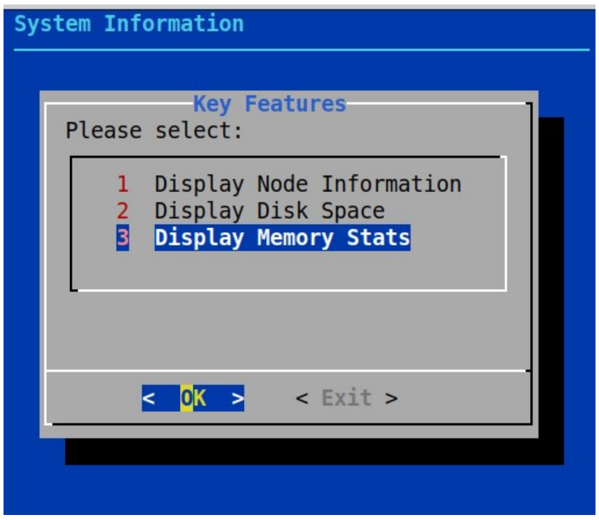
Text Interfaces with Whiptail and Dialog


System Information

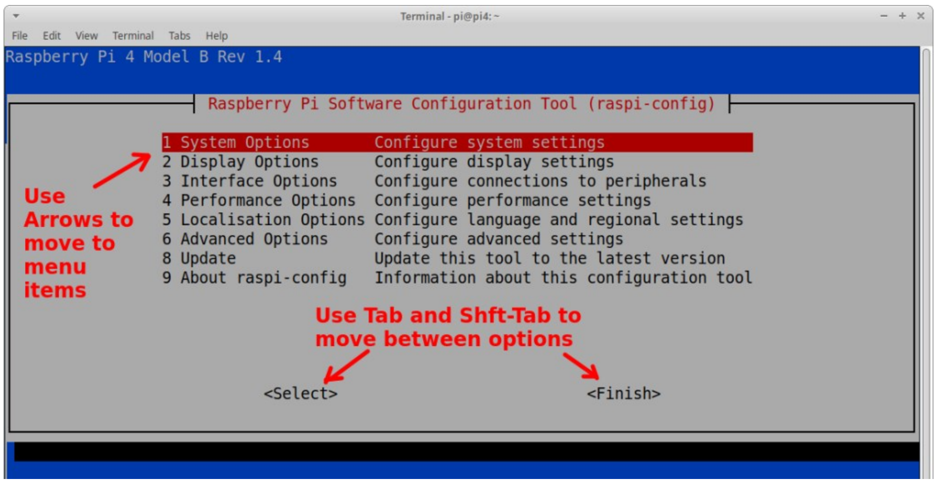
Key Features

Please select:

1 Display Node Information
2 Display Disk Space
3 Display Memory Stats

Text interfaces are extremely useful when an MS-Window client is trying to connect into a Linux or Raspberry Pi node.

There are some good options for creating Bash text interfaces, two of the most popular are Whiptail and Dialog. Raspberry Pi users will be familiar with Pi configuration tool, ***raspi-config***, which uses Whiptail to create all its menu and application screens.



```
Raspberry Pi 4 Model B Rev 1.4

Raspberry Pi Software Configuration Tool (raspi-config)

1 System Options          Configure system settings
2 Display Options         Configure display settings
3 Interface Options       Configure connections to peripherals
4 Performance Options     Configure performance settings
5 Localisation Options    Configure language and regional settings
6 Advanced Options        Configure advanced settings
8 Update                  Update this tool to the latest version
9 About raspi-config      Information about this configuration tool

<Select>                  <Finish>
```

Use Arrows to move to menu items
Use Tab and Shft-Tab to move between options

Whiptail comes pre-installed on Raspbian and on many Linux images. Whiptail is a lighter version of the more complete Dialog package.

This blog will show some examples using Whiptail and Dialog.

Getting Started

For my work I preferred using Dialog, I found that it had many more options and a slightly easier interface than Whiptail. However for projects where I needed to distribute my code to other users or nodes I would keep things simple and use Whiptail.

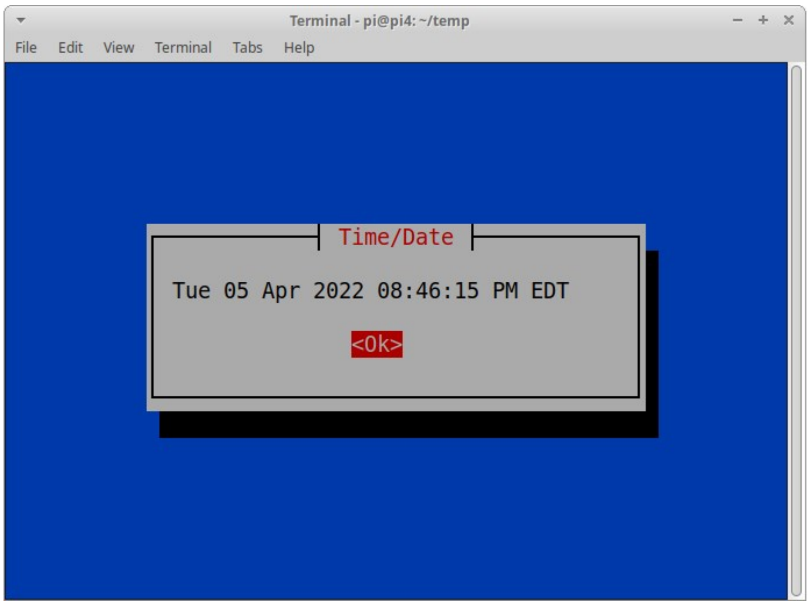
To install Whiptail and Dialog on a Raspbian/Debian/Ubuntu:

```
sudo apt install whiptail
sudo apt install dialog
```

A simple Whiptail test to show the date/time in a message box:

```
whiptail --title "Time/Date" --msgbox  "$(date)" 0 0
```

This statement will clear the terminal screen and show a default background with message box centered in the window. A height and width of : 0 0 will autosize the message box.



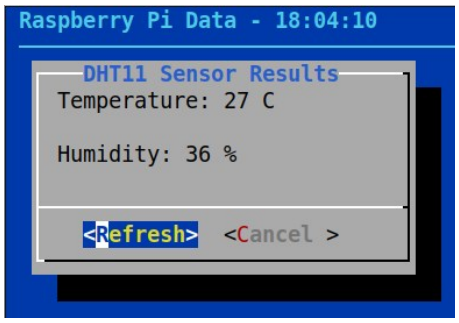
```
Time/Date
Tue 05 Apr 2022 08:46:15 PM EDT
<Ok>
```

Refreshing YESNO Dialog

A message box only has a single OK button. A ***yesno*** dialog has two button.

Whiptail only supports a basic yesno function. The Dialog utility supports changing the button labels and a timeout so the window can be automatically refreshed.

Below is an example that refreshed an yesno dialog every 5 seconds. The ***show_dlg*** function generates a random temperature and humidity valid, and then calls the dialog utility. The YES button, relabelled as "Refresh", will manually force a refresh of the data and redraw the window. The NO button, relabelled as "Cancel" will close the script and clear the screen.



```
Raspberry Pi Data - 18:04:10
DHT11 Sensor Results
Temperature: 27 C

Humidity: 36 %

<Refresh>  <Cancel >
```

```
#!/usr/bin/bash
#
# dyesno.sh - A freshing yes/no dialog with simulated sensor data

YES=0
TIMEOUT=255

show_dlg() {
   # simulate some data
   data1="Temperature: $(( ( RANDOM % 10 )  + 20 )) C";
   data2="Humidity: $(( ( RANDOM % 10 )  + 30 )) %";
   message="$data1\n\n$data2";
   dialog --begin 2 2 --backtitle "Raspberry Pi Data - $(date +'%T' )" \
       --yes-label "Refresh" --no-label "Cancel" --timeout 5 \
       --title "DHT11 Sensor Results" --yesno  "$message" 8 30;
}

response=0
# Cycle is the response is YES button or the dialog timed out
while [ "$response" == "$YES" ] || [ "$response" == "$TIMEOUT" ]; do
  show_dlg
  response=$? ;# Get the output from the yesno dialog
done
clear
```

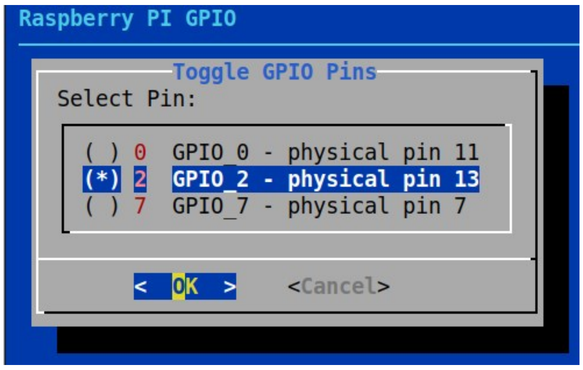
Radio Dialog to Toggle Pi GPIO Pins

A radio dialog allows a user to select one option from a list.

For this example I'm using 3 GPIO pins (0, 2, and 7) and the user can select one of these pins and toggle the output using the Raspberry Pi ***gpio*** utility.

The Dialog utility will output the item selected (0, 2 or 7), and the OK/Cancel button code (0=OK,1=Cancel).

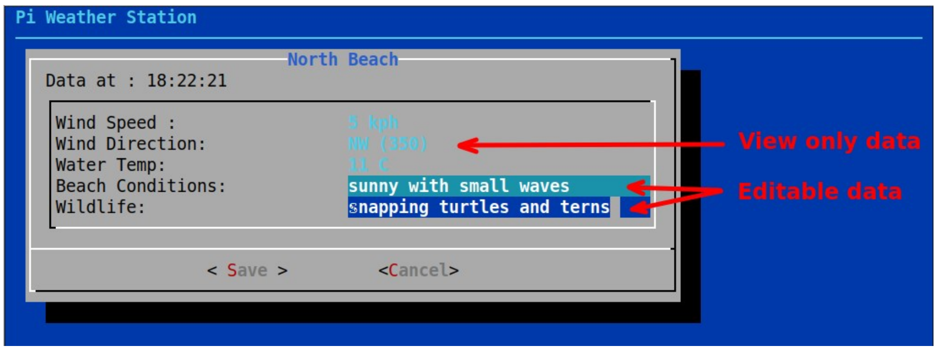
```
#!/usr/bin/bash
#
# dradio1.sh - toggle a GPIO output pin
#
thepin=$(dialog --begin 2 2 --title "Toggle GPIO Pins" \
     --backtitle "Raspberry PI GPIO" --stdout \
     --radiolist "Select Pin:" 10 40 3 \
      0 "GPIO_0 - physical pin 11" off \
      2 "GPIO_2 - physical pin 13" off \
      7 "GPIO_7 - physical pin 7 " off )
clear
# Toggle if OK entered and a pin is selected
if [ "$?" == "0" ] && (( ${#thepin} > 0 )) ; then
  echo "Toggling (wPi) pin: $thepin"
  gpio toggle $thepin
  echo "Pin $thepin is: $(gpio read $thepin)"
fi
```



```
Raspberry PI GPIO

Toggle GPIO Pins
Select Pin:

( ) 0  GPIO_0 - physical pin 11
(*) 2  GPIO_2 - physical pin 13
( ) 7  GPIO_7 - physical pin 7
```

Weather Station Form

A Form dialog can be used text with captions and allow user input to saved. Below is an example of a Weather station with view only and editable data.



```
Pi Weather Station

North Beach
Data at : 18:22:21

Wind Speed :          5 kph
Wind Direction:       NW (350)
Water Temp:           11 C
Beach Conditions:     sunny with small waves
Wildlife:             snapping turtles and terns
```

```
#!/usr/bin/bash
#
# dform1.sh - Form to show data, and allow data entry

# Weather Sensor inputs (connect real inputs here)
wspeed="5 kph"
wdir="NW (350)"
wtemp="11 C"

# Show a dialog with viewonly and data entry values, save to a file
dialog --begin 2 2 --ok-label "Save" --backtitle "Pi Weather Station" \
     --title "North Beach" --stdout  \
     --form  "Data at : $(date +'%T' )"  12 65 0 \
    "Wind Speed :"       1 1 "$wspeed"  1 30 0 0 \
    "Wind Direction:"    2 1 "$wdir"    2 30 0 0 \
    "Water Temp:"        3 1 "$wtemp"   3 30 0 0 \
    "Beach Conditions:" 4 1 ""          4 30 30 30 \
    "Wildlife:"          5 1 ""          5 30 30 30 > beach.txt
clear
```

The syntax for the form is:

```
--form text height width formheight [ label y x item y x flen ilen ]

where: y = line position
       x = position in line
    item = view only or editable data
    flen = field length , 0 = view only
    ilen = input length , 0 = view only
```

For this example the last 2 items (Beach Conditions and Wildlife) have a field and input length defined to 30 characters so data can be entered into these fields. If the OK button is selected the user entered data is saved to the file *beach.txt*.

Menu Example

Menuing applications are probably the most useful feature in the Whiptail and Dialog utilities.

Below is an System Information application, that has 3 options: node, disk space, and memory stats. Each of the menu items call a *display_dialog* function that presents the results of a Bash statement in a message box.

```
#!/bin/bash
#
# dmenu.sh - Bash Dialog Menu showing some system stats
#

# Display menu results in a msgbox
display_result() {
  dialog --title "$1" \
    --backtitle "System Information" \
    --no-collapse \
    --msgbox "$result" 0 0
}

while true; do
  selection=$(dialog --stdout \
    --backtitle "System Information" \
    --title "Key Features" \
    --clear \
    --cancel-label "Exit" \
    --menu "Please select:" 0 0 4 \
    "1" "Display Node Information" \
    "2" "Display Disk Space" \
    "3" "Display Memory Stats" \
     )
  exit_status=$?
  if [ $exit_status == 1 ] ; then
      clear
      exit
  fi
  case $selection in
    1 )
      result=$(echo "Hostname: $HOSTNAME"; uptime)
      display_result "System Information"
      ;;
    2 )
      result=$(df -h)
      display_result "Disk Space"
      ;;
    3 )
      result=$(vmstat --stats)
      display_result "Memory Stats"
      ;;
  esac
done
```

Changing Whiptail Default Colors

Whiptail uses the newt graphic library. A NEWT_COLORS variable can be created with custom colors. An example would be:

```
export NEWT_COLORS='
  window=,red
  border=white,red
  textbox=white,red
  button=black,white'
# to reset the color back to default use:
# unset NEWT_COLORS
```

A full definition of all the options and colors:

```
root                    root fg, bg
border                  border fg, bg
window                  window fg, bg
shadow                  shadow fg, bg
title                   title fg, bg
button                  button fg, bg
actbutton               active button fg, bg
checkbox                checkbox fg, bg
actcheckbox             active checkbox fg, bg
entry                   entry box fg, bg
label                   label fg, bg
listbox                 listbox fg, bg
actlistbox              active listbox fg, bg
textbox                 textbox fg, bg
acttextbox              active textbox fg, bg
helpline                help line
roottext                root text
emptyscale              scale full
fullscale               scale empty
disentry                disabled entry fg, bg
compactbutton           compact button fg, bg
actsellistbox           active & sel listbox
sellistbox              selected listbox

bg and fg can be:

color0  or black
color1  or red
color2  or green
color3  or brown
color4  or blue
color5  or magenta
color6  or cyan
color7  or lightgray
color8  or gray
color9  or brightred
color10 or brightgreen
color11 or yellow
color12 or brightblue
color13 or brightmagenta
color14 or brightcyan
color15 or white
```

Changing Default Dialog Colors

The custom Dialog colors are defined in the file: ~/***.dialogrc***

To create and edit this file:

```
dialog --create-rc  ~/.dialogrc
nano $HOME/.dialogrc
```

Within the ~/***.dialogrc*** file, an important option is:

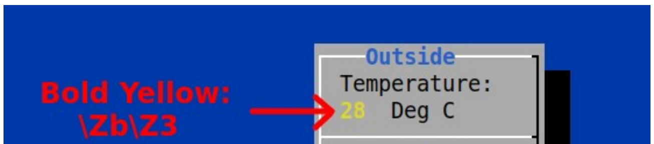
```
# Shadow dialog boxes? This also turns on color.
use_shadow = OFF
```

Dialog supports inline color (this isn’t supported in Whiptail) with the –***colors*** option. Inline colors are defined with by "/Zx":

```
ANSI colors:

/Z0 = black
/Z1 = red
/Z2 = green
/Z3 = yellow
/Z4 = blue
/Z5 = magenta
/Z6 = cyan
/Z7 = white

Other options:

/Zb = bold, /ZB = reset bold
/Zr = reverse, /ZR = reset reverse
/Zu = underline, /ZU = reset underline
/Zn = restore settings to normal
```

An example:

```
$ msg="Temperature: \Zb\Z3 28 \Zn Deg C"
$ dialog --title "Outside" --colors --msgbox  "$msg" 0 0 ; clear
```



Outside
Temperature:
28 Deg C

Final Comments

I’m a big fan of Zenity and YAD X-Window dialog tools, and I found that it wasn’t a big transition to use Whiptail and Dialog.

It’s important to note that the Dialog option –stdout is needed if you want to pass the Dialog output to a variable. Passing the output from Whiptail is a little trickier, use: *3>&2 2>&1 1>&3*

REDIRECTIONS :

How to retrieve or store the user input? If the user presses “Ok”, the input it provided is printed to standard error. Supposing we want to “store” the input in a variable, we would run:

```
$ username="$(whiptail --inputbox "What is your username?" 10 30 "root" 3>&1 1>&2 2>&3)"
```

What are all those redirections at the end of the command? As we know there are three **file descriptors** used by the shell: standard input (0), standard output (1) and standard error (2). Whiptail uses the **standard output** file descriptor to display widgets, and, as we already said, prints the input of the user to **standard error**. When we use the command substitution mechanism, a command is substituted by its output, which is what we usually store in a variable. In this case, however, what we want to store in the variable is the standard error produced by the command, therefore we need to **swap standard output and standard error** to achieve our goal. How can we do this?

When we perform a redirection, we **duplicate** a file descriptor into another. For, example, if we write:

```
1>&2
```

What we are saying is: “use a duplicate of the standard error file descriptor (2) for standard output”. To swap the standard output and standard error file descriptors, therefore we can write:

```
3>&1 1>&2 2>&3
```

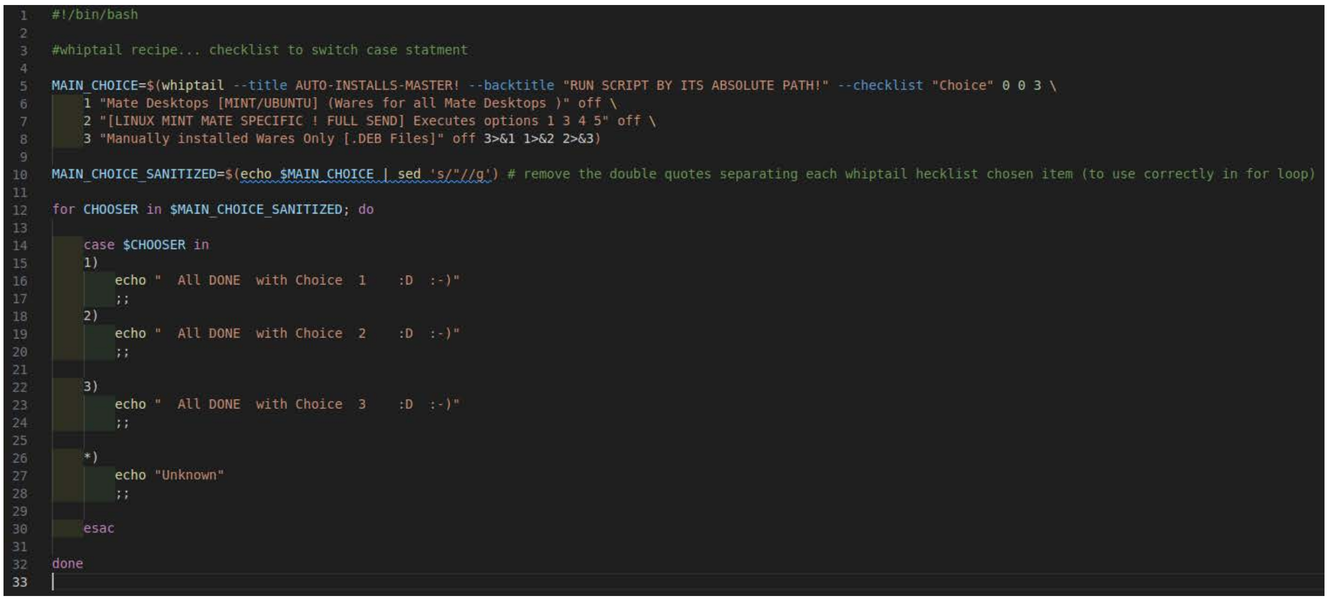
With those redirections we:

1. Duplicate file descriptor 1 (standard output) to file descriptor 3
2. Duplicate file descriptor 2 (standard error) to file descriptor 1 (standard output)
3. Duplicate file descriptor 3 (which, in the first step we made a duplicate of the original standard output) to file descriptor 2 (standard error)

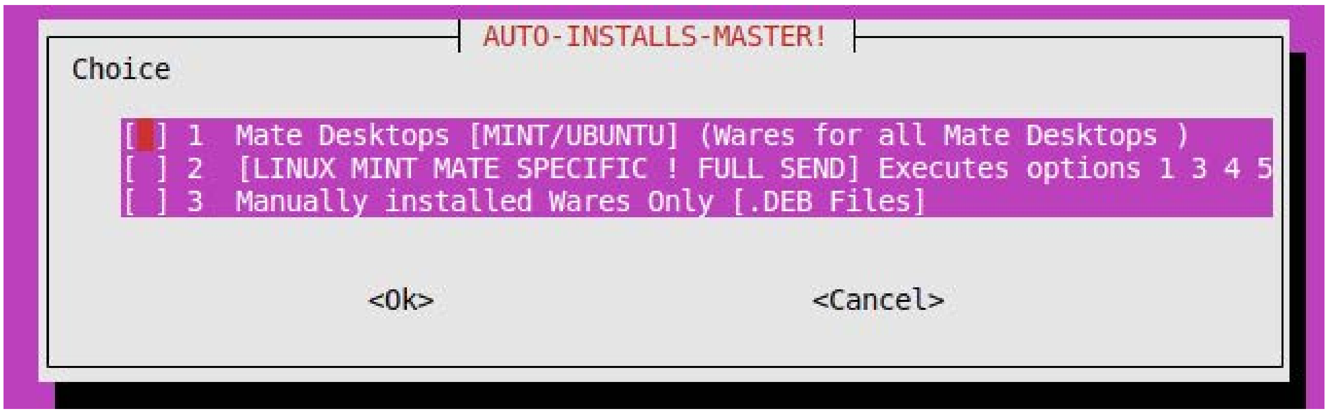
In other words, we just swapped standard error and standard output. We used an extra file descriptor (3) as a “temporary place” to store the original standard output, which otherwise would be lost. At this point the string entered as input by the user is printed to standard output and, since we used command substitution, is “stored” in the “username” variable:

A FEW WHIPTAIL RECIPES

create static whiptail checklist menu items and use a switch case statement (within a for loop) to execute commands based on selected choices:

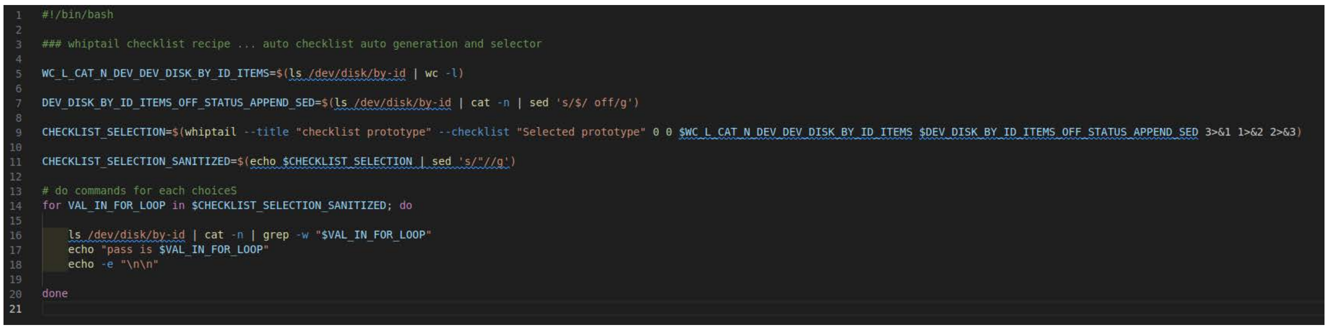


```
#!/bin/bash

#whiptail recipe... checklist to switch case statment

MAIN_CHOICE=$(whiptail --title AUTO-INSTALLS-MASTER! --backtitle "RUN SCRIPT BY ITS ABSOLUTE PATH!" --checklist "Choice" 0 0 3 \
    1 "Mate Desktops [MINT/UBUNTU] (Wares for all Mate Desktops )" off \
    2 "[LINUX MINT MATE SPECIFIC ! FULL SEND] Executes options 1 3 4 5" off \
    3 "Manually installed Wares Only [.DEB Files]" off 3>&1 1>&2 2>&3)

MAIN_CHOICE_SANITIZED=$(echo $MAIN_CHOICE | sed 's/"//g') # remove the double quotes separating each whiptail hecklist chosen item (to use correctly in for loop)

for CHOOSER in $MAIN_CHOICE_SANITIZED; do

    case $CHOOSER in
    1)
        echo "  All DONE  with Choice  1    :D  :-)"
        ;;
    2)
        echo "  All DONE  with Choice  2    :D  :-)"
        ;;

    3)
        echo "  All DONE  with Choice  3    :D  :-)"
        ;;

    *)
        echo "Unknown"
        ;;

    esac

done

```



```
              ┤ AUTO-INSTALLS-MASTER! ├
 Choice

    [ ] 1  Mate Desktops [MINT/UBUNTU] (Wares for all Mate Desktops )
    [ ] 2  [LINUX MINT MATE SPECIFIC ! FULL SEND] Executes options 1 3 4 5
    [ ] 3  Manually installed Wares Only [.DEB Files]


              <Ok>                          <Cancel>
```

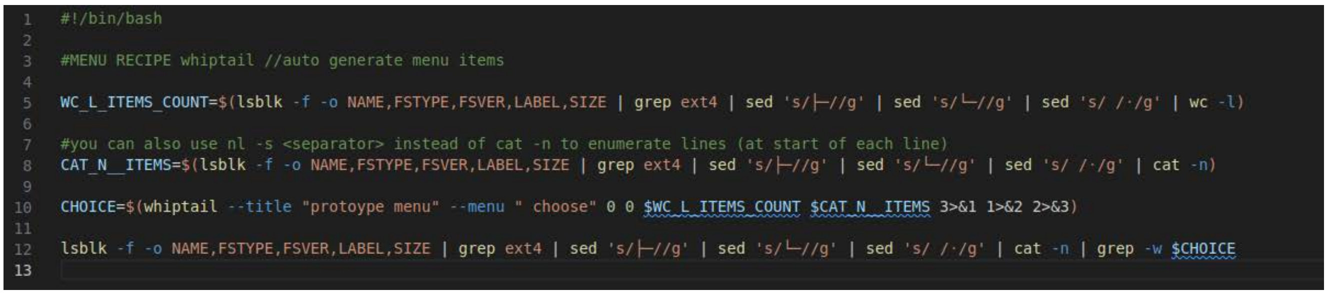
auto generate dynamic whiptail checklist menu items and use a for loop to execute commands based on selected choices:



```
#!/bin/bash

### whiptail checklist recipe ... auto checklist auto generation and selector

WC_L_CAT_N_DEV_DEV_DISK_BY_ID_ITEMS=$(ls /dev/disk/by-id | wc -l)

DEV_DISK_BY_ID_ITEMS_OFF_STATUS_APPEND_SED=$(ls /dev/disk/by-id | cat -n | sed 's/$/ off/g')

CHECKLIST_SELECTION=$(whiptail --title "checklist prototype" --checklist "Selected prototype" 0 0 $WC_L_CAT_N_DEV_DEV_DISK_BY_ID_ITEMS $DEV_DISK_BY_ID_ITEMS_OFF_STATUS_APPEND_SED 3>&1 1>&2 2>&3)

CHECKLIST_SELECTION_SANITIZED=$(echo $CHECKLIST_SELECTION | sed 's/"//g')

# do commands for each choiceS
for VAL_IN_FOR_LOOP in $CHECKLIST_SELECTION_SANITIZED; do

    ls /dev/disk/by-id | cat -n | grep -w "$VAL_IN_FOR_LOOP"
    echo "pass is $VAL_IN_FOR_LOOP"
    echo -e "\n\n"

done

```

auto generate dynamic whiptail Menu items and isolate the selected choices with grep:

may need to use :: sed "s/$SELECTION//1" :: to eliminate first occurence of pattern match or :: cut -d <delimiter> -f<field#> :: to isolate the item



```
#!/bin/bash

#MENU RECIPE whiptail //auto generate menu items

WC_L_ITEMS_COUNT=$(lsblk -f -o NAME,FSTYPE,FSVER,LABEL,SIZE | grep ext4 | sed 's/├─//g' | sed 's/└─//g' | sed 's/ /·/g' | wc -l)

#you can also use nl -s <separator> instead of cat -n to enumerate lines (at start of each line)
CAT_N__ITEMS=$(lsblk -f -o NAME,FSTYPE,FSVER,LABEL,SIZE | grep ext4 | sed 's/├─//g' | sed 's/└─//g' | sed 's/ /·/g' | cat -n)

CHOICE=$(whiptail --title "protoype menu" --menu " choose" 0 0 $WC_L_ITEMS_COUNT $CAT_N__ITEMS 3>&1 1>&2 2>&3)

lsblk -f -o NAME,FSTYPE,FSVER,LABEL,SIZE | grep ext4 | sed 's/├─//g' | sed 's/└─//g' | sed 's/ /·/g' | cat -n | grep -w $CHOICE

```

NOTE!: remember to replace spaces with a character for menu items, or else it will break into a new menu item at each space location

How to use whiptail to create more user-friendly interactive scripts

Do you script in bash? If so, you can provide your users with a more robust and simple TUI for entering information into scripts.

Posted: January 20, 2021 | | Damon Garn

*Photo by **Pexels***

Few sysadmins in the Linux world need to be convinced of the power and importance of scripts. Scripts are everywhere, and you know they're essential to Linux system administration. Many scripts run silently, even if they are initiated manually by a user or an admin.

Some scripts, however, pass information to users or solicit information from them. You can use scripting features such as `echo` or `read` to accomplish these goals. Unfortunately, neither of these tools display the information in a fancy way or in a manner that gets the user's attention.

Whiptail adds a more interactive dialog box to your scripts. These boxes provide information, solicit input, or force an acknowledgment. The content displayed is in a text-based user interface (TUI) format and is navigated with the **Tab** key. Selections are chosen with the **Space** key.

In this article, I walk you through the installation of `whiptail` (it's easy!) and then demonstrate a few basic examples.

[Readers also liked: Which programming languages are useful for sysadmins?]

Install whiptail

Installation is easy. I am using Fedora 33 for these examples. Whiptail is included as part of the larger newt library, which adds functionality to TUI windows.

Install `whiptail` by using the following command:

```
[damon@localhost ~]$ sudo dnf install newt
```

That's it for the installation.

In this article, you will create individual vim files with the whiptail-specific code. Remember to accomplish the following tasks for each demonstrated file:

- Name it clearly
- Make it executable by typing `chmod 744 demo.sh`
- Run it by using `./` before the filename if the location is not along the PATH

Note: I assume vim because I prefer it. Nano or any other text editor is sufficient.

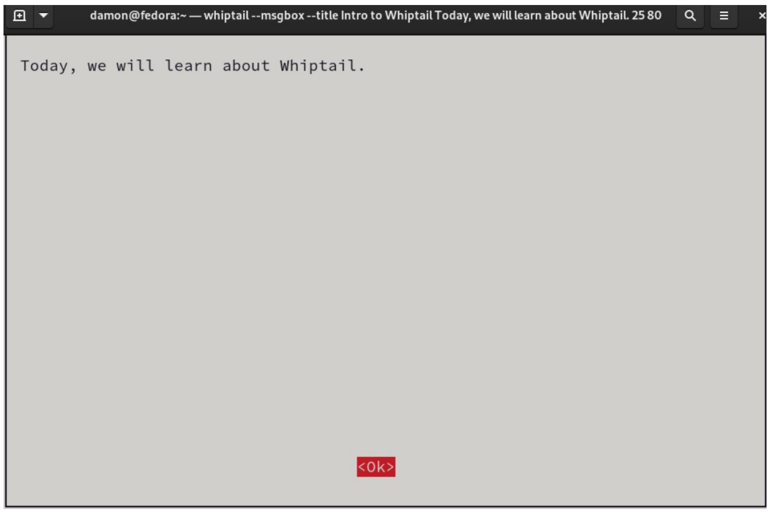
Next, it's time to get some scripts involved.

Display a basic dialog box

You don't actually need a script to display a basic dialog box from `whiptail`. In this example, you'll declare and then call a variable. The variable is merely a message of some sort that will be acknowledged with an **OK** button. No choices are offered, and no navigation is used.

At the command prompt, type the following information:

```
[damon@localhost ~]$ message="Today, we will learn about Whiptail."
[damon@localhost ~]$ whiptail --msgbox --title "Intro to Whiptail"
"$message" 25 80
```



damon@fedora:~ — whiptail --msgbox --title Intro to Whiptail Today, we will learn about Whiptail. 25 80

```
Today, we will learn about Whiptail.
```

Dialog box generated by whiptail. Note the title, message, and OK fields.

Once the interface is displayed, notice the title that you specified appears in the top bar. Your message is also present. The **OK** button is available. In the `whiptail` command, you entered two values: **25** and **80**. Those values are column measurements and may be adjusted. They define the size of the interface window. Be careful to select a size that doesn't consume the entire screen and prevent the user from seeing the whole message or selecting **OK**. Most Terminal windows will be set to 80 columns or more.

Once you have observed all of the components of the interface, select **OK** by pressing the **Enter** key.

In this example, you used two options: `--msgbox` and `--title`

And now a more interesting example.

Automation advice

- Ansible Automation Platform beginner's guide
- A system administrator's guide to IT automation
- Ansible Automation Platform trial subscription
- Automate Red Hat Enterprise Linux with Ansible and Satellite

Generate a query

Scripts may be written that accepts a user's input. If you are writing a script that will interact with non-technical users, it may be beneficial to create a more user-friendly interface. In this example, the user will be asked two questions: Name and country.

Create a file named `query.sh` with the following content.

```
#Part 1 - Query for the user's name
NAME=$(whiptail --inputbox "What is your name?" 8 39 --title "Getting to
know you" 3>&1 1>&2 2>&3)

exitstatus=$?
if [ $exitstatus = 0 ]; then
echo "Greetings," $NAME
else
echo "User canceled input."
fi

echo "(Exit status: $exitstatus)"

#Part 2 - Query for the user's country

COUNTRY=$(whiptail --inputbox "What country do you live in?" 8 39 --
title "Getting to know you" 3>&1 1>&2 2>&3)

exitstatus=$?
if [ $exitstatus = 0 ]; then
echo "I hope the weather is nice in" $COUNTRY
else
echo "User canceled input."
fi

echo "(Exit status: $exitstatus)"
```

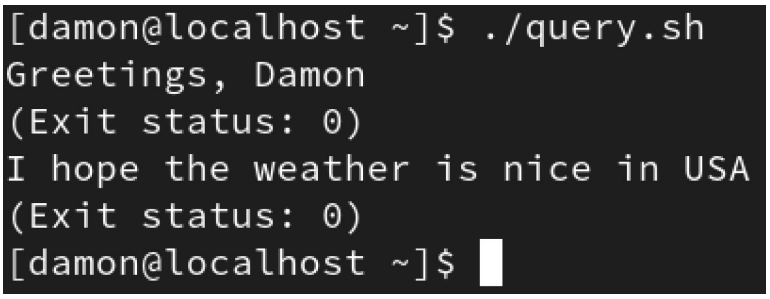
Observe that in this example, you used `--inputbox` instead of `--msgbox`. You've organized the `whiptail` code as an if/then statement.

The **8** and **39** values define the size of the dialog box. If you are following along in your own lab environment, replace **39** with **10**, save your changes, and rerun the `whiptail` code. You will see that the dialog box is too small to be useful. Use the **Tab** key to select **Cancel**, and then set the size value back to **39** from **10**.

Don't forget to set the permissions to make the file executable and use `./` to run it from the current location.

```
[damon@localhost ~]$ chmod 777 query.sh
[damon@localhost ~]$ ./query.sh
```

Here is a look at the results after completing the dialog box:



```
[damon@localhost ~]$ ./query.sh
Greetings, Damon
(Exit status: 0)
I hope the weather is nice in USA
(Exit status: 0)
[damon@localhost ~]$
```

Result of the whiptail dialog box displaying the user name and country, as queried.

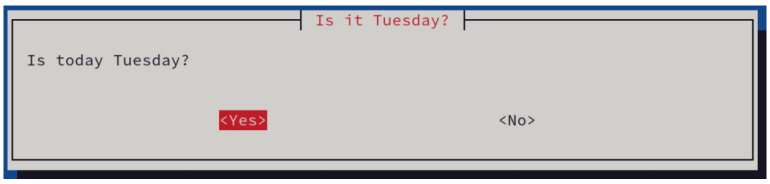
You've given users information via a message box and gathered information from the user via an input box. Next, ask the user some additional questions.

Create a yes/no dialog box

There are plenty of variations for asking questions of the user. In this case, you'll use a simple yes/no query to discover whether today is Tuesday. You can do this by creating a test file named `tuesday.sh` and placing the following content into it:

```
if (whiptail --title "Is it Tuesday?" --yesno "Is today Tuesday?" 8
78); then
    echo "Happy Tuesday, exit status was $?."
else
    echo "Maybe it will be Tuesday tomorrow, exit status was $?."

fi
```

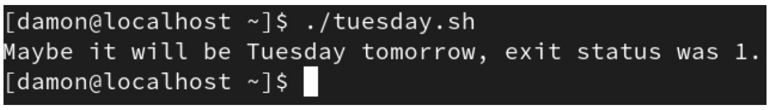
Instead of the `--inputbox` or `--msgbox` from the earlier examples, you used the `--yesno` option. Like the previous example, this one is organized as an if/then query.



```
Is it Tuesday?

Is today Tuesday?

<Yes>                    <No>
```

Query as to whether today is Tuesday. Note the "Yes" and "No" answer options.

Here is the resulting output:



```
[damon@localhost ~]$ ./tuesday.sh
Maybe it will be Tuesday tomorrow, exit status was 1.
[damon@localhost ~]$
```

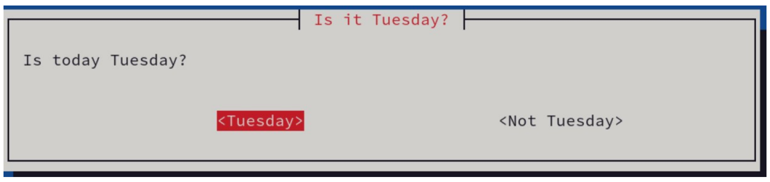
Output from the tuesday.sh query.

The `--yesno` box option also permits us to edit the content of the "Yes" and "No" fields. Here is an example:

```
if (whiptail --title "Is it Tuesday?" --yesno "Is today Tuesday?" 8 78
--no-button "Not Tuesday" --yes-button "Tuesday"); then
echo "Happy Tuesday, exit status was $?."
else
echo "Maybe it will be Tuesday tomorrow, exit status was $?."
fi
```

The only modification to the example is the addition of the two `--no-button` and `--yes-button` options, with their corresponding text (in this case, "Not Tuesday" and "Tuesday").

Here is what the resulting dialog box looks like:


```
Is it Tuesday?

Is today Tuesday?

<Tuesday>                    <Not Tuesday>
```

Observe that the buttons no longer offer Yes and No options, but instead Tuesday and Not Tuesday options.

One last thing. Whiptail sends the user's input to stderr. Yes, you read that correctly: stderr, and

not stdout, which is where you pick up the user's input to consume it in the script. The way around this issue is to reverse the redirection so that the user's input goes to stdout.

Here is the phrase for doing that:

3>&1 1>&2 2>&3

Explanation:

- Create a file descriptor 3 that points to 1 (stdout)
- Redirect 1 (stdout) to 2 (stderr)
- Redirect 2 (stderr) to the 3 file descriptor, which is pointed to stdout

Linux containers

- A practical introduction to container terminology
- Containers primer
- Download now: Red Hat OpenShift trial
- eBook: Podman in Action
- Why choose Red Hat for containers

Here is how it looks in a script snippet from the above `--inputbox` example:

```
NAME=$(whiptail --inputbox "What is your name?" 8 39 --title "Getting to know you" 3>&1 1>&2 2>&3)
```

[Free cheat sheet: Kubernetes glossary]

Wrap up

I've shown several basic examples, but `whiptail` can do a lot more. I've provided enough here to get you going and there are many useful tutorials online. Your scripts will need a way to consume the input the users have entered. I encourage you to review your interactive scripts to determine whether adding TUI dialog boxes would helpful.

Here's a list of the primary box options available for `whiptail`:

- --title
- --infobox
- --msgbox
- --yesno
- --inputbox
- --passwordbox
- --menu
- --textbox
- --checklist
- --radiolist
- --gauge