Appendix Bash Script Samples

As promised here is the unadulterated add users (and groups) script, that includes trapping of missing file name input :

```
#!/bin/bash


expiredate=2009-02-18


if [[ -z "${1}" ]] ; then
echo ""
echo "Please give exactly one file name."
echo "The file will have one user per line."
echo "Each line will have:"
echo " username"
echo " group"
echo " personal real name"
echo ""
exit 1
fi


cat "${1}" | while read username groupname realname
do
```

```
# Check if the user already exists.
# If so, then report this and skip this user.
result=$( egrep "^${username}:" < /etc/passwd )
if [[ -n "${result}" ]] ; then
echo "User '${username}' already exists"
continue
fi

# Check if the group already exists.
# If not, then add the group.
result=$( egrep "^${groupname}:" < /etc/group )
if [[ -z "${result}" ]] ; then
groupadd "${groupname}"
fi

# Add the user.
useradd -c "${realname}" \
-d "/home/${username}" \
-e "${expiredate}" \
-f 365 \
-g "${groupname}" \
-m \
-s /bin/bash \
"${username}" \
```

```
done
```

I would suggest removing the last line continuation backslash follow the line "${username}", since I had problems getting the code to work. However, I have been told the code works as is and it just might be my strong propensity to break any potential weakness that may be at play here.

In the text of Chapter 10 it was mentioned that a script had been written to auto generate supposedly random passwords. Since it was further noted that despite that the *en masse* addition by the add user script was able to set up nearly every user needed item, the initial password was missing. Hence, I have abstracted the suggested methods here:

```
#!/bin/bash

n="${1}"

[[ -n "${n}" ]] || n=12

if [[ $n -lt 8 ]]; then

echo "A password of length $n would be too weak"

exit 1

fi

c=""

while [[ -z "${c}" ]]; do

p=$( dd if=/dev/urandom bs=512 count=1 2>/dev/null \

| mimencode \

| tr -d '\012/+' \

| cut -c 1-$n )

c=$( echo "${p}" | tr -d 'a-zA-Z' )

done

echo "${p}"
```

If this is clear to you as it stands, you indeed are a better coder than I. Whomever you are, you can skip forward while the rest of us look a bit more closely at the inherent flaws in this code. One of the biggest problems with this code is that one character variable names are nearly meaningless. Those that write in such a fashion are not interested in imparting information to their likely successors. Note that last phrase made a very important point, leave your system usable, stable and understandable. While I will list the "documentation" written to explain this code, it is quite unlikely one could expect similar effort in the routine support of a real system, hence, it would be best to leave comments telling of the code's purpose. This should be split into two parts: preferably an overview right in the headed and explicit explanations in close proximity to difficult to understand processes. However, do not waste time just running through the command listed, since those may be looked up if the needed. However, where you employ a more exotic variant be explicit about its features. It is the results you are seeking with command sets that should be noted and why you

are pursuing it in the manner you have chosen. Do that and you will know yourself to be a systems administrator of the highest rank (no sarcasm).

Now here is the explanation of the above code in detail you are unlikely to see in the real world:

“The script begins with the usual starting comment that tells the system to run the bash interpreter. Next, we assign the first argument string to the variable n which will be the number of characters to generate. We put this in quotes because it may be a null string when used with no arguments given. That string is then tested to determine if it really is null. The -n actually means “non-zero length”, so the test is actually true if a string is given. The two vertical bars mean “logical or” and will execute the assignment that follows if the test fails. This forces a default length of 12 for our password. The next four lines check to see if the given length is too small.

We use a loop to generate the actual password string from some system commands because it is possible for the result to be weak. The variable named c will contain just the numeric digits part of the password. We initialize it to be empty to make the loop run through the first time. The while statement that controls the loop tests this variable and runs the loop body if it is empty. The -z means “zero length”.

The first statement in the loop body uses four system commands in a pipeline to generate one trial password. Because the pipeline is so long, we break it across four lines by placing a backslash at the end of the first three of these lines. The backslash at the end of the line continues it on to the next line. All four lines are placed inside **$()** to capture the output as a string that is then assigned to the variable p.

The script begins with the usual starting comment that tells the system to run the bash interpreter. Next, we assign the first argument string to the variable n which will be the number of characters to generate. We put this in quotes because it may be a null string when used with no arguments given. That string is then tested to determine if it really is null. The -n actually means “non-zero length”, so the test is actually true if a string is given. The two vertical bars mean “logical or” and will execute the assignment that follows if the test fails. This forces a default length of 12 for our password. The next four lines check to see if the given length is too small.

We use a loop to generate the actual password string from some system commands because it is possible for the result to be weak. The variable named c will contain just the numeric digits part of the password. We initialize it to be empty to make the loop run through the first time. The while statement that controls the loop tests this variable and runs the loop body if it is empty. The -z means “zero length”.

The first statement in the loop body uses four system commands in a pipeline to generate one trial password. Because the pipeline is so long, we break it across four lines by placing a backslash at the end of the first three of these lines. The backslash at the end of the line continues it on to the next line. All four lines are placed inside **$()** to capture the output as a string that is then assigned to the variable p.”

One final example: the remove file or directory script that was enhanced several times in Chapter 10:

```
admin@server1:~$ cat delete

#!/bin/bash

if rm $1

then

echo file $1 deleted

else

if rmdir $1
```

```
then

echo directory $1 deleted

fi

fi
```

just shows how can be done to a simple, short script. Nonetheless, less one that attempts to accomplish too much.

Here are more samples that you may find useful, although many of these are both long and complex. Enjoy:

Below are three example scripts developed for actual system administration use. You can type them in for your own use. Or you can study them for ideas to write your own scripts.

[Bill, most of what follows is Phil's words with no changes, I just lacked the energy to work through his code.]

Easy authoritative DNS lookup

This script is used to do DNS lookups to examine authoritative data directly, bypassing any caching of the local DNS caching server. One feature of this script is how it uses its own name to specify what DNS record type to look up. If this script is named ***a*** then it will look up DNS *a records*. If this script is named ***soa*** then it will look up DNS *soa records*. The name ***ptr*** is a special case which takes an IPv4 address and converts it to the proper *in-addr.arpa* form to do the actual lookup. It is suggested to make a copy of this script named for each of the common DNS record types you may need to look up: **a**, **aaaa**, **any**, **cname**, **mx**, **ns**, **ptr**, **soa**, and **txt**. You can also use hard links or symlinks to create the aliases.

The syntax for this script is simple. Type the command name that matches the record type to look up, follow by one or more names to be queried in the domain name system.

```
#!/bin/bash

#------------------------------------------------------------------

# Copyright © 2006 - Philip Howard - All rights reserved

#

# This program is free software; you can redistribute it and/or

# modify it under the terms of the GNU General Public License

# as published by the Free Software Foundation; either version 2

# of the License, or (at your option) any later version.

#

# This program is distributed in the hope that it will be useful,

# but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

# GNU General Public License for more details.

#

# You should have received a copy of the GNU General Public

# License along with this program; if not, write to the

# Free Software, Foundation, Inc., 59 Temple Place - Suite 330,

# Boston, MA 02111-1307, USA.

#-----------------------------------------------------------------

# script a, aaaa, cname, mx, ns, ptr, soa, txt

#

# purpose Perform direct DNS lookups for authoritative DNS

# data. This lookup bypasses the local DNS cache

# server.

#

# syntax a [ names ... ]

# aaaa [ names ... ]

# any [ names ... ]

# cname [ names ... ]

# mx [ names ... ]

# ns [ names ... ]

# ptr [ names ... ]

# soa [ names ... ]

# txt [ names ... ]

#

# author Philip Howard
```

```
#-----------------------------------------------------------------

function inaddr {

awk -F. '{print $4 "." $3 "." $2 "." $1 ".in-addr.arpa.";}'

return

}

q=$( exec basename "${0}" )

case "${q}" in

( dns* | dig* )

q="${q:3}"

;;

( d* )

q="${q:1}"

;;

esac

for n in "$@" ; do

if [[ "${q}" == ptr ]] ; then

case "x${n}y" in

( x[0-9]*\.[0-9]*\.[0-9]*\.[0-9]*y )

n=$( echo "${n}" | inaddr )

;;

( * )

;;

esac

fi

dig +trace +noall +answer "${q}" "${n}" | egrep "^${n}"
```

```
done

exit
```

Sending files between shell sessions

You can use the following script to send a file, or a directory of files (including all subdirectories), from one shell session to another shell session. This script works by creating an rsync daemon in the foreground to send the specified file or directory. It will display a few different forms of rsync commands that could be used to receive that file or directory. This script does not need to exist on the receiving system so it can even be used to send a copy of itself. The rsync package must already be installed on both systems.

The sending system must have network access open for the port number it uses to accept incoming rsync connections. The port number is chosen at random in the range 12288 through 28671. You can override the random port selection by using the -p option followed by a port number. If your firewall rules only allow one or a few ports to be connected, you must use those port numbers with this script.

To transfer data, first run this script on the sending system. Once it outputs sample commands, select which command would be appropriate to use based on the IP address or hostname that can reach the sending system, and the target location where the file or directory is to be stored at on the receiving system. Copy the selected command line, and paste that command into the shell of the receiving system to execute the rsync command to receive the data. The daemon will continue to run when the transfer is complete, allowing you to transfer multiple times to different computers. Stop the daemon when thel transfer(s) are complete by pressing cntrl-C in the sending system shell window.

This script has no security. Anyone who can reach the address and port number it is listening on can retrieve the data. You should not use this script to transfer confidential or secret data. Be sure to terminate the daemon once the desired transfer(s) are completed.

The suggested name for this script is: **`rsend`**

```
#!/bin/bash

#-----------------------------------------------------------------

# Copyright © 2006 - Philip Howard - All rights reserved

#

# This program is free software; you can redistribute it and/or

# modify it under the terms of the GNU General Public License

# as published by the Free Software Foundation; either version 2

# of the License, or (at your option) any later version.

#

# This program is distributed in the hope that it will be useful,

# but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

# GNU General Public License for more details.

#

# You should have received a copy of the GNU General Public

# License along with this program; if not, write to the

# Free Software, Foundation, Inc., 59 Temple Place - Suite 330,

# Boston, MA 02111-1307, USA.

#-----------------------------------------------------------------

# script rsend

#

# purpose To start an rsync daemon in the shell foreground

# to send a specified directory or file when

# retrieved using one of the rsync command lines

# shown by pasting it in a shell session on another

# host.

#

# usage rsend [options] directory | file

#

# options -c include checksum in the rsync command lines

# -d change daemon to the specified directory

# -n include dryrun in the rsync command lines

# -p use the specified port number, else random

# -s include sparse in the rsync command lines

# -u user to run as, if started as root

# -v show extra information
```

```
#

# author Philip Howard

#------------------------------------------------------------------

umask 022

hostname=$( exec hostname -f )

whoami=$( exec whoami )

uid="${whoami}"



#------------------------------------------------------------------

# Set defaults.

#------------------------------------------------------------------

checksum=""

delete=""

delmsg=""

dryrun=""

padding="-------"

port=""

sparse=""

verbose=""



bar1="--------------------------"

bar1="#${bar1}${bar1}${bar1}"



bar2="##########################"

bar2="#${bar2}${bar2}${bar2}"
```

```
#-----------------------------------------------------------------

# Include paths for ifconfig.

#-----------------------------------------------------------------

export PATH="${PATH}:/usr/sbin:/sbin"



#-----------------------------------------------------------------

# Scan options.

#-----------------------------------------------------------------

while [[ $# -gt 0 && "x${1:0:1}" = "x-" ]]; do

case "x${1}" in

( x-c | x--checksum )

checksum="c"

;;

( x--delete )

delete=" --delete"

delmsg="/delete"

padding=""

;;

( x-d | x--directory )

shift

cd "${1}" || exit 1

;;

( x--directory=* )

cd "${1:12}" || exit 1
```

```
;;

( x-n | x--dry-run )

dryrun="n"

;;

( x-p | x--port )

shift

port="${1}"

;;

( x--port=* )

port="${1:7}"

;;

( x-s | x--sparse )

sparse="S"

;;

( x-u | x--user )

shift

uid="${1}"

;;

( x--user=* )

uid="${1:7}"

;;

( x-v | x--verbose )

verbose=1

;;

esac
```

```
shift

done

[[ -z "${abort}" ]] || exit 1



#---------------------------------------------------------------

# Get a random number for a port.

#---------------------------------------------------------------

if [[ -z "${port}" || "${port}" = 0 || "${port}" = . ]]; then

port=$( dd if=/dev/urandom ibs=2 obs=2 count=1 2>/dev/null \

| od -An -tu2 | tr -d ' ' )

port=$[ $port % 16384 ]

port=$[ $port + 12288 ]

fi



#---------------------------------------------------------------

# Make up names for temporary files to be used.

#---------------------------------------------------------------

conffile="/tmp/rsync-${whoami}-${port}-$$.conf"

lockfile="/tmp/rsync-${whoami}-${port}-$$.lock"



#---------------------------------------------------------------

# This function adds quotes to strings that need them.

# Add single quotes if it has one of these: space $ " `

# Add double quotes if it has one of these: '

# Note: not all combinations will work.
```

```
#-------------------------------------------------------------

function q {

local s

s=$( echo "${1}" | tr -d ' $"`' )

if [[ "${s}" != "${1}" ]]; then

echo "'${1}'"

return

fi

s=$( echo "${1}" | tr -d "'" )

if [[ "${s}" != "${1}" ]]; then

echo '"'"${1}"'"'

return

fi

echo "${1}"

return 0

}



#-------------------------------------------------------------

# Only one name can be handled.

#-------------------------------------------------------------

if [[ $# -gt 1 ]]; then

echo "Only one name (directory or file)" 1>&2

exit 1

elif [[ $# -eq 1 ]]; then

name="${1}"
```

```
else

name=$( exec pwd )

fi



#----------------------------------------------------------------

# Set up a temporary config file.

#----------------------------------------------------------------

function configout {

echo "lock file = ${lockfile}"

echo "log file = /dev/stderr"

echo "use chroot = false"

echo "max connections = 32"

echo "socket options = SO_KEEPALIVE"

echo "list = yes"

echo "[.]"

echo "path = ${1}"

echo "read only = yes"

echo "uid = ${uid}"

echo "comment = ${2}"

if [[ -n "${3}" ]]; then

echo "include = **/${3}"

echo "exclude = **"

fi

}
```

```
#-----------------------------------------------------------------

# Get directory and file.

#-----------------------------------------------------------------

if [[ ! -e "${name}" ]]; then

echo "does not exist:" $( q "${name}" ) 1>&2

exit 1

elif [[ -d "${name}" ]]; then

p=$( exec dirname "${name}" )

b=$( exec basename "${name}" )

d="${name}"

f=""

r=$( cd "${name}" && exec pwd )

announce="${d}"

rsyncopt="-a${checksum}${dryrun}H${sparse}vz${delete}"

configout "${d}/." "directory:${d}/" >"${conffile}"

elif [[ -f "${name}" ]]; then

p=$( exec dirname "${name}" )

b=$( exec basename "${name}" )

d="${p}"

f="${b}"

r=$( cd "${p}" && exec pwd )

r="${r}/${b}"

announce="${d}/${f}"

rsyncopt="-a${checksum}${dryrun}${sparse}vz"

configout "${d}/." "file:${d}/${f}" >"${conffile}"
```

```
elif [[ -L "${name}" ]]; then

p=$( exec dirname "${name}" )

b=$( exec basename "${name}" )

d="${p}"

f="${b}"

r=$( cd "${p}" && exec pwd )

r="${r}/${b}"

announce="${d}/${f}"

rsyncopt="-a${checksum}v"

configout "${d}/." "symlink:${d}/${f}" "${f}" >"${conffile}"

fi



#---------------------------------------------------------------

# Show config file if verbose is requested.

#---------------------------------------------------------------

if [[ -n "${verbose}" ]]; then

echo "${bar2}"

ls -ld "${conffile}"

echo "${bar2}"

cat "${conffile}"

fi



#---------------------------------------------------------------

# This function outputs example receive commands.

#---------------------------------------------------------------
```

```
function showrsync {

echo -n "rsync ${rsyncopt} "

if [[ -n "${oldfmt}" ]]; then

echo "--port=${port}" $( q "${1}::${2}" ) $( q "${3}" )

else

echo $( q "rsync://${1}:${port}/${2}" ) $( q "${3}" )

fi

return

}



#-----------------------------------------------------------------

# These functions show rsync commands for hostname and IP.

#-----------------------------------------------------------------

function getip {

case $( exec uname -s ) in

( SunOS )

netstat -i -n | awk '{print $4;}'

;;

( Linux )

ifconfig -a | awk '{if($1=="inet")print substr($2,6);}'

;;

( * )

netstat -i -n | awk '{print $4;}'

;;

esac
```

```
return

}



function ipaddr {

getip \

| egrep '^[0-9]*\.[0-9]*\.[0-9]*\.[0-9]*$' \

| egrep -v '^0\.|^127\.' \

| head -2 \

| while read ipv4 more ; do

showrsync "${ipv4}" "$@"

done

return

}



function showcmd {

ipaddr "${2}" "${3}"

showrsync "${1}" "${2}" "${3}"

return

}



#-----------------------------------------------------------------

# Announce the shell commands to receive this data.

#-----------------------------------------------------------------

echo "${bar2}"

echo "# sending ${announce}"
```

```
echo "# paste ONE of these commands in a remote shell to receive"



if [[ -d "${name}" ]]; then

echo "${bar1}"

showcmd "${hostname}" . .



echo "${bar1}"

showcmd "${hostname}" . "${b}"



if [[ "${d}" != "${b}" && "${d}" != "${r}" ]]; then

echo "${bar1}"

showcmd "${hostname}" . "${d}"

fi



echo "${bar1}"

showcmd "${hostname}" . "${r}"

else

echo "${bar1}"

showcmd "${hostname}" "./${f}" "${b}"



s=$( exec basename "${d}" )

s="${s}/${f}"

if [[ "${s}" != "${b}" ]]; then

echo "${bar1}"
```

```
showcmd "${hostname}" "./${f}" "${s}"

fi



if [[ "${name}" != "${b}" \

&& "${name}" != "${s}" \

&& "${name}" != "${r}" ]]; then

echo "${bar1}"

showcmd "${hostname}" "./${f}" "${name}"

fi



echo "${bar1}"

showcmd "${hostname}" "./${f}" "${r}"

fi



echo "${bar1}"

echo "# press ^C here when done"

echo "${bar2}"



#-----------------------------------------------------------------

# Start rsync in daemon mode.

#-----------------------------------------------------------------

s="DONE"

trap 's="SIGINT ... DONE"' INT

trap 's="SIGTERM ... DONE"' TERM
```

```
rsync --daemon --no-detach "--config=${conffile}" "--port=${port}"

rm -f "${conffile}" "${lockfile}"

echo "${s}"
```

Integrating `ssh` and `screen`

You should already be familiar with the ssh command to remotely connect a secure shell session on another computer. The screen command is a useful tool that allows holding a shell session in an active state with its screen contents intact, while disconnecting from the remote computer. The held shell session can then be reconnected later, even from a different computer. It is also possible to have 2 or more connections to the same shell session.

The following script allows making the the ssh connection and starting a named screen session in one command. The benefit is quicker connecting and disconnecting when working with multiple servers.

This script is used much like the ssh command. The syntax to specify the username and hostname of the remote session is expanded to also include a session name. You can create multiple sessions on the remote host under the same username with different session names. The session name is option. If it is not given, then this script runs the ssh command in the normal way, without running screen. The full syntax of this script, including the ssh options it supports, can be seen in the script comments.

The suggested name for this script is: **`ss`**

```
#!/usr/bin/env bash

#-----------------------------------------------------------------

# Copyright © 2006 - Philip Howard - All rights reserved

#

# This program is free software; you can redistribute it and/or

# modify it under the terms of the GNU General Public License

# as published by the Free Software Foundation; either version 2

# of the License, or (at your option) any later version.

#

# This program is distributed in the hope that it will be useful,

# but WITHOUT ANY WARRANTY; without even the implied warranty of

# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

# GNU General Public License for more details.

#
```

```
# You should have received a copy of the GNU General Public

# License along with this program; if not, write to the

# Free Software, Foundation, Inc., 59 Temple Place - Suite 330,

# Boston, MA 02111-1307, USA.

#-----------------------------------------------------------------

# command ss (secure screen)

#

# purpose Establish a screen based background shell session

# via secure shell communications.

#

# syntax ss [options] session/username@hostname

# ss [options] session@username@hostname

# ss [options] username@hostname/session

# ss [options] username@hostname session

#

# options -h hostname

# -h=hostname

# -i identity

# -i=identity

# -l loginuser

# -l=loginuser

# -m Multi-display mode

# -p portnum

# -p=portnum

# -s session
```

```
# -s=session

# -t Use tty allocation (default)

# -T Do NOT use tty allocation

# -4 Use IPv4 (default)

# -6 Use IPv6

# -46 | -64 Use either IPv6 or IPv4

#

# requirements The local system must have the "openssh" package

# installed. The remote system must have the

# "openssh" package installed and have the "sshd"

# daemon running. It must also have the "screen"

# program installed. Configuring a ".screenrc"

# file on each system is recommended.

#

# note The environment variable SESSION_NAME will be set

# in the session created under screen for potential

# use by other scripts.

#

# author Philip Howard

#------------------------------------------------------------------

whoami=$( exec whoami )

hostname=$( exec hostname )



h=""

i=( )
```

```
m=""

p=( )

s=''

t=( -t )

u="${whoami}"

v=( -4 )



#-----------------------------------------------------------------

# Parse options and arguments by recognition.

#-----------------------------------------------------------------

while [[ $# -gt 0 ]]; do

case "x${1}" in

( x*/*@* )

u=$( echo "x${1}" | cut -d @ -f 1 )

u="${u:1}"

s=$( echo "x${u}" | cut -d / -f 2 )

u=$( echo "x${u}" | cut -d / -f 1 )

u="${u:1}"

h=$( echo "x${1}" | cut -d @ -f 2 )

shift

break

;;

( x*@*/* )

u=$( echo "x${1}" | cut -d @ -f 1 )

u="${u:1}"
```

```
h=$( echo "x${1}" | cut -d @ -f 2 )

s=$( echo "x${h}" | cut -d / -f 2 )

h=$( echo "x${h}" | cut -d / -f 1 )

h="${h:1}"

shift

break

;;

( x*@*@* )

s=$( echo "x${1}" | cut -d @ -f 1 )

s="${s:1}"

u=$( echo "x${1}" | cut -d @ -f 2 )

h=$( echo "x${1}" | cut -d @ -f 3 )

shift

break

;;

( x*@* )

u=$( echo "x${1}" | cut -d @ -f 1 )

u="${u:1}"

h=$( echo "x${1}" | cut -d @ -f 2 )

shift

if [[ $# -gt 0 ]]; then

s="${1}"

shift

fi

break
```

```
;;

( x-h=* )

h="${1:3}"

;;

( x-h )

shift

h="${1}"

;;

( x-i=* )

i="${1:3}"

if [[ -z "${i}" ]]; then

i=( )

else

i=( -i "${1:3}" )

fi

;;

( x-i )

shift

i=( -i "${1}" )

;;

( x-l=* | x-u=* )

u="${1:3}"

;;

( x-l | x-u )

shift
```

```
u="${1}"

;;

( x-m | x--multi )

m=1

;;

( x-p=* )

p="${1:3}"

if [[ -z "${p}" ]]; then

p=( )

else

p=( -p "${1:3}" )

fi

;;

( x-p )

shift

p=( -p "${1}" )

;;

( x-s=* )

s="${1:3}"

;;

( x-s )

shift

s="${1}"

;;

( x-t )
```

```
t=( -t )

;;

( x-T )

t=( )

;;

( x-4 )

v=( -4 )

;;

( x-6 )

v=( -6 )

;;

( x-46 | x-64 )

v=()

;;

( x-* )

echo "Invalid option: '${1}'"

die=1

;;

( * )

echo "Invalid argument: '${1}'"

die=1

;;

esac

shift

done
```

```
#-----------------------------------------------------------------

# Make sure essential information is present.

#-----------------------------------------------------------------

if [[ -z "${u}" ]]; then

echo "User name is missing"

die=1

fi



if [[ -z "${h}" ]]; then

echo "Host name is missing"

die=1

fi



[[ -z "${die}" ]] || exit 1



#-----------------------------------------------------------------

# Run screen on the remote only if a session name is given.

#-----------------------------------------------------------------

c=( ssh "${v[@]}" "${i[@]}" "${p[@]}" "${t[@]}" "${u}@${h}" )

if [[ -n "${s}" ]]; then

o="-DR"

[[ -n "${m}" ]] && o="-x"

x="exec /usr/bin/env SESSION_NAME='${s}' screen ${o} '${s}'"
```

```
    c=( "${c[@]}" "${x}" )

    fi

    exec "${c[@]}"
```

1 Please limit yourself to being a programmer where you are likely to do less damage.

2