.

returns: "1" if password matches for that user; "0" if not

getuser <handle> <entry-type> [extra info]

this is a generic interface to the new generic userfile support, it return info specific to each entry-type, valid entry types are:

- BOTFL - **returns:** the current bot-specific flags for the user (if it's a bot :)
- BOTADDR - (another bot-only thing :) **returns:** a list containing the bot's address, the bot's telnet port, and its relay port.
- HOSTS - **returns:** a list of the host for the user
- LASTON - **returns:** a list containing the unixtime last seen, and the last seen place.
- OR LASTON #channel - **returns:** the time last seen on the channel or 0 if no info
- INFO - **returns:** the user's global info line
- XTRA - **returns:** the old xtra info
- COMMENT - **returns:** the master-visible only comment for the user
- EMAIL - **returns:** the users email address
- URL - **returns:** the users url address
- HANDLE - **returns:** the users handle as it is saved in the userfile

setuser <handle> <entry-type> [extra info]

this is the counterpart of getuser, it lets you set the various values extra ones not supported about :

- PASS - use this to set a users password (no 3rd arg will clear it)
- HOSTS - for setting hosts, no extra info = clear, otherwise **1** hostmask is added :P
- LASTON - 2 forms:
 - setuser <handle> laston <unixtime> <place> - sets global laston time
 - setuser <handle> laston <unixtime> - sets global laston time, leaving the place field empty
 - setuser <handle> laston <unixtime> <channel> - will set a users laston time in a channel record (if it already exists)

chandle <old-handle> <new-handle>

changes a user's handle

returns: "1" on success; "0" if the handle is already used, the handle is invalid, or the user can't be found

chattr <handle> [changes [channel]]

changes the attributes for a user record, if you include any – changes are of the form '+f', '-o', '+dk', '-o+d', etc; if a channel is specified, the channel-specific flags for that channel are altered you can now use the +o|o #channel format here too.

returns: new flags for the user (if you made no changes, returns current flags); if a channel was specified, the global AND the channel-specific flags for that channel are returned in the format of the new flagging system (globalflags|channelflags) -- returns "*" if that user does not exist

botattr <handle> [changes [channel]]

similar to chattr except for bot attributes rather than normal user attributes, this includes the channel-specific +s share flag

matchattr <handle> <flags> [channel]

returns: "1" if the specified user has the matching flags. (using the new matching system)

adduser <handle> [hostmask]

creates a new user entry with the handle and hostmask given (with no password, and the default flags)

returns: "1" if successful, "0" if it already existed

addbot <handle> <address>

creates a new bot entry with the handle and bot linking address given (with no password and no flags)

returns: "1" if successful, "0" if it already existed

deluser <handle>

attempts to erase a user record with that handle

returns: "1" if successful, "0" if no such user exists

delhost <handle> <hostmask>

deletes a hostmask from a user's hostmask list

returns: "1" on success, "0" if that hostmask wasn't in the list or the user does not exist

addchanrec <handle> <channel>

add a channel record for the user

returns: "1" on success, "0" if the user does not exist or if there isn't such a channel

delchanrec <handle> <channel>

removes a channel record for the user; this includes all associated channel flags

returns: "1" on success, "0" if the user does not exist or if there isn't such a channel

haschanrec <handle> <channel>

returns: "1" if handle has a chanrec for that channel, "0" otherwise

getchaninfo <handle> <channel>

returns: info line for a specific channel (behaves just like 'getinfo')

setchaninfo <handle> <channel> <info>

sets the info line on a specific channel for a user if info is "none" it will be removed.

returns: nothing

newchanban <channel> <ban> <creator> <comment> [lifetime] [options]

adds a ban to the enforced ban list of a channel; creator is given credit for the ban in the ban list; lifetime is specified in minutes; if lifetime is not specified, ban-time (usually 60) is used; setting the lifetime to 0 makes it a permanent ban; valid options are:

- sticky forces the ban to be always active on a channel, even with dynamic bans on
- none (no effect)

returns: nothing

newban <ban> <creator> <comment> [lifetime] [options]

adds a ban to the global ban list (which takes effect on all channels); other arguments work exactly like newchanban

returns: nothing

newchanexempt <channel> <exempt> <creator> <comment> [lifetime] [options]

adds a exempt to the enforced exempt list of a channel; creator is given credit for the exempt in the exempt list; lifetime is specified in minutes; if lifetime is not specified, exempt-time (usually 60) is used; setting the lifetime to 0 makes it a permanent exempt; valid options are:

- sticky forces the exempt to be always active on a channel, even with dynamicexempts on
- none (no effect)

returns: nothing

N.B. The exempt will not be removed until the corresponding ban has been removed. For timed bans once the time period has expired the exempt will not be removed until the corresponding ban has either expired or removed

newexempt <exempt> <creator> <comment> [lifetime] [options]

adds a exempt to the global exempt list (which takes effect on all channels); other arguments work exactly like newchanexempt

returns: nothing

newchaninvite <channel> <invite> <creator> <comment> [lifetime] [options]

adds a invite to the enforced invite list of a channel; creator is given credit for the invite in the invite list; lifetime is specified in minutes; if lifetime is not specified, invite-time (usually 60) is used; setting the lifetime to 0 makes it a permanent invite; valid options are:

- sticky forces the invite to be always active on a channel, even with dynamicinvites on
- none (no effect)

returns: nothing

N.B. The invite will not be removed until the channel has gone -i.

newinvite <invite> <creator> <comment> [lifetime] [options]

adds a invite to the global invite list (which takes effect on all channels); other arguments work exactly like newchaninvite

returns: nothing

stick <banmask> [channel]

makes a ban sticky, or if a channel is specified, then it is set sticky on that channel.

returns: "1" is successful, "0" otherwise

unstick <banmask> [channel]

makes a ban no longer sticky, or if a channel is specified, then it is unstuck on that channel.

returns: "1" is successful, "0" otherwise

stickexempt <exemptmask> [channel]

makes an exempt sticky, or if a channel is specified, then it is set sticky on that channel.

returns: "1" is successful, "0" otherwise

unstickexempt <exemptmask> [channel]

makes an exempt no longer sticky, or if a channel is specified, then it is unstuck on that channel.

returns: "1" is successful, "0" otherwise

stickinvite <invitemask> [channel]

makes an invite sticky, or if a channel is specified, then it is set sticky on that channel.

returns: "1" is successful, "0" otherwise

unstickinvite <invitemask> [channel]

makes an invite no longer sticky, or if a channel is specified, then it is unstuck on that channel.

returns: "1" is successful, "0" otherwise

killchanban <channel> <ban>

removes a ban from the enforced ban list for a channel

returns: "1" if successful, "0" otherwise

killban <ban>

removes a ban from the global ban list

returns: "1" if successful, "0" otherwise

killchanexempt <channel> <exempt>

removes a exempt from the enforced exempt list for a channel

returns: "1" if successful, "0" otherwise

killexempt <exempt>

removes a exempt from the global exempt list

returns: "1" if successful, "0" otherwise

killchaninvite <channel> <invite>

removes a invite from the enforced invite list for a channel

returns: "1" if successful, "0" otherwise

killinvite <invite>

removes a invite from the global invite list

returns: "1" if successful, "0" otherwise

ischanjuped [channel]

returns: "1" if the channel is juped and bot is unable to join, "0" otherwise

isban <ban> [channel]

returns: "1" if that ban is in the global ban list, "0" otherwise; if a channel is specified, that channel's ban list is checked too

ispermban <ban> [channel]

returns: "1" if that ban is in the global ban list AND is marked as permanent, "0" otherwise; if a channel is specified, that channel's ban list is checked too

isexempt <exempt> [channel]

returns: "1" if that exempt is in the global exempt list, "0" otherwise; if a channel is specified, that channel's exempt list is checked too

ispermexempt <exempt> [channel]

returns: "1" if that exempt is in the global exempt list AND is marked as permanent, "0" otherwise; if a channel is specified, that channel's exempt list is checked too

isinvite <invite> [channel]

returns: "1" if that invite is in the global invite list, "0" otherwise; if a channel is specified, that channel's invite list is checked too

isperminvite <invite> [channel]

returns: "1" if that invite is in the global invite list AND is marked as permanent, "0" otherwise; if a channel is specified, that channel's invite list is checked too

isbansticky <ban> [channel]

returns: "1" if that ban is a sticky ban in the global ban list, "0" otherwise; if a channel is specified, that channel's ban list is checked too

isexemptsticky <exempt> [channel]

returns: "1" if that exempt is a sticky exempt in the global exempt list, "0" otherwise; if a channel is specified, that channel's exempt list is checked too

isinvitesticky <invite> [channel]

returns: "1" if that invite is a sticky invite in the global invite list, "0" otherwise; if a channel is specified, that channel's invite list is checked too

matchban <nick!user@host> [channel]

returns: "1" if that user address matches a ban in the global ban list, "0" otherwise; if a channel is specified, that channel's ban list is checked too

matchexempt <nick!user@host> [channel]

returns: "1" if that user address matches a exempt in the global exempt list, "0" otherwise; if a channel is specified, that channel's exempt list is checked too

matchinvite <nick!user@host> [channel]

returns: "1" if that user address matches a invite in the global invite list, "0" otherwise; if a channel is specified, that channel's invite list is checked too

banlist [channel]

returns: list of global bans, or (if a channel is specified) list of channel-specific bans; each entry is itself a list, containing: hostmask, comment, expiration timestamp, time added, last time active, and creator (the three timestamps are in unixtime format)

exemptlist [channel]

returns: list of global exempts, or (if a channel is specified) list of channel-specific exempts; each entry is itself a list, containing: hostmask, comment, expiration timestamp, time added, last time active, and creator (the three timestamps are in unixtime format)

invitelist [channel]

returns: list of global invites, or (if a channel is specified) list of channel-specific invites; each entry is itself a list, containing: hostmask, comment, expiration timestamp, time added, last time active, and creator (the three timestamps are in unixtime format)

newignore <hostmask> <creator> <comment> [lifetime]

adds an entry to the ignore list; creator is given credit for the ignore; lifetime is how many minutes until the ignore expires and is removed; if lifetime is not specified, ignore-time (usually 60) is used; setting the lifetime to 0 makes it a permanent ignore

returns: nothing

killignore <hostmask>

removes an entry from the ignore list

returns: "1" if successful, "0" otherwise

ignorelist

returns: list of ignores; each entry is itself a list, containing: hostmask, comment, expiration timestamp, time added, and creator (the timestamps are in unixtime format)

isignore <hostmask>

returns: "1" if the ignore is in the list, "0" otherwise

save

writes the user and channel files to disk

returns: nothing

reload

loads the userfile from disk (replacing whatever is in memory)

returns: nothing

backup

makes a simple backup of the userfile that's on disk if the channels module is loaded, this also makes a simple backup of the channel file.

returns: nothing

getting-users

returns: "1" if the bot is currently downloading a userfile from a sharebot (and hence, user records are about to drastically change), "0" if not

CHANNEL COMMANDS

channel add <name> <option-list>

adds a channel record for the bot to monitor; the full list of possible options are given in the eggdrop.complete.conf sample config file; note that the channel options must be in a list (enclosed in {})

returns: nothing

channel set <name> <options...>

sets options for the channel specified; the full list of possible options are given in the eggdrop.complete.conf sample config file

returns: nothing

channel info <name>

returns: list of info about that channel record: enforced mode, idle kick limit, need-op script, need-invite script, and then various +/- options as seen in the config file

channel remove <name>

destroys a channel record for the bot and makes the bot no longer monitor that channel

returns: nothing

savechannels

saves the channel settings to the channel-file if one is defined.

returns: nothing

loadchannels

reloads the channel settings from the channel-file if one is defined.

returns: nothing

channels

returns: list of the channels the bot is monitoring (or trying to)

channame2dname <channel-name>

chandname2name <channel-dname>

These two functions are important to correctly support !channels. The bot differentiates between channel description names (chan dnames) and real channel names (chan names). The chan dnames are what you would normally call the channel; "!channel". The chan names are what the IRC server uses to identify the channel, they consist of the chan dname prefixed with an ID; "!ABCDEchannel".

For bot functions like isop, isvoice, etc. you need to know the chan dnames. If you communicate with the server, you usually get the chan name though. That's what you need the channame2dname function for.

If you only have the chan dname and want to directly send raw server commands, you need the chan name. Use the chandname2name function for that case.

NOTE: For non-!channels, chan dname and chan name are the same.

isbotnick <nick>

returns: "1" if the nick matches the botnick; "0" otherwise

botisop [channel]

returns: "1" if the bot is an op on that channel (or a channel if no channel is specified); "0" otherwise

botisvoice [channel]

returns: "1" if the bot is has a voice on that channel (or a channel if no channel is specified); "0" otherwise

botonchan [channel]

returns: "1" if the bot is on that channel (or a channel if no channel is specified); "0" otherwise

isop <nickname> [channel]

returns: "1" if someone by that nickname is on the channel (or a channel if no channel name is specified) and has chop; "0" otherwise

wasop <nickname> <channel>

returns: "1" if someone that just got opped/deopped in the chan had op before the modechange; "0" otherwise

isvoice <nickname> [channel]

returns: "1" if someone by that nickname is on the channel (or a channel if no channel is specified) and has voice (+v); "0" otherwise

onchan <nickname> [channel]

returns: "1" if someone by that nickname is on the bot's channel (or channels if none is specified); "0" otherwise

nick2hand <nickname> [channel]

returns: handle of <nickname> on <channel>, if <channel> is not specified, bot will check all of its channels, if <nickname> not found, returns "", if <nickname> found but unknown, returns "*"

handonchan <handle> [channel]

returns: "1" if the the user@host for someone on the channel (or a channel if no channel name is specified) matches for the handle given; "0" otherwise

hand2nick <handle> [channel]

returns: nickname of the first person on the <channel> whose user@host matches that handle, if there is one; "" otherwise if <channel> is not specified, bot will check all of its channels

ischanban <ban> <channel>

returns: "1" if that is a ban on the bot's channel

ischanexempt <exempt> <channel>

returns: "1" if that is an exemption (+e mode) on the bot's channel. this is only useful on networks that support +e

ischaninvite <invite> <channel>

returns: "1" if that is an invitation (+I mode) on the bot's channel. this is only useful on networks that support +I

chanbans <channel>

returns: a list of the current bans on the channel, each element is of the form {ban bywho age} age is seconds from the bot's POV

chanexempts <channel>

returns: a list of the current exemptions (+e mode) on the channel, each element is of the form {exemption bywho age} age is seconds from the bot's POV. this is only useful on networks that support +e

chaninvites <channel>

returns: a list of the current invitations (+I modes) on the channel, each element is of the form {invitation bywho age} age is seconds from the bot's POV. this is only useful on networks that support +I

resetbans <channel>

removes all bans on the channel that aren't in the bot's ban list, and refreshes any bans that should be on the channel but aren't.

returns: nothing

resetexempts <channel>

removes all exemptions on the channel. this is an IRCNET feature.

resetinvites <channel>

removes all invitations on the channel. this is an IRCNET feature.

resetchan <channel>

rereads in the channel info from the server

returns: nothing

getchanhost <nickname> [channel]

returns: user@host of <nickname>, if <channel> is not specified, bot will check all of its channels, if <nickname> is not on the channel(s), returns ""

getchanjoin <nickname> <channel>

returns: timestamp of when that person joined the channel

onchansplit <nick> [channel]

returns: "1" if that nick is split from the channel (or a channel if no channel is specified); "0" otherwise

chanlist <channel> [flags[&chanflags]]

flags are any flags that are global flags, the '&' denotes to look for channel specific flags. Examples:

- n (Botowner)
- &n (Channel owner)
- o&m (Global op, Channel master)

now you can use even more complex matching of flags, including +&- flags and & or | (and or or) matching

returns: list of nicknames currently on the bot's channel that have all of the flags specified; if no flags are given, all of the nicknames are returned.

Please note, that if you're executing chanlist after a part or sign bind, the gone user will still be listed so you can check for wasop, isop, etc.

getchanidle <nickname> <channel>

returns: number of minutes that person has been idle; "0" if the specified user isn't even on the channel

getchanmode <channel>

returns: string of the type "+ntik key" for the channel specified

jump [server [port [password]]]

jumps to the server specified, or (if none is specified) the next server in the list

returns: nothing

pushmode <channel> <mode> [arg]

sends out a channel mode change (ex: pushmode #lame +o goober) through the bot's queuing system; all the mode changes will be sent out at once (combined into one line as much as possible) after the script finishes, or when 'flushmode' is called.

flushmode <channel>

forces all previously pushed channel mode changes to go out right now, instead of when the script is done (just for the channel specified)

topic <channel>

returns: string of the current topic on the specified channel

validchan <channel>

checks if the bot is monitoring that channel

returns: 1 if the channel exists, 0 if not

isdynamic <channel>

returns: 1 if the channel is an existing dynamic channel, 0 if not

setundef <flag/int> <name>

initializes a user defined channel flag or integer setting. You can use it like any other flag/setting.

IMPORTANT: Don't forget to reinitialize your flags/settings after a restart, or it'll be lost.

renundef <flag/int> <oldname> <newname>

renames a user defined channel flag or integer setting.

delundef <flag/int> <name>

deletes a user defined channel flag or integer setting.

DCC COMMANDS

putdcc <idx> <text>

sends text to the dcc user indicated

returns: nothing

dccbroadcast <message>

sends your message to everyone on the party line on the bot net, in the form "*** <message>" for local users, and "*** [Bot] <message>" for users on other bots

dccputchan <channel> <message>

sends your message to everyone on a certain channel on the bot net, in a form exactly like dccbroadcast does -- valid channels are 0 through 99999

returns: nothing

boot <user@bot> [reason]

boots a user from the partyline

returns: nothing

restart

rehashes the bot and kills all timers. note that as this will unload and reload all the modules and your bot will jump servers.

returns: nothing

rehash

rehashes the bot

returns: nothing

dccsimul <idx> <text...>

simulates text typed in by the dcc user specified -- note that in v0.9, this only simulated commands; now a command must be preceded by a '!' to be simulated

returns: nothing

hand2idx <handle>

returns: the idx (a number greater than or equal to zero) for the user given, if she is on the party line in chat mode (even if she is currently on a channel or in chat off), the file area, or in the control of a script; "-1" otherwise -- if the user is on multiple times, the oldest idx is returned

idx2hand <idx>

returns: handle of the user with that idx

valididx <idx>

returns: "1" if the idx currently exists; "0" if not

getchan <idx>

returns: the current party line channel for a user on the party line; "0" indicates he's on the group party line, "-1" means he has chat off, and a value from 1 to 99999 is a private channel

setchan <idx> <channel>

sets a party line user's channel rather suddenly (the party line user is not notified that she is now on a new channel); a channel name can be used (provided it exists)

returns: nothing

console <idx> [channel] [console-modes]

changes a dcc user's console mode, either to an absolute mode (like "mpj") or just adding/removing flags (like "+pj" or "-moc" or "+mp-c"); the user's console channel view can be changed also (as long as the new channel is defined in the bot)

returns: a list containing the user's (new) channel view, and (new) console mode, or nothing if that user isn't currently in dcc chat

echo <idx> [status]

turns a user's echo on or off; the status has to be a 1 or 0

returns: new value of echo for that user (or the current value, if status was omitted)

putbot <bot-nick> <message>

sends a message across the bot-net to another bot; if no script intercepts the message on the other end, the message just vanishes

returns: nothing

putallbots <message>

broadcasts a message across the bot-net to all currently connected bots

returns: nothing

killdcc <idx>

kills a party-line or file area connection, rather abruptly

returns: nothing

bots

returns: list of the bots currently connected to the botnet

botlist

returns: a list of bots currently on the botnet; each item in the list will be a sublist with four elements: bot, uplink, version, sharing status.

- bot : the bot's nickname
- uplink: who the bot is connected through
- version: its current numeric version
- sharing: a "+" if the bot is sharing, "-" otherwise

islinked <bot>

returns: "1" if the bot is currently linked, "0" otherwise

dccused

returns: number of dcc connections currently in use

dcclist ?type?

returns: a list of active connections, each item in the list will be a sublist of six elements:

{<idx> <handle> <hostname> <type> {<other>} <timestamp>}

the types are: chat, bot, files, file_receiving, file_sending, file_send_pending, script, socket (these are connections that have not yet been put under 'control'), telnet, and server.

whom <chan>

returns: list of people on the botnet who are on that channel (0 is the default party line); each item in the list will be a sublist with six elements: nickname, bot, hostname, access flag ('-', '@', '+', or '*'), minutes idle, and away message (blank if the user is not away) if you specify a channel of * every user on the botnet is returned with an extra argument indicating the channel the user is on

getdccidle <idx>

returns: number of seconds the dcc chat/file system/script user has been idle

getdccaway <idx>

returns: away message for a dcc chat user (or "" if the user is not set away)

setdccaway <idx> <message>

sets a party line user's away message and marks them away; if set to "", the user is marked un-away

returns: nothing

connect <host> <port>

makes an outgoing connection attempt and creates a dcc entry for it; a 'control' command should be used immediately after a successful 'connect' so no input is lost

returns: idx of the new connection

listen <port> <type> [options] [flag]

opens a listening port to accept incoming telnets; type must be one of "bots", "all", "users", "script", or "off":

listen <port> bots [mask]

accepts connections from bots only; the optional mask is used to identify permitted bot names; if the mask begins with '@' it is interpreted to be a mask of permitted hosts to accept connections from

returns: port #

listen <port> users [mask]

accepts connections from users only (no bots); the optional mask is used to identify permitted nicknames; if the mask begins with '@' it is interpreted to be a mask of permitted hosts to accept connections from

returns: port #

listen <port> all [mask]

accepts connections from anyone; the optional mask is used to identify permitted nicknames/botnames; if the mask begins with '@' it is interpreted to be a mask of permitted hosts to accept connections from

returns: port #

listen <port> script <proc> [flag]

accepts connections which are immediately routed to a proc; the proc is called with one parameter: the idx of the new connection flag may currently only be 'pub', which makes the bot allow anyone to connect.

returns: port #

listen <port> off

stop listening at a port

returns: nothing

dccdumpfile <idx> <filename>

dumps out a file from the text directory to a dcc chat user; the flag matching that's used everywhere else works here too

MISCELLANEOUS COMMANDS

bind <type> <attr(s)> <command-name> [proc-name]

adds a new keyword command to the bot; valid types are listed below; the <attr(s)> are the flags that a user must have to trigger this command; the <command-name> for each type is listed below; <proc-name> is the name of the Tcl procedure to call for this command (see below for the format of the procedure call); if the proc-name is omitted, no binding is added -- instead, the current binding is returned (if it's stackable, a list of the current bindings is returned) yes, you can use the new flag binding method here too, and this is where it becomes truly phearfull since you may never need to check attributes inside functions again...imagine:

```
bind pub -o&+o command command_proc
```

to only allow channel-spec ops to use it! no problem! works fine!

returns: name of the command that was added, or (if proc-name was omitted), a list of the current bindings for this command

unbind <type> <attr(s)> <command-name> <proc-name>

removes a previously-made binding

returns: name of the command that was removed

binds ?type/mask?

returns: a list of Tcl binds, each item in the list will be a sublist of five elements:

```
{<type> <flags> <name> <hits> <proc>}
```

logfile [<modes> <channel> <filename>]

creates a new logfile, which will log the modes given for the channel listed -- or, if no logfile is specified, just returns a list of logfiles; "*" can be used to mean all channels; you can also change the modes and channel of an existing logfile with this command. Entering a blank mode and channel makes the bot stop logging there.

returns: filename of logfile created, or (if no logfile is specified) a list of logfiles like: "{mco * eggdrop.log} {jp #lame lame.log}"

maskhost <nick!user@host>

returns: hostmask for the string given ("n!u@1.2.3.4" -> "!u@1.2.3.*", "n!u@lame.com" -> "!u@lame.com", "n!u@a.b.edu" -> "!u@*.b.edu")

timer <minutes> <tcl-command>

executes the Tcl command after a certain number of minutes have passed

returns: a timerID

utimer <seconds> <tcl-command>

executes the Tcl command after a certain number of seconds have passed

returns: a timerID

timers

returns: list of active minutely timers; each entry in the list contains the number of minutes left till activation, the command that will be executed, and the timerID

utimers

returns: list of active secondly timers, identical in format to the output from 'timers'

killtimer <timerID>

removes a minutely timer from the list

returns: nothing

killutimer <timerID>

removes a secondly timer from the list

returns: nothing

unixtime

returns: a long integer which is the current time according to unix

duration <seconds>

returns: the number of seconds converted into years, weeks, days, hours, minutes, and seconds. 804600 seconds is turned into 1 week 2 days 7 hours 30 minutes.

strftime <formatstring> [time]

returns: a formatted string of time using standard strftime format, uses the value of time, or now if no time specified

ctime <unixtime>

returns: a formatted date/time string based on the current locale settings, from the unix time string given; "Fri Aug 3 11:34:55 1973"

myip

returns: a long number representing the bot's IP address, as it might appear in (for example) a DCC request

rand <limit>

returns: a random integer between 0 and limit-1

control <idx> <command>

removes a user from the party line and sends all future input from them to the Tcl command given; the command will be called with two parameters: the idx of the user, and the input text; the command should return "0" to indicate success and "1" to indicate that it relinquishes control of the user back to the bot; the idx must be for a user in the party line area or the file area; if the input text is blank (""), it indicates that the dcc user has dropped connection. Also, if the input text is blank, never call killdcc on it, it will fail with "invalid idx".

returns: nothing

sendnote <from> <to> <message>

simulates what happens when one user sends a note to another (this can also do cross-bot notes)

returns: "1" if the note was delivered locally or sent to another bot, "2" if the note was stored locally, "3" if the user's notepad is too full to store a note, "4" if a Tcl binding caught the note, "5" if the note was stored because the user is away, or "0" if the send failed

link [via-bot] <bot>

attempts to link to another bot directly (or, if you give a via-bot, it tells the via-bot to try)

returns: "1" if it looks okay and it will try; "0" if not

unlink <bot>

attempts to remove a bot from the botnet

returns: "1" if it will try or has passed the request on; "0" if not

encrypt <key> <string>

returns: encrypted string (using blowfish), encoded into ascii using base-64 so it can be sent over the botnet

decrypt <key> <encrypted-base64-string>

returns: decrypted string (using blowfish)

encpass <password>

returns: encrypted string (using blowfish)

die [reason]

causes the bot to log a fatal error and exit completely; if no reason is given, "EXIT" is used

unames

returns: The current operating system the bot is using.

dnslookup <ip-address/hostname> <proc> [[arg1] [arg2] ... [argN]]

This issues an asynchronous dns lookup request. The command will block if dns module is not loaded; otherwise it will either return immediately or immediately call the specified proc (e.g. if the lookup is already cached). As soon as the request completes, <proc> will be called as follows:

<proc> ipaddress hostname status [[arg1] [arg2] [argN]]

status is 1 if the lookup was successful and 0 if it wasn't. All additional parameters (called arg1, arg2 and argN above) get appended to the proc's other parameters.

md5 <string>

returns: the 128 bit MD5 message-digest of the specified string.

callevent <event>

Triggers the event bind manually for a certain event. For example: callevent rehash

traffic

returns: a list of sublists containing information about the bot's traffic usage in bytes. Each sublist contains five elements: type, in-traffic today, in-traffic total, out-traffic today, out-traffic total.

GLOBAL VARIABLES:

(All config-file variables are global, too. But these variables are set by the bot.)

botnick

current nickname the bot is using; "Valis", "Valis0", etc

botname

current nick!user@host that the server sees; "Valis!valis@crappy.com"

server

current server the bot is using; "irc.math.ufl.edu:6667"

version

current bot version "1.1.2+pl1 1010201 pl1"; first item is the text version, second item is a numerical version, and any following items are the names of patches that have been added

numversion

current numeric bot version "1010201"; Numerical version is "MNNRRPP" where:

- M is the Major release number
- NN is the Minor release number
- RR is the sub-release number
- PP is the patch level for that sub-release

uptime

unixtime value for when the bot was started

server-online

unixtime value for when the bot connected to its current server

lastbind

The last command binding which triggered. This allows you to identify which command triggered a Tcl routine.

isjuped

value is 1 if bot's nick is juped(437) 0 otherwise.

hand-len

the value of the HANDLEN setting in src/eggdrop.h

COMMAND EXTENSION:

You can use the 'bind' command to attach Tcl procedures to certain events. For example, you can write a Tcl procedure that gets called every time a user says "danger" on the channel. The following is a list of the types of bindings, and how they work. Under each binding type is the format of the bind command, the list of arguments sent to the Tcl proc, and an explanation.

Some bindings are marked as "stackable". That means that you can bind multiple commands to the same trigger. Normally, for example, a binding of 'bind msg - stop msg_stop' (which makes a msg-command "stop" call the Tcl proc "msg_stop") will overwrite any previous binding you had for the msg-command "stop". With stackable bindings, like 'msgm' for example, you can bind to the same command or mask again and again. When the binding is triggered, ALL the Tcl procs that are bound to it will be called, one after another.

To remove a binding, use 'unbind'. For example, to remove that binding for the msg-command "stop", use 'unbind msg - stop msg_stop'.

(1)MSG

```
bind msg <flags> <command> <proc>
procname <nick> <user@host> <handle> <text>
```

used for /msg commands; the first word of the user's msg is the command, and everything else becomes the argument string

(2)DCC

```
bind dcc <flags> <command> <proc>
procname <handle> <idx> <text>
```

used for commands from a dcc chat on the party line; as in MSG, the command is the first word and everything else is the argument string; the idx is valid until the user disconnects; after that it may be reused, to be careful about storing an idx for long periods of time

(3)FIL

```
bind fil <flags> <command> <proc>
procname <handle> <idx> <text>
```

the same as DCC, except this is triggered if the user is in the file area instead of the party line

(This is only available when the 'filesys' module is loaded.)

(4)PUB

```
bind pub <flags> <command> <proc>
procname <nick> <user@host> <handle> <channel> <text>
```

used for commands given on a channel; just like MSG, the first word becomes the command and everything else is the argument string

(5)MSGM (stackable)

```
bind msgm <flags> <mask> <proc>
procname <nick> <user@host> <handle> <text>
```

matches the entire line of text from a /msg with the mask; this is more useful for binding Tcl procs to words or phrases spoken anywhere within a line of text

(6)PUBM (stackable)

```
bind pubm <flags> <mask> <proc>
procname <nick> <user@host> <handle> <channel> <text>
```

just like MSGM, except it's triggered by things said on a channel instead of things /msg'd to the bot; the mask is matched against the channel name followed by the text, "#nowhere hello there!", and can contain wildcards

(7)NOTC (stackable)

```
bind notc <flags> <mask> <proc>
procname <nick> <user@host> <handle> <text> <dest>
```

destination will be a nickname (the bot's nickname, obviously) or a channel name; matches the entire line of text from a /notice with the mask; it is considered a breach of protocol to respond to a /notice on IRC, so this is intended for internal use (logging, etc) only.

new Tcl procs should be declared as

```
proc notcproc {nick uhost hand text {dest ""}} {
    global botnick; if {$dest == ""} {set dest $botnick}
    ...
}
```

for compatibility

(8)JOIN (stackable)

```
bind join <flags> <mask> <proc>
procname <nick> <user@host> <handle> <channel>
```

triggered by someone joining the channel; the <mask> in the bind is matched against "#channel nick!user@host" and can contain wildcards

(9)PART (stackable)

```
bind part <flags> <mask> <proc>
procname <nick> <user@host> <handle> <channel> <msg>
```

triggered by someone leaving the channel; as in JOIN, the <mask> is matched against "#channel nick!user@host" and can contain wildcards. if no part message is specified msg will be set to "".

new Tcl procs should be declared as

```
proc partproc {nick uhost hand chan {msg ""}} {
    ...
}
```

for compatibility

(10) SIGN (stackable)

bind sign <flags> <mask> <proc>

procname <nick> <user@host> <handle> <channel> <reason>

triggered by a signoff, or possibly by someone who got netsplit and never returned; the signoff message is the last argument to the proc; wildcards can be used in <mask>, which contains 'channel + nick!user@host'

(11) TOPC (stackable)

bind topc <flags> <mask> <proc>

procname <nick> <user@host> <handle> <channel> <topic>

triggered by a topic change; can use wildcards in <mask>, which is matched against the channel name and new topic

(12) KICK (stackable)

bind kick <flags> <mask> <proc>

procname <nick> <user@host> <handle> <channel> <target> <reason>

triggered when someone is kicked off the channel; the <mask> is matched against "#channel target" where the target is the nickname of the person who got kicked off (can use wildcards); the proc is called with the nick, user@host, and handle of the kicker, plus the channel, the nickname of the person who was kicked, and the reason; <flags> is unused here

(13) NICK (stackable)

bind nick <flags> <mask> <proc>

procname <nick> <user@host> <handle> <channel> <newnick>

triggered when someone changes nicknames; wildcards are allowed; the mask is matched against "#channel newnick"

(14) MODE (stackable)

bind mode <flags> <mask> <proc>

proc-name <nick> <user@host> <handle> <channel> <mode-change> <victim>

mode changes are broken down into their component parts before being sent here, so the <mode-change> will always be a single mode, like "+m" or "-o" and victim will show the value of the mode change (for o/v/b) otherwise ""; flags are ignored; the bot's automatic response to a mode change will happen AFTER all matching Tcl procs are called; the <mask> will have the channel prefixed ("#turtle +m") if it is a server mode <nick> will be "", <user@host> is the server address and handle is *

note that "victim" was added in 1.3.23 and that this will break Tcl scripts that were written for pre-1.3.23 versions and use this binding. An easy fix (by guppy) is as follows (example):

Old script looks as follows:

```
bind mode - * mode_proc
proc mode_proc {nick uhost hand chan mc} { ... }
```

To make it work with 1.3.23+ and stay compatible with older bots, do:

```
bind mode - * mode_proc_fix

proc mode_proc_fix {nick uhost hand chan mc {victim ""}} {
    if {$victim != ""} {append mc " $victim"}
    mode_proc $nick $uhost $hand $chan $mc
}

proc mode_proc {nick uhost hand chan mc} { ... }
```

(15) CTCP

```
bind ctcp <flags> <keyword> <proc>
proc-name <nick> <user@host> <handle> <dest> <keyword> <arg...>
```

destination will be a nickname (the bot's nickname, obviously) or a channel name; keyword is the ctcp command and arg may be empty; if the proc returns 0, the bot will attempt its own processing of the ctcp command

(16) CTCR

```
bind ctc <flags> <keyword-mask> <proc>
proc-name <nick> <user@host> <handle> <dest> <keyword> <text...>
```

just like ctcp, but this is triggered for a ctcp-reply (ctcp embedded in a notice instead of a privmsg)

(17) RAW (stackable)

```
bind raw <flags> <keyword-mask> <proc>
procname <from> <keyword> <text...>
```

previous versions of Eggdrop required a special compile option to enable this binding, but it's now standard; the mask is checked against the keyword (either a numeric like "368" or a keyword like "PRIVMSG"); from will be the server name or the source user (depending on the keyword); flags are ignored; the order of the arguments is identical to the order that the IRC server sends to the bot – the pre-processing only splits it apart enough to determine the keyword; if the proc returns 1, Eggdrop will not process the line any further

(This could cause your bot to behave oddly in some cases.)

(18) BOT

```
bind bot <flags> <command> <proc>
proc-name <from-bot> <command> <text>
```

triggered by a message coming from another bot in the botnet; works similar to a DCC binding; the first word is the command and the rest becomes the argument string; flags are ignored

(19) CHON (stackable)

```
bind chon <flags> <mask> <proc>
proc-name <handle> <idx>
```

when someone first enters the "party-line" area of the bot via dcc chat or telnet, this is triggered before they are connected to a chat channel (so yes, you can change the channel in a 'chon' proc); mask matches against handle; this is NOT triggered when someone returns from the file area, etc

(20) CHOF (stackable)

```
bind chof <flags> <mask> <proc>
proc-name <handle> <idx>
```

triggered when someone leaves the party line to disconnect from the bot; mask matches against the handle; note that the connection may have already been dropped by the user, so don't send output to that idx

(21) SENT (stackable)

```
bind sent <flags> <mask> <proc>
proc-name <handle> <nick> <path/to/file>
```

after a user has successfully downloaded a file from the bot, this binding is triggered; mask is matched against the handle of the user that initiated the transfer; nick is the actual recipient (on IRC) of the file; the path is relative to the dcc directory (unless the file transfer was started by a script call to 'dcsend', in which case the path is the exact path given in the call to 'dcsend')

(This is only available when the 'transfer' module is loaded.)

(22) RCVD (stackable)

```
bind rcvd <flags> <mask> <proc>
proc-name <handle> <nick> <path/to/file>
```

triggered after a user uploads a file successfully; mask is matched against the user's handle; nick is the nickname on IRC that the file transfer originated from; the path is where the file ended up, relative to the dcc directory (usually this is your incoming dir)

(This is only available when the 'transfer' module is loaded.)

(23) CHAT (stackable)

```
bind chat <flags> <mask> <proc>
proc-name <handle> <channel#> <text>
```

when someone says something on the botnet, it invokes this binding; flags are ignored; handle could be a user on this bot ("DronePup") or on another bot ("Eden@Wilde") and therefore you can't rely on a local user record; the mask is checked against the text

(24) LINK (stackable)

```
bind link <flags> <mask> <proc>
proc-name <botname> <via>
```

triggered when a bot links into the botnet; botname is the botnetnick of the bot that just linked in; via is the bot it linked through; the mask is checked against the bot that linked; flags are ignored

(25) DISC (stackable)

```
bind disc <flags> <mask> <proc>
```

proc-name <botname>

triggered when a bot disconnects from the botnet for whatever reason; just like the link bind, flags are ignored; mask is checked against the botnetnick of the bot that left

(26) SPLT (stackable)

bind splt <flags> <mask> <proc>

procname <nick> <user@host> <handle> <channel>

triggered when someone gets netsplit on the channel; be aware that this may be a false alarm (it's easy to fake a netsplit signoff message); <mask> may contain wildcards, and is matched against the channel and nick!user@host just like join; anyone who is SPLT will trigger a REJN or SIGN within the next 15 minutes

(27) REJN (stackable)

bind rejn <flags> <nick!user@host> <proc>

procname <nick> <user@host> <handle> <channel>

someone who was split has rejoined; <mask> can contain wildcards, and contains channel and nick!user@host just like join

(28) FILT (stackable)

bind filt <flags> <mask> <proc>

procname <idx> <text>

DCC party line and file system users have their text sent through filt before being processed; if the proc a blank string, the text is considered parsed; otherwise the bot will use the text returned from the proc and continue parsing that

(29) NEED (stackable)

bind need <flags> <mask> <proc>

procname <channel> <type>

this bind is triggered on certain events, like when the bot needs operator status or the key for a channel; the types are: op, unban, invite, limit, and key; the <mask> in the bind is matched against "#channel type" and can contain wildcards; flags are ignored example:

- bind need - "% op" needop < handles only need op
- bind need - "*" needall < handles all needs

(30) FLUD (stackable)

bind flud <flags> <type> <proc>

procname <nick> <user@host> <handle> <type> <channel>

any floods detected through the flood control settings (like 'flood-ctcp') are sent here before processing; if the proc returns 1, no further action is taken on the flood; if the proc returns 0, the bot will do its normal "punishment" for the flood; the flood type is "pub", "msg", "join", or "ctcp" (and can be masked to "*" for the bind); flags are ignored

(31) NOTE

bind note <flags> <handle> <proc>
procname <from> <to> <text>

incoming notes (either from the party line, someone on IRC, or someone on another bot on the botnet) are checked against these binds before being process; if a bind exists, the bot will not deliver the note; the handle must be an exact match (no wildcards), but it is not case sensitive; flags are ignored

(32) ACT (stackable)

bind act <flags> <mask> <proc>
proc-name <handle> <channel#> <action>

when someone does an action on the botnet, it invokes this binding; flags are ignored; the mask is checked against the text of the action (this is very similar to the CHAT binding)

(33) WALL (stackable)

bind wall <flags> <mask> <proc>
proc-name <handle> <msg>

when the bot receives a wallops, it invokes this binding; flags are ignored; the mask is checked against the text of the wallops msg

(34) BCST (stackable)

bind best <flags> <mask> <proc>
proc-name <botname> <channel#> <text>

when a bot says something on the botnet, it invokes this binding; flags are ignored; the mask is checked against the text

(35) CHJN (stackable)

bind chjn <flags> <mask> <proc>
proc-name <botname> <handle> <channel#> <flag> <idx> <from>

when someone joins a botnet channel, it invokes this binding; flags are ignored; the mask is checked against the text

(36) CHPT (stackable)

bind chpt <flags> <mask> <proc>
proc-name <botname> <handle> <idx> <channel#>

when someone parts a botnet channel, it invokes this binding; flags are ignored; the mask is checked against the channel

(37) TIME (stackable)

bind time - <mask> <proc>
proc-name <minute> <hour> <day> <month> <year>

allows you to schedule procedure calls at certain times, mask matches 5 space separated integers of the form: "minute hour day month year" minute, hour, day, month have a zero padding so they are exactly two characters long, year is extended to four characters in the same way if needed ;)

(38) AWAY (stackable)

```
bind away - <mask> <proc>
proc-name <botname> <idx> <text>
```

triggers when a user goes away or comes back on the botnet, text is the reason than has been specified (text == "" when returning)

(39) LOAD (stackable)

```
bind load - <mask> <proc>
proc-name <module>
```

triggers when a module is loaded.

(40) UNLD (stackable)

```
bind unld - <mask> <proc>
proc-name <module>
```

triggers when a module is unloaded.

(41) NKCH (stackable)

```
bind nkch - <mask> <proc>
proc-name <oldhandle> <newhandle>
```

triggered whenever a local users handle is changed (in the userfile)

(42) EVNT (stackable)

```
bind evnt - <type> <proc>
proc-name <type>
```

triggered whenever one of these events happens. valid events are:

- sighup (called on a kill -HUP <pid>)
- sigterm (called on a kill -TERM <pid>)
- sigill (called on a kill -ILL <pid>)
- sigquit (called on a kill -QUIT <pid>)
- save (called when the userfile is saved)
- rehash (called just after a rehash)
- prerehash (called just before a rehash)

- prerestart (called just before a restart)
- logfile (called when the logs are switched daily)
- loaded (called when the bot is done loading itself)
- connect-server (called just before we connect to an irc server)
- init-server (called when we actually get on our irc server)
- disconnect-server (called when we disconnect from our irc server)

(43) LOST (stackable)

bind lost <flags> <mask> <proc>

proc-name <handle> <nick> <path> <bytes-transferred> <length-of-file>

Triggered when a DCC SEND transfer gets lost, such as when the connection is terminated before all data was successfully sent/received. This is typically caused by a user abort.

(This is only available when the 'transfer' module is loaded.)

(44) TOUT (stackable)

bind tout <flags> <mask> <proc>

proc-name <handle> <nick> <path> <bytes-transferred> <length-of-file>

Triggered when a DCC SEND transfer times out. This may either happen because the dcc connection was not accepted or because the data transfer stalled for some reason.

(This is only available when the 'transfer' module is loaded.)

(A) RETURN VALUES

Several bindings pay attention to the value you return from the proc (using 'return \$value'). Usually they expect a 0 or 1, and returning an empty return any value is interpreted as a 0. Be aware if you omit the return statement, the result of the last Tcl command executed will be returned by the proc. This will not likely produce the results you intended (this is a "feature" of Tcl).

Here's a list of the bindings that use the return value from procs they trigger:

- **MSG Return 1** to make the command get logged like so: (nick!user@host) !handle! command
- **DCC Return 1** to make the command get logged like so: #handle# command
- **FIL Return 1** to make the command get logged like so: #handle# files: command
- **PUB Return 1** to make the command get logged like so: <<nick>> !handle! command
- **CTCP Return 1** to ask the bot not to process the CTCP command on its own. Otherwise it would send its own response to the CTCP (possibly an error message if it doesn't know how to deal with it).
- **FILT Return 1** to indicate the text has been processed, and the bot should just ignore it. Otherwise it will treat the text like any other.

- **FLUD Return 1** to ask the bot not to take action on the flood. Otherwise it will do its normal punishment.
- **RAW Return 1** to ask the bot not to process the server text. This can affect the bot's performance (by causing it to miss things that it would normally act on) -- you have been warned.
- **CHON Return 1** to ask the bot not to process the partyline join event
- **CHOF Return 1** to ask the bot not to process the partyline part event
- **WALL Return 1** to make the command get logged liked so: !nick! msg

(B) CONTROL PROCEDURES

Using the 'control' command you can put a DCC connection (or outgoing TCP connection) in control of a script. All text that comes in on the connection is sent to the proc you specify. All outgoing text should be sent with 'putdcc'.

The control procedure is called with these parameters:

procname <idx> <input-text>

This allows you to use the same proc for several connections. The idx will stay the same until the connection is dropped -- after that, it will probably get reused for a later connection.

To indicate that the connection has closed, your control procedure will be called with blank text (the input-text will be ""). This is the only time it will ever be called with "" as the text, and it is the last time your proc will be called for that connection. Don't call killdcc on the idx when text is blank, it will always fail with "invalid idx".

If you want to hand control of your connection back to Eggdrop, your proc should return 1. Otherwise, return 0 to retain control.

(C) TCP CONNECTIONS

Eggdrop allows you to make two types of TCP ("telnet") connections: outgoing and incoming. For an outgoing connection, you specify the remote host and port to connect to. For an incoming connection, you specify a port to listen at.

All of the connections are **event driven**. This means that the bot will trigger your procs when something happens on the connection, and your proc is expected to return as soon as possible. Waiting in a proc for more input is a no-no.

To initiate an outgoing connection, use:

set idx [connect hostname.goes.here 3333] (as an example).

\$idx now contains a new DCC entry for the outgoing connection. All connections use non-blocking (commonly called "asynchronous", which is a misnomer) I/O. Without going into a big song and dance about asynchronous I/O, what this means to you is:

- assume the connection succeeded immediately
- if the connection failed, an EOF will arrive for that idx

The only time a 'connect' call will return an error is if you gave a hostname and it couldn't find the IP for that hostname (this is considered a "DNS error"). Otherwise it will appear to have succeeded, and if the connection failed, you will immediately get an EOF.

Right after doing a 'connect' call, you should set up a 'control' for the new idx (see the section above). From then on, the connection will act just like a normal DCC connection that has been put under the control of a script. If you ever return "1" from the control proc (indicating that you want control to return to Eggdrop), the bot will just close the connection and dispose of it. Other commands that work on normal DCC connections, like 'killdcc' and 'putdcc', will work on this idx too. 'killdcc' will fail with "invalid idx" if you attempt to use it on a closed socket, such as when text is blank.

To create a listening port, use:

listen 6667 script grabproc

which will create a new listening port at 6667, and assign it to the script 'grabproc'.

When a new connection arrives, Eggdrop will connect it up and create a new idx for the connection. That idx is sent to 'grabproc'. The proc will generally want to immediately put this idx under control:

```
proc grabproc {newidx} {  
    control $newidx my_control  
}
```

Once your grabproc has been called, the idx behaves exactly like an outgoing connection would. The best way to learn how to use these commands is to find a script that uses them and follow it carefully. Hopefully this has given you a good start though.

(D) MATCH CHARACTERS

Many of the bindings allow match characters in the arguments. Here are the four special characters:

- ? matches any single character
- * matches 0 or more characters of any type
- % matches 0 or more non-space characters (can be used to match a single word)
- ~ matches 1 or more space characters (can be used for whitespace between words)

(E) TEXTFILE SUBSTITUTION

These %-variables can be inserted into help files, the banner, the MOTD, and other text files.

There are four variables that can be used to format text:

- %b display bold
- %v display inverse
- %_ display underline
- %f display flashing via telnet; bold underline via irc

These variables will be interpreted by Eggdrop and replaced by their respective values:

- %B bot's nickname (i.e., "LamestBot")

- %V current Eggdrop version (i.e., "eggdrop v1.6.8")
- %E long form of %V (i.e., "Eggdrop v1.6.8 (C) 1997 Robey Pointer (C) 2002 Eggheads")
- %C channels the bot is on (i.e., "#lamest, #botnetcentral")
- %A whatever is set in the config file by 'set admin'
- %n whatever is set in the config file by 'set network'
- %T the current time (i.e., "15:00")
- %N the current user's nickname (i.e., "Robey")
- %U the current operating system the bot is running on
- %% a percent sign ("%")

You can also encode messages which can only be read by people with certain flags:

```
%{+m}
Only masters would see this.
%{-}

%{+A}
Only people with the user flag A see this.
%{-}

%{+b}
This is only displayed to users doing a remote '.motd' from another bot.
%{-}

%{+|m}
Only channel masters would see this.
%{-}
```

Other variables:

- %{cols=N} start splitting output into N columns
- %{cols=N/W} same as above, but use a screen width of W
- %{end} end columnated or restricted (i.e. %{+m}) block
- %{center} center the following text (70 columns)

NOTES MODULE COMMANDS

these commands are provided by notes.so to allow you to store notes for users to read later

notes <user> [numberlist]

gets info on notes stored for a user

returns: (if no numbers specified) number of notes for user, -1 if no such user, -2 if notefile failure (if a note numberlist specified) a list of notes, -1 if no such user, -2 if notefile failure, 0 if no such note. Each note of the list is also a list: first element from, 2nd element timestamp, 3rd element the note itself. ('notes mynick "2-4;8;16-")

erasenotes <user> <numberlist>

erases some or all stored notes for a user

returns: -1 if no such user, -2 if notefile failure, 0 if no such note, or number of erased notes. 'erasenote mynick "-" erase all notes for mynick.

listnotes <user> <numberlist>

lists existing notes according to the numberlist (ex: "2-4;8;16-")

returns: -1 if no such user, -2 if notefile failure, 0 if no such note, list of existing notes.

storenote <from> <to> <msg> <idx>

stores a note for later reading, notify idx of any results (use idx == -1 for no notify).

returns: 0 on success non-0 on failure

ASSOC MODULE COMMANDS

assoc <chan> [name]

sets the name associated with a botnet channel, if you specify one

returns: current name for that channel, if any

killassoc <chan>

removes the name associated with a botnet channel, if any exists, use 'killassoc &' to kill all assocs.

returns: nothing

COMPRESS MODULE COMMANDS

`compressfile [-level <level>] <src-file> [target-file]`

`uncompressfile <src-file> [target-file]`

Compress or uncompress files. The level option specifies the compression mode to use when compressing. Available modes are from 0 (minimum CPU usage, minimum compression) all the way up to 9 (maximum CPU usage, maximum compression). If you don't specify the target-file, the src-file will be overwritten.

`iscompressed <filename>`

Determines whether <filename> is gzip compressed.

returns: "1" if it is, "0" if it isn't and "2" if some kind of error prevented the checks from succeeding.

FILE SYSTEM MODULE COMMANDS

setpwd <idx> <dir>

changes the directory of a file system user, in exactly the same way as a 'cd' command would. The directory can be specified relative or absolute.

returns: nothing

getpwd <idx>

returns: the current directory of a file system user

getfiles <dir>

returns: list of files in the directory given; the directory is relative to dcc-path

getdirs <dir>

returns: list of subdirectories in the directory given; the directory is relative to dcc-path

dccsend <filename> <ircnick>

attempts to start a dcc file transfer to the given nick; the filename must be specified either by full pathname or in relation to the bot's startup directory

returns: "0" on success, "1" if the dcc table is full (too many connections), "2" if it can't open a socket for the transfer, "3" if the file doesn't exist, and "4" if the file was queued for later transfer (which means that person has too many file transfers going right now)

filesend <idx> <filename> [ircnick]

like dccsend, except it operates for a current filesystem user, and the filename is assumed to be a relative path from that user's current directory

returns: "0" on failure; "1" on success (either an immediate send or a queued send)

fileresend <idx> <filename> [ircnick]

like filesend, only that it starts a DCC RESEND instead of a DCC SEND, which allows people to resume aborted file transfers if their client supports that protocol. ircII/BitchX/etc. support it, mIRC does not.

returns: "0" on failure; "1" on success (either an immediate send or a queued send)

setdesc <dir> <file> <desc>

sets the description for a file in a file system directory; the directory is relative to the dcc-path

returns: nothing

getdesc <dir> <file>

returns: the description for a file in the file system, if one exists

setowner <dir> <file> <handle>

changes the owner for a file in the file system; the directory is relative to the dcc-path

returns: nothing

getowner <dir> <file>

returns: the owner of a file in the file system

setlink <dir> <file> <link>

creates or changes a linked file (a file that actually exists on another bot); the directory is relative to dcc-path

returns: nothing

getlink <dir> <file>

returns: the link for a linked file, if it exists

getfileq <handle>

returns: list of files queued by someone; each item in the list will be a sublist with two elements: nickname the file is being sent to, and the filename

getfilesendtime <idx>

lets you figure out the unix time when a file transfer started.

returns: "-1" when no matching transfer with the specified idx is found, "-2" if the idx matches an entry which is not a file transfer and the unix start-time of the file-transfer in all other cases.

mkdir <directory> [<required-flags> [channel]]

creates a directory in the files system, only users with the required flags may access

returns: 0 on success; 1 on can't create directory; 2 on directory exists but is not a directory

rmdir <directory>

removes a directory from the file system.

returns: 0 on success, 1 on failure

mv <file> <destination>

moves the file from it's source to the given destination, file can also be a mask, e.g. /incoming/* provided the destination is a directory

returns: number of files copied on success or negative numbers to indicate errors: -1 = invalid source file; -2 = invalid destination; -3 = you're trying to copy onto itself (duh!); -4 = no matches found

cp <file> <destination>

exactly the same as mv except it leaves the original file there as well

returns: same as mv

getflags <dir>

returns: the flags required to access this directory

setflags <dir> [<flags> [channel]]

sets the flags required to access the directory

returns: 0 on success, -1 on failure