qtile-extras

Release 0.22.2.dev0+gbed30ac.d20220922

elParaguayo

Sep 22, 2022

GETTING STARTED

`qtile-extras` is a collection of mods made by elParaguayo for Qtile.

These are mods that I've made but which, for various reasons, may not end up in the main Qtile codebase.

They're designed for me but I've shared them in case they're of interest to anyone else.

Currently, this repo houses some *widgets* that I made as well as my "*popup toolkit*" which can be used to extend widgets or make standalone menus/launchers.

The new widgets are:

- *ALSAWidget*
- *AnalogueClock*
- *BrightnessControl*
- *CurrentLayoutIcon*
- *GithubNotifications*
- *GlobalMenu*
- *LiveFootballScores*
- *ScriptExit*
- *SnapCast*
- *StatusNotifier*
- *StravaWidget*
- *TVHWidget*
- *UPowerWidget*
- *UnitStatus*
- *Visualiser*
- *Visualizer*
- *WiFiIcon*
- *WordClock*

There's a *mixin* if you want to add tooltips to widgets.

I've also added some "eye candy" in the form of:

- *Widget Decorations*
- *Wallpapers*

Lastly, I've created a new `ImgMask` class which, rather than drawing the source image, uses the source as a mask for the drawing. This can be used to change the colour of icons without needing to recrate the icons themselves. You can see the class *here*.

Note: These items are made primarily for my use and are not officially supported by Qtile. You are most welcome to install it and I hope that you find some parts of is useful. However, please note, I cannot guarantee that I will continue to maintain certain aspects of this repo if I am no longer using them so, be warned, things may break!

Important: This repo is designed to be installed alongside the git version of Qtile. It is highly recommended that you run the latest version to ensure compatibility.

I will not maintain a version of this repo that is linked to tagged releases of Qtile.

CHAPTER

ONE

INSTALLATION

Note: I have no current intentions to package this on PyPi. This means installation may be a bit more "manual" than for other packages.

1.1 Arch users

This is the easiest option as the package is in the AUR. Using your favourite helper, you just need to download and install the `qtile-extras-git` package.

Alternatively, you can download the PKGBUILD file from the repo and run `makepkg -sci`.

1.2 Fedora

There is no official package for Fedora yet but you can install it from Copr:

```
dnf copr enable frostyx/qtile
dnf install qtile-extras
```

1.3 Everyone else

Clone the repo and run `python setup.py install`.

CHAPTER

TWO

USING THE POPUP TOOLKIT

This guide explains how to create popups that can be used to add functionality to widgets or create standalone launchers.

2.1 What's in the toolkit?

The Toolkit has two types of object, a layout and a control. The layout is the container that helps organise the presentation of the popup. The controls are the objects that display the content.

A simple comparison would be to think of the `Bar` as the layout and widgets as the controls. However, a key difference of this toolkit is that the controls can be placed anywhere in a 2D space whereas widgets can only be ordered in one dimension.

2.2 Layouts

The toolkit provides three layouts: `PopupGridLayout`, `PopupRelativeLayout` and `PopupAbsoluteLayout`.

Descriptions and configuration options of these layouts can be found on *the reference page*.

2.3 Controls

Currently, the following controls are provided:

- `PopupText`: a simple text display object
- `PopupImage`: a control to display an image
- `PopupSlider`: a control to draw a line which marks a particular value (e.g. volume level)
- `PopupWidget`: a control to display a Qtile widget in the popup

Configuration options for these controls can be found on *the reference page*.

2.4 Callbacks

To add functionality to your popup, you need to bind callbacks to the individual controls. This is achieved in the same way as widgets i.e. a dictionary of `mouse_callbacks` is passed as a configuration option for the control. The control accepts any callable function but also accepts `lazy` objects like those used for key bindings.

2.5 Building a popup menu

Below is an example of creating a power menu in your `config.py`.

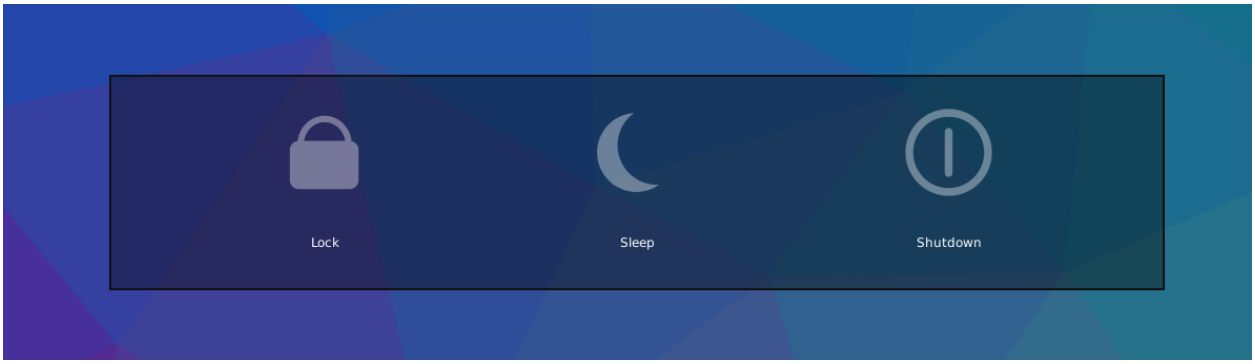
```
from qtile_extras.popup.toolkit import (
    PopupRelativeLayout,
    PopupImage,
    PopupText
)

def show_power_menu(qtile):

    controls = [
        PopupImage(
            filename="~/Pictures/icons/lock.svg",
            pos_x=0.15,
            pos_y=0.1,
            width=0.1,
            height=0.5,
            mouse_callbacks={
                "Button1": lazy.spawn("/path/to/lock_cmd")
            }
        ),
        PopupImage(
            filename="~/Pictures/icons/sleep.svg",
            pos_x=0.45,
            pos_y=0.1,
            width=0.1,
            height=0.5,
            mouse_callbacks={
                "Button1": lazy.spawn("/path/to/sleep_cmd")
            }
        ),
        PopupImage(
            filename="~/Pictures/icons/shutdown.svg",
            pos_x=0.75,
            pos_y=0.1,
            width=0.1,
            height=0.5,
            highlight="A00000",
            mouse_callbacks={
                "Button1": lazy.shutdown()
            }
        ),
        PopupText(
```

(continues on next page)

(continued from previous page)

```
            text="Lock",
            pos_x=0.1,
            pos_y=0.7,
            width=0.2,
            height=0.2,
            h_align="center"
        ),
        PopupText(
            text="Sleep",
            pos_x=0.4,
            pos_y=0.7,
            width=0.2,
            height=0.2,
            h_align="center"
        ),
        PopupText(
            text="Shutdown",
            pos_x=0.7,
            pos_y=0.7,
            width=0.2,
            height=0.2,
            h_align="center"
        ),
    ]

    layout = PopupRelativeLayout(
        qtile,
        width=1000,
        height=200,
        controls=controls,
        background="00000060",
        initial_focus=None,
    )

    layout.show(centered=True)

keys = [
    ...
    Key([mod, "shift"], "q", lazy.function(show_power_menu))
    ...
]
```

Now, when you press `Mod+shift+q` you should see a menu looking like this:



2.6 Using widgets in a popup

It is possible to display widgets in a popup window and not just in the bar. This is possible by using the `PopupWidget` control.

Below is a quick example for displaying a number of graph widgets in a popup:

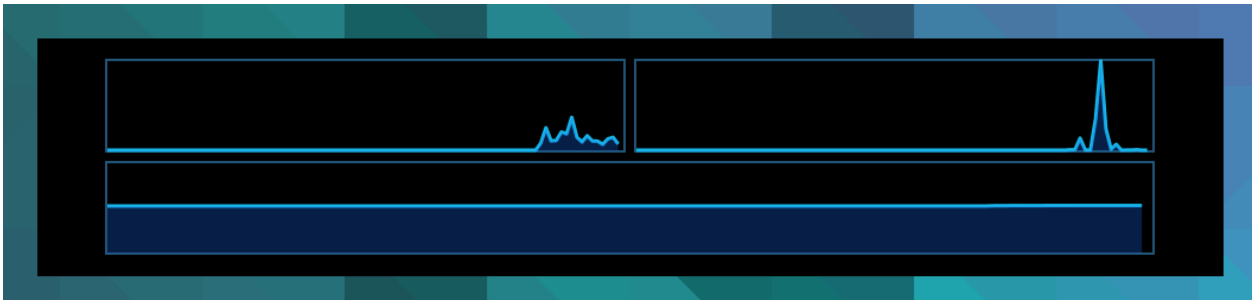
```
from libqtile import widget
from qtile_extras.popup.toolkit import (
    PopupRelativeLayout,
    PopupWidget
)

def show_graphs(qtile)
    controls = [
        PopupWidget(
            widget=widget.CPUGraph(),
            width=0.45,
            height=0.45,
            pos_x=0.05,
            pos_y=0.05
        ),
        tk.PopupWidget(
            widget=widget.NetGraph(),
            width=0.45,
            height=0.45,
            pos_x=0.5,
            pos_y=0.05
        ),
        tk.PopupWidget(
            widget=widget.MemoryGraph(),
            width=0.9,
            height=0.45,
            pos_x=0.05,
            pos_y=0.5
        )
    ]

    layout = tk.PopupRelativeLayout(
        qtile,
```

(continues on next page)

(continued from previous page)

```
        width=1000,
        height=200,
        controls=controls,
        background="00000060",
        initial_focus=None,
        close_on_click=False
    )
    layout.show(centered=True)

keys = [
    ...
    Key([mod, "shift"], "g", lazy.function(show_graphs))
    ...
]
```

Pressing `Mod+shift+g` will present a popup window looking like this:



2.7 Extending widgets

[To be drafted]

CHAPTER

THREE

WIDGET DECORATIONS

Widget decorations are additional content that is drawn to your widget before the main content is rendered i.e. you can add drawings behind your widgets.

3.1 Types of decoration

The following decorations are available:

- *BorderDecoration*
- *PowerLineDecoration*
- *RectDecoration*

3.2 Adding decorations to your widgets

All widgets available from this repo can have decorations added to them.

In addition, all widgets in the main Qtile repository can also have decorations attached. To do this, you simply need to change the import in your config file, replacing:

```
from libqtile import widget
```

with:

```
from qtile_extras import widget
```

A fuller example would look like this:

```
from qtile_extras import widget
from qtile_extras.widget.decorations import RectDecoration

decor = {
    "decorations": [
        RectDecoration(colour="#600060", radius=10, filled=True, padding_y=5)
    ],
    "padding": 18,
}

screens = [
```

(continues on next page)

(continued from previous page)

```
    Screen(
        bottom=bar.Bar(
            [
                widget.GroupBox(**decor),
                ...
                widget.Clock(**decor),
                widget.QuickExit(**decor),
            ]
        )
    )
]
```

3.3 Adding decorations to user-defined widgets

You can also add the ability to draw decorations to your own widgets.

First, you need to import `modify` from `qtile_extras.widget` and use this to wrap your widget class and its configuration parameters. i.e. calling `modify(WidgetClass, *args, **kwargs)` will return `WidgetClass(*args, **kwargs)`.

```
from libqtile.config import Screen
from libqtile.widget.base import _TextBox

from qtile_extras.bar import Bar
from qtile.extras.widget import modify


class MyTextWidget(_TextBox):
    pass


screens = [
    Screen(
        bottom=Bar(
            [
                ...
                modify(
                    MyTextWidget,
                    text="Modded widget",
                    decorations=[
                        ...
                    ]
                ),
                ...
            ]
        )
    )
]
```

CHAPTER

FOUR

IMAGE MASK

This is a new image class that allows you provide a source image to use as a mask. Painting with a colour then renders that colour in unmasked areas. The advantage of this is that the colour can be set dynamically without having to preload different images.

The example below shows a simple widget using this class to display three icons.

```
from libqtile import bar
from libqtile.widget.base import _Widget

from qtile_extras.images import ImgMask

ICON_PATH = "/path/to/icon_folder"

class MaskWidget(_Widget):
    def __init__(self):
        _Widget.__init__(self, bar.CALCULATED)

    def _configure(self, qtile, bar):
        _Widget._configure(self, qtile, bar)
        self.img = ImgMask.from_path(f"{ICON_PATH}/icon.svg")
        self.img.attach_drawer(self.drawer)
        self.img.resize(self.bar.height - 1)

    def calculate_length(self):
        if not self.configured:
            return 0

        return self.img.width * 3

    def draw(self):
        self.drawer.clear(self.background or self.bar.background)
        offset = 0
        for col in [
            "ff0000",
            "00ff00",
            ["ff00ff", "0000ff", "00ff00", "ff0000", "ffff00"]
        ]:
            self.img.draw(colour=col, x=offset)
            offset += self.img.width

        self.drawer.draw(
```

(continues on next page)

(continued from previous page)

```
            offsetx=self.offset,
            offsety=self.offsety,
            width=self.length
        )
```

Placing an instance of `MaskWidget()` in your bar will then give you something like this:

Fig. 1: Note you can use gradients too.

Note: It is important that the `ImgMask` object has a reference to the widget's `drawer` attribute. In the example above, this is achieved via the call to `self.img.attach_drawer(self.drawer)`.

4.1 Batch Loader

If you want to use the `Loader` class to load a batch of images to use as masks, you can do that as follows (note the use of the `masked=True` keyword argument):

```
from qtile_extras.images import Loader

image_dict = Loader(IMAGE_FOLDER, masked=True)(*IMAGE_NAMES)
```

As above, the images will need to have the widget's `drawer` object attached.

CHAPTER

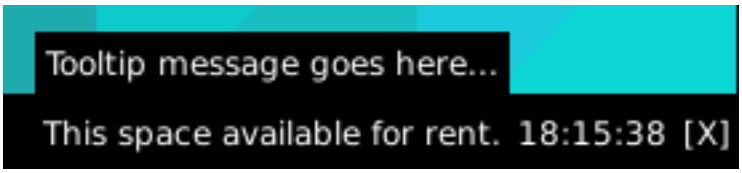
FIVE

WIDGET TOOLTIPS

Using the `TooltipMixin` allows you to add a tooltip to any widget. This is best illustrated with a simple example:

```
from libqtile.widget import TextBox

from qtile_extras.widget.mixins import TooltipMixin


class TooltipTextBox(TextBox, TooltipMixin):

    def __init__(self, *args, **kwargs):
        TextBox.__init__(self, *args, **kwargs)
        TooltipMixin.__init__(self, **kwargs)
        self.add_defaults(TooltipMixin.defaults)

        # The tooltip text is set in the following variable
        self.tooltip_text = "Tooltip message goes here..."

# Add an instance of TooltipTextBox to your bar
# e.g. TooltipTextBox("This space available for rent.")
```

When you hover your mouse over the widget you will see a message appear after a short delay:



See the *reference page* for instructions on how to customise the mixin.

CHAPTER

SIX

WIDGETS

6.1 ALSAWidget

class qtile_extras.widget.**ALSAWidget**(***config*)

The widget is very simple and, so far, just allows controls for volume up, down and mute.

Volume control is handled by running the appropriate amixer command. The widget is updated instantly when volume is changed via this code, but will also update on an interval (i.e. it will reflect changes to volume made by other programs).

The widget displays volume level via an icon, bar or both. The icon is permanently visible while the bar only displays when the volume is changed and will hide after a user-defined period.

Supported bar orientations: horizontal only

Fig. 1: 'icon' mode

Fig. 2: 'bar' mode

Fig. 3: 'both' mode

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

cmd_toggle_mute(**args*, ***kwargs*)

Mute audio output

cmd_volume_down(**args*, ***kwargs*)

Decrease volume

cmd_volume_up(**args*, ***kwargs*)

Increase volume

6.2 AnalogueClock

class qtile_extras.widget.**AnalogueClock**(***config*)

An analogue clock for your Bar.

Supported bar orientations: horizontal and vertical

Fig. 4: Default config

Fig. 5: With square clock face

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

6.3 BrightnessControl

class qtile_extras.widget.**BrightnessControl**(***config*)

This module provides basic screen brightness controls and a simple widget showing the brightness level for Qtile.

Brightness control is handled by writing to the appropriate `/sys/class/backlight` device. The widget is updated instantly when the brightness is changed via this code and will autohide after a user-defined timeout.

Note: This script will not work unless the user has write access to the relevant backlight device.

This can be achieved via a udev rule which modifies the group and write permissions. The rule should be saved at /etc/udev/rules.d

An example rule is as follows:

```
# Udev rule to change group and write permissions for screen backlight
ACTION=="add", SUBSYSTEM=="backlight", KERNEL=="intel_backlight", RUN+="/bin/chgrp␣
↪video /sys/class/backlight/%k/brightness"
ACTION=="add", SUBSYSTEM=="backlight", KERNEL=="intel_backlight", RUN+="/bin/chmod␣
↪g+w /sys/class/backlight/%k/brightness"
```

You should then ensure that your user is a member of the `video` group.

Supported bar orientations: horizontal only

cmd_brightness_down()

Decrease the brightness level

cmd_brightness_up()

Increase the brightness level

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

cmd_set_brightness_percent(*percent*)

Set brightness to percentage (0.0-1.0) of max value

cmd_set_brightness_value(*value*)

Set brightess to set value

6.4 CurrentLayoutIcon

class qtile_extras.widget.**CurrentLayoutIcon**(***config*)

A modified version of Qtile's `CurrentLayoutIcon`.

The default version behaves the same as the main qtile version of the widget. However, if you set `use_mask` to `True` then you can set the colour of the icon via the `foreground` parameter.

Supported bar orientations: horizontal only

Fig. 6: You can use a single colour or a list of colours.

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

cmd_set_font(*font=UNSPECIFIED*, *fontsize=UNSPECIFIED*, *fontshadow=UNSPECIFIED*)

Change the font used by this widget. If font is None, the current font is used.

6.5 GithubNotifications

class qtile_extras.widget.**GithubNotifications**(***config*)

A widget to show when you have new github notifications.

The widget requires a personal access token (see here). The token needs the `notifications` scope to be enabled. This token should then be saved in a file and the path provided to the `token_file` parameter.

If your key expires, re-generate a new key, save it to the same file and then call the `reload_token` command (e.g. via `qtile cmd-obj`).

Required Dependencies

This module requires the following third-party libraries: `requests`

Supported bar orientations: horizontal only

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

cmd_reload_token()

Force reload of access token.

6.6 GlobalMenu

Warning: This class has been marked as experimental.

The widget may behave unexpectedly, have missing features and will probably crash at some point!

Feedback on any issues would be appreciated.

class qtile_extras.widget.**GlobalMenu**(***config*)

A widget to display a Global Menu (File Edit etc.) in your bar.

Only works with apps that export their menu via DBus.

This is not currently available on Wayland backends but I may try to see if I can get it working at some point!

Required Dependencies

This module requires the following third-party libraries: `dbus-next`

Supported bar orientations: horizontal only

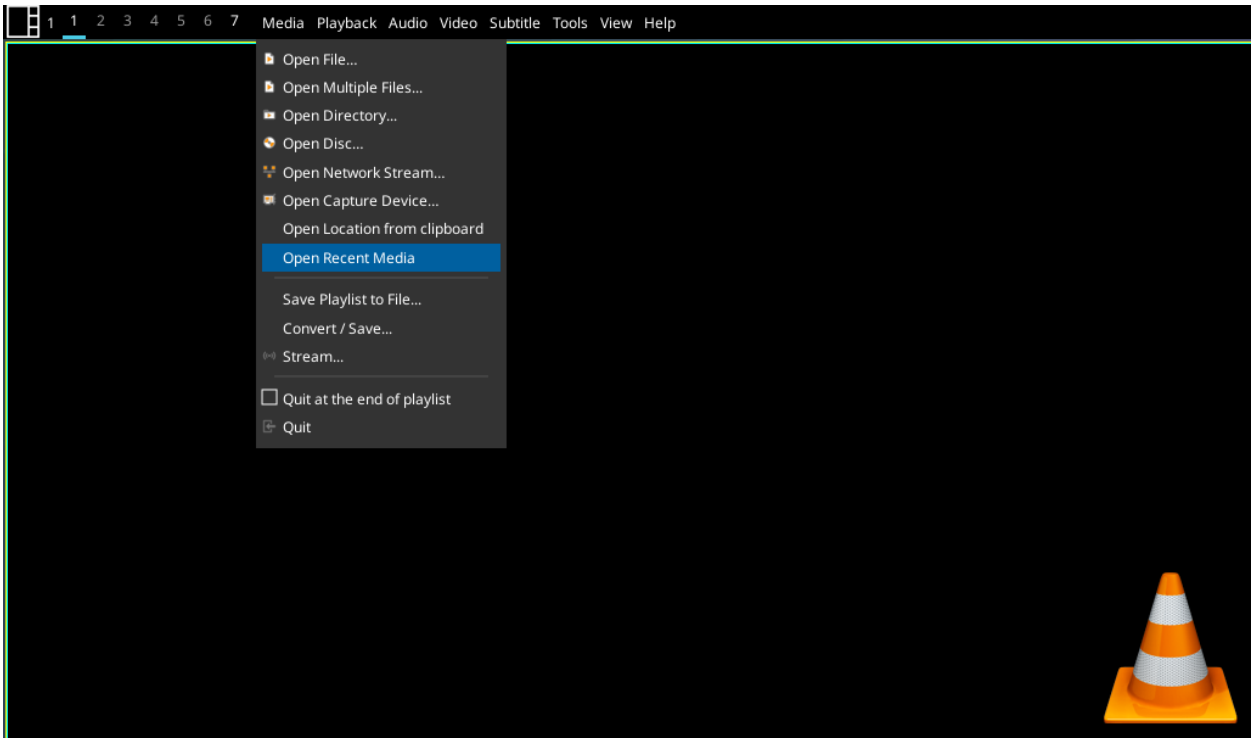


Fig. 7: Showing VLC menu in the bar.

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

cmd_set_font(*font=UNSPECIFIED*, *fontsize=UNSPECIFIED*, *fontshadow=UNSPECIFIED*)

Change the font used by this widget. If font is None, the current font is used.

6.7 LiveFootballScores

Warning: This class has been marked as experimental.

The widget may behave unexpectedly, have missing features and will probably crash at some point!

Feedback on any issues would be appreciated.

class qtile_extras.widget.**LiveFootballScores**(***config*)

The module uses a module I wrote a number of years ago that parses data from the BBC Sport website.

The underlying module needs work so it will probably only work if you pick a "big" team.

You can select more than one team and league. Scores can be scrolled by using the mousewheel over the widget.

Goals and red cards are indicated by a coloured bar next to the relevant team name. The match status is indicated by a coloured bar underneath the match summary. All colours are customisable.

Required Dependencies

This module requires the following third-party libraries: `requests`

Supported bar orientations: horizontal only

Fig. 8: The different screens show: live score, elapsed time, home and away goalscorers and competition name. In addition, the amount of text shown can be customised by using python's string formatting techniques e.g. the default line `{H:.3} {h}-{a} {A:.3}` shows the first 3 letters of team names rather than the full name as shown above.

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_get()

Get displayed text. Removes padding.

cmd_info()

Show information about all matches

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

cmd_popup()

Display popup window

cmd_reboot()

Restart the widget. Useful if updates seem to stop.

cmd_refresh()

Force a poll of match data

6.8 ScriptExit

class qtile_extras.widget.**ScriptExit**(***config*)

An updated version of Qtile's QuickExit widget.

Takes an additional argument `exit_script` which will be run before qtile exits.

Supported bar orientations: horizontal and vertical

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

cmd_set_font(*font=UNSPECIFIED*, *fontsize=UNSPECIFIED*, *fontshadow=UNSPECIFIED*)

Change the font used by this widget. If font is None, the current font is used.

cmd_trigger()

6.9 SnapCast

Warning: This class has been marked as experimental.

The widget may behave unexpectedly, have missing features and will probably crash at some point!

Feedback on any issues would be appreciated.

class qtile_extras.widget.**SnapCast**(***config*)

A widget to run a snapclient instance in the background.

This is a work in progress. The plan is to add the ability for the client to change groups from widget.

Required Dependencies

This module requires the following third-party libraries: `requests`

Supported bar orientations: horizontal only

Fig. 9: Snapclient active running in background

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

6.10 StatusNotifier

Warning: This class has been marked as experimental.

The widget may behave unexpectedly, have missing features and will probably crash at some point!

Feedback on any issues would be appreciated.

class qtile_extras.widget.**StatusNotifier**(***config*)

A modified version of the default Qtile StatusNotifier widget.

Added the ability to render context menus by right-clicking on the icon.

Required Dependencies

This module requires the following third-party libraries: `dbus-next`, `xdg`

Supported bar orientations: horizontal and vertical

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

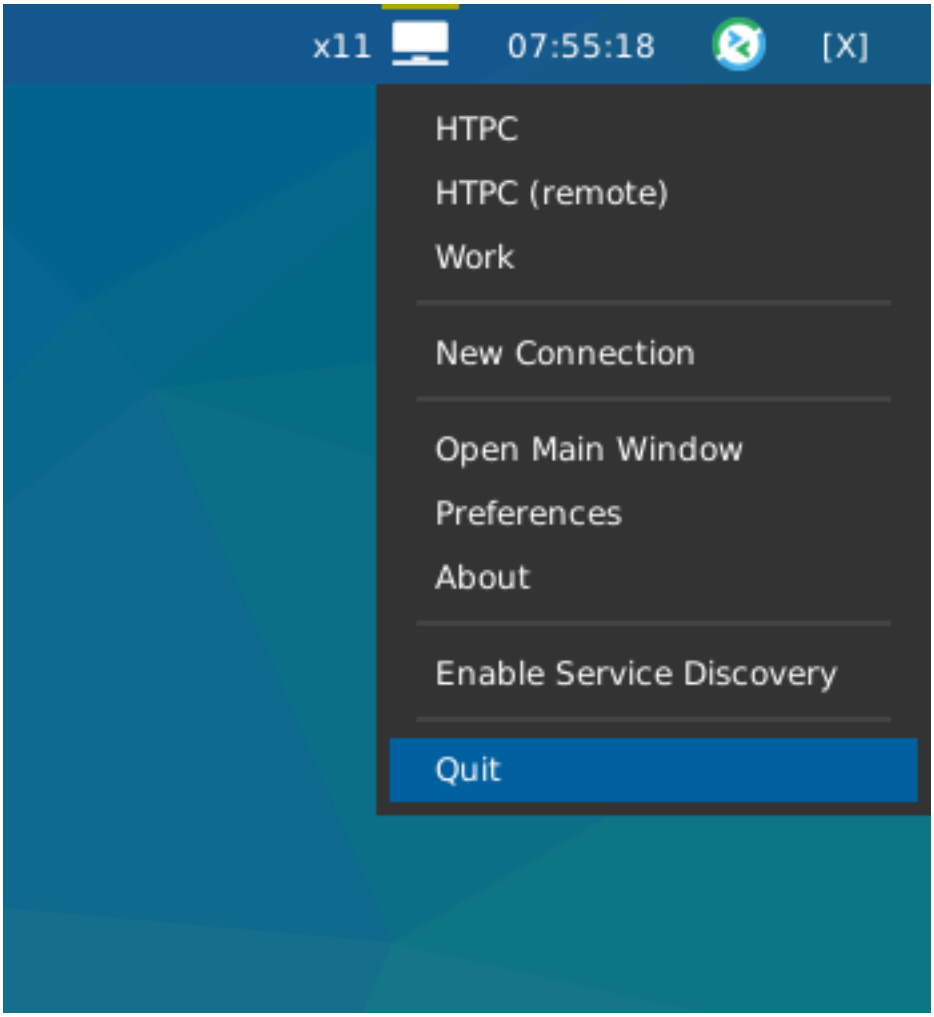


Fig. 10: Widget showing Remmina icon and context menu.

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

6.11 StravaWidget

Warning: This class has been marked as experimental.

The widget may behave unexpectedly, have missing features and will probably crash at some point!

Feedback on any issues would be appreciated.

class qtile_extras.widget.**StravaWidget**(***config*)

This module provides a simple widget showing some Strava stats.

The widget text can be customised using the following keys:

prefix	suffix	value
C		Current month
Y		Calendar year
A		All time
	D	Distance
	C	Count
	T	Time
	P	Pace
	N	Name
	A	Date

For example, the default text `{CA:%b} {CD:.1f}km` displays the current date in abbreviated month name format and the distance run that month: "Aug 143.1km".

Extended info is provided by clicking on the widget.

Note: You will need to follow the instuctions at https://developers.strava.com/ to create a new app and authorise it.

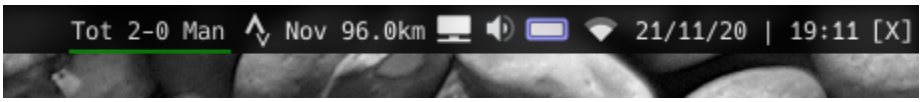
Your id and secret should be put in a json file called `auth.json` in `~/.cache/stravawidget`

The token file generated by the authorisation process, strava.json, should also be placed in the same folder.

Required Dependencies

This module requires the following third-party libraries: `stravalib`, `units`

Supported bar orientations: horizontal only



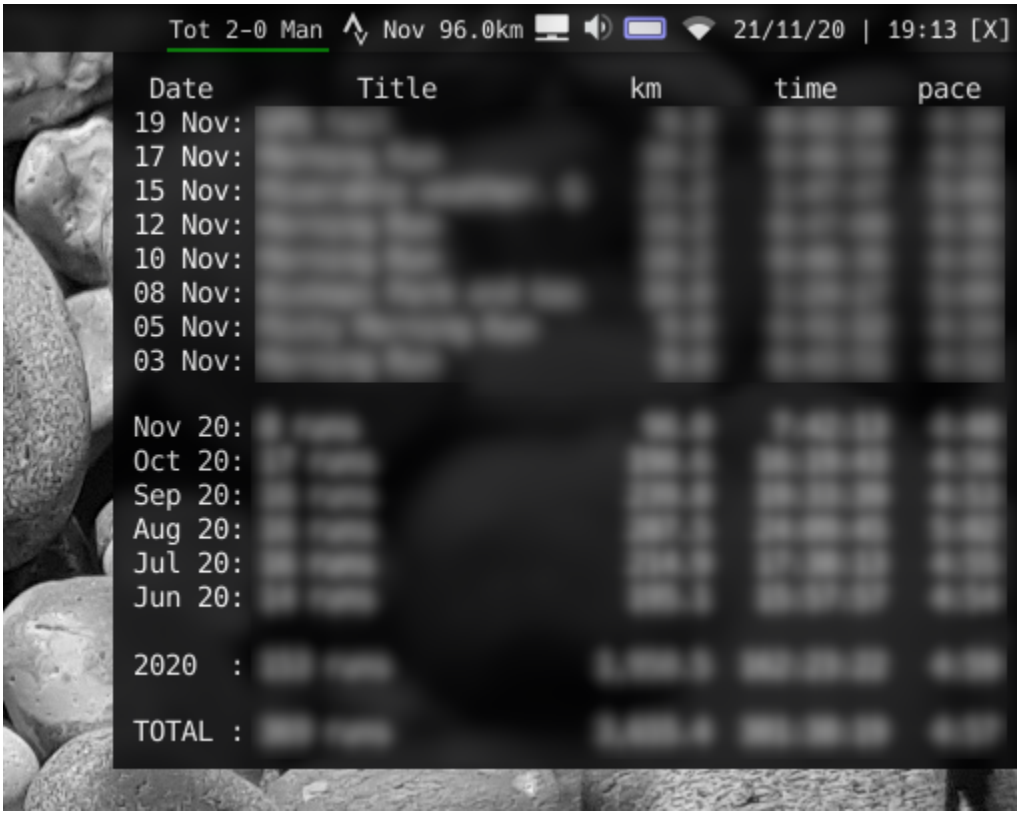


Fig. 11: Extended info. I've blurred out details of my runs for privacy reasons.

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

6.12 TVHWidget

class qtile_extras.widget.**TVHWidget**(***config*)

A widget to show whether a TVHeadend server is currently recording or not.

The widget will also show a popup displaying upcoming recordings.

When the server is recording a red line will be shown under the icon. If there's an error, a yellow line will show above the icon (and check the logs).

NB if you use a username and password, these are stored in plain text. You may therefore wish to create an unprivileged user account in TVHeadend that only has access to scheduled recordings data.

Required Dependencies

This module requires the following third-party libraries: `requests`

Supported bar orientations: horizontal only

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

6.13 UPowerWidget

class qtile_extras.widget.**UPowerWidget**(***config*)

A graphical widget to display laptop battery level.

The widget uses dbus to read the battery information from the UPower interface.

The widget will display one icon for each battery found or users can specify the name of the battery if they only wish to display one.

Clicking on the widget will display the battery level and the time to empty/full.

All colours can be customised as well as low/critical percentage levels.

Required Dependencies

This module requires the following third-party libraries: `dbus-next`

Supported bar orientations: horizontal only

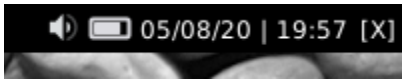


Fig. 12: Normal

Fig. 13: Low

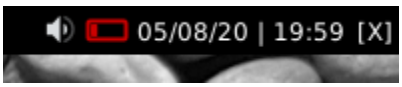


Fig. 14: Critical

Fig. 15: Charging

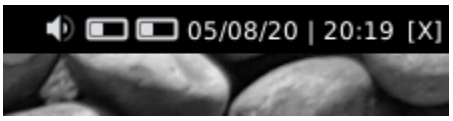


Fig. 16: Multiple batteries

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

6.14 UnitStatus

class qtile_extras.widget.**UnitStatus**(***config*)

UnitStatus is a basic widget for Qtile which shows the current status of systemd units.

It may not be particular useful for you and was primarily written as an exercise to familiarise myself with writing Qtile widgets and interacting with d-bus.

The widget is incredibly basic. It subscribes to the systemd d-bus interface, finds the relevant service and displays an icon based on the current status. The widget listens for announced changes to the service and updates the icon accordingly.

Fig. 17: Showing text

Required Dependencies

This module requires the following third-party libraries: `dbus-next`

Supported bar orientations: horizontal only

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

6.15 Visualiser

Warning: This class has been marked as experimental.

The widget may behave unexpectedly, have missing features and will probably crash at some point!

Feedback on any issues would be appreciated.

class qtile_extras.widget.**Visualiser**(***config*)

A widget to draw an audio visualiser in your bar.

The widget requires cava to be installed. This may also be packaged by your distro.

cava is configured through the widget. Currently, you can set the number of bars and the framerate.

Supported bar orientations: horizontal only

Fig. 18: Default config.

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

cmd_start()

Start the visualiser.

cmd_stop()

Stop this visualiser.

cmd_toggle()

Toggle visualiser state.

6.16 Visualizer

Warning: This class has been marked as experimental.

The widget may behave unexpectedly, have missing features and will probably crash at some point!

Feedback on any issues would be appreciated.

qtile_extras.widget.**Visualizer**

alias of *Visualiser*

6.17 WiFiIcon

class qtile_extras.widget.**WiFiIcon**(***config*)

An simple graphical widget that shows WiFi status.

Left-clicking the widget will show the name of the network.

Required Dependencies

This module requires the following third-party libraries: `iwlib`

Supported bar orientations: horizontal only

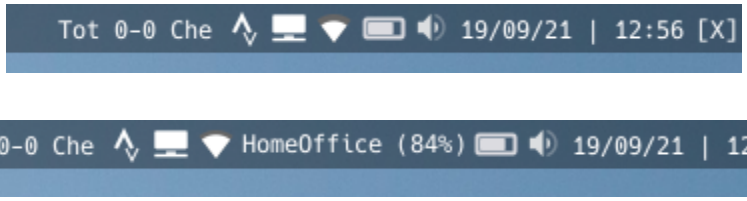


Fig. 19: Additional detail is visible when clicking on icon

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

cmd_show_text()

6.18 WordClock

class qtile_extras.widget.**WordClock**(***config*)

A widget to draw a word clock to the screen.

This is not a traditional widget in that you will not see anything displayed in your bar. The widget works in the background and updates the screen wallpaper when required. However, having this as a widget provides an easy way for users to install and configure the clock.

The clocks are currently designed to update on 5 minute intervals "five past" -> "ten past" etc. This may be changed in the future.

Custom layouts can be added by referring to the instructions in `qtile_extras/resources/wordclock/english.py`.

Supported languages

Available languages: `Dutch`, `English`, `Finnish`, `French`, `Portuguese`, `Spanish`, `Swedish`

Supported bar orientations: horizontal and vertical

cmd_commands() → list[str]

Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help

cmd_doc(*name*) → str

Returns the documentation for a specified command name

Used by __qsh__ to provide online help.

cmd_eval(*code: str*) → tuple[bool, str | None]

Evaluates code in the module namespace of the command object

Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(*function*, **args*, ***kwargs*) → None

Call a function with current object as argument

cmd_info()

Info for this object.

cmd_items(*name*) → tuple[bool, list[str | int] | None]

Returns a list of contained items for the specified name

Used by __qsh__ to allow navigation of the object graph.

6.19 Mixins

6.19.1 TooltipMixin

class qtile_extras.widget.mixins.**TooltipMixin**

Mixin that provides a tooltip for widgets.

To use it, subclass and add this to `__init__`:

```
TooltipMixin.__init__(self, **kwargs)
self.add_defaults(TooltipMixin.defaults)
```

Widgets should set `self.tooltip_text` to change display text.

CHAPTER

SEVEN

POPUP TOOLKIT

7.1 Popup Toolkit Layouts

Layouts are the container that houses all the controls. In order to give the greatest flexibility to users, there are a number of different layouts available in the toolkit.

7.1.1 PopupAbsoluteLayout

class qtile_extras.popup.toolkit.**PopupAbsoluteLayout**(*qtile*, ***config*)

The absolute layout is the simplest layout of all. Controls are placed based on the following parameters:

```
`pos_x`, `pos_y`: top left corner
`width`, `height`: size of control
```

No further adjustments are made to the controls.

Note: the layout currently ignores the `margin` attribute i.e. a control placed at (0,0) will display there even if a margin is defined.

7.1.2 PopupGridLayout

class qtile_extras.popup.toolkit.**PopupGridLayout**(*qtile*, ***config*)

The grid layout should be familiar to users who have used Tkinter.

In addition to the *width* and *height* attributes, the grid layout also requires *rows* and *cols* to define the grid. Grid cells are evenly sized.a

Controls can then be placed in the grid via the *row*, *col*, *row_span* and *col_span* parameters.

For example:

```
PopupGridLayout(rows=6, cols=6, controls=[
    PopupImage(filename="A",row=0, col=2, row_span=2, col_span=2),
    PopupImage(filename="B",row=2, col=2, row_span=2, col_span=2),
    PopupImage(filename="C",row=3, col=1),
    PopupImage(filename="D",row=3, col=4),
    PopupText(row=4,col_span=6),
])
```

would result in a tooltip looking like:

```
------------------------
|   |   |       |   |   |
---------   A   ---------
|   |   |       |   |   |
------------------------
|   |   |       |   |   |
---------   B   ---------
|   | C |       | D |   |
------------------------
|         TEXT           |
------------------------
|   |   |   |   |   |   |
------------------------
```

row and col are both zero-indexed.

7.1.3 PopupRelativeLayout

class qtile_extras.popup.toolkit.**PopupRelativeLayout**(*qtile*, ***config*)

The relative layout positions controls based on a percentage of the parent tooltip's dimensions.

The positions are defined with the following parameters:

```
`pos_x`, `pos_y`: top left corner
`width`, `height`: size of control
```

All four of these parameters should be a value between 0 and 1. Values outside of this range will generate a warning in the log but will not raise an exception.

For example:

```
PopupRelativeLayout(rows=6, cols=6, controls=[
    PopupImage(filename="A",pos_x=0.1, pos_y=0.2, width=0.5, height=0.5)
])
```

Would result in a tooltip with dimensions of 200x200 (the default), with an image placed at (20, 40) with dimensions of (100, 100).

Note: images are not stretched but are, instead, centered within the rect.

7.2 Popup Toolkit Controls

Controls are the building blocks for your popup. You can place text, images and progress bars and have each of these react to input events or display information dynamically.

7.2.1 PopupImage

class qtile_extras.popup.toolkit.**PopupImage**(***config*)

Control to display an image.

Image will be scaled (locked aspect ratio) to fit within the control rect. The image will also be centered vertically and horizontally.

7.2.2 PopupSlider

class qtile_extras.popup.toolkit.**PopupSlider**(*value=None*, ***config*)

Control to display slider/progress bar.

Bar can be displayed horizontally (draws left-to-right) or vertically (bottom-to-top).

7.2.3 PopupText

class qtile_extras.popup.toolkit.**PopupText**(*text=''*, ***config*)

Simple control to display text.

7.2.4 PopupWidget

class qtile_extras.popup.toolkit.**PopupWidget**(***config*)

Control to display a Qtile widget in a Popup window.

Mouse clicks are passed on to the widgets.

Currently, widgets will be sized based on the dimensions of the control. This will override any width/stretching settings in thw widget.

CHAPTER

EIGHT

DECORATIONS

8.1 BorderDecoration

class qtile_extras.widget.decorations.**BorderDecoration**(***config*)

Widget decoration that draws a straight line on the widget border. Padding can be used to adjust the position of the border further.

Only one colour can be set but decorations can be layered to achieve multi-coloured effects.

Fig. 1: Stacked borders

8.2 PowerLineDecoration

class qtile_extras.widget.decorations.**PowerLineDecoration**(***config*)

Widget decoration that can be used to recreate the PowerLine style.

The advantages of the decoration are:

- No fonts needed
- The same decoration definition can be used for all widgets (the decoration works out which background and foreground colours to use)
- Different styles can be used by changing a few parameters of the decoration

The decoration works by adding the shape **after** the current widget. The decoration will also be hidden if a widget has zero width (i.e. is hidden).

The shape is drawn in area whose size is defined by the `size` parameter. This area is drawn after the widget but can be shifted back by using the `shift` parameter. Shifting too far will result in widget contents being drawn over the shape.

The default behavious is to draw an arrow pointing right. To change the shape you can use pre-defined paths: "arrow_left", "arrow_right", "forward_slash", "back_slash" and "zig_zag". Alternatively, you can create a custom shape by defining a path. The format is a list of (x, y) tuples. x and y values should be between 0 and 1 to represent the relative position in the additional space created by the decoration. (0, 0) is the top left corner (on a horizontal widget) and (1, 1) is the bottom right corner. The first point in the list is the starting point and then a line will be drawn to each subsequent point. The path is then closed by returning to the first point. Finally, the shape is filled with the background of the current widget.

Note: The decoration currently only works on horizontal bars. The `padding_y` parameter can be used to adjust the vertical size of the decoration. However, note that this won't change the size of the widget's own background. If you want to do that, you can use the following:

```
powerline = {
    "decorations": [
        RectDecoration(use_widget_background=True, padding_y=5, filled=True,␣
↪radius=0),
        PowerLineDecoration(path="arrow_right", padding_y=5)
    ]
}
```

The RectDecoration has the same padding and will use the widget's `background` parameter as its own colour.

Example code:

```
from qtile_extras import widget
from qtile_extras.widget.decorations import PowerLineDecoration

powerline = {
    "decorations": [
        PowerLineDecoration()
    ]
}

screens = [
    Screen(
        top=Bar(
            [
                widget.CurrentLayoutIcon(background="000000", **powerline),
                widget.WindowName(background="222222", **powerline),
                widget.Clock(background="444444", **powerline),
                widget.QuickExit(background="666666")
            ],
            30
        )
    )
]
```

The above code generates the following bar:

Fig. 2: path='arrow_right'

Fig. 3: path='rounded_left'

Fig. 4: path='rounded_right'

8.3 RectDecoration

class qtile_extras.widget.decorations.**RectDecoration**(***config*)

Widget decoration that draws a rectangle behind the widget contents.

Only one colour can be set but decorations can be layered to achieve multi-coloured effects.

Rectangles can be drawn as just the the outline or filled.

Curved corners can be obtained by setting the `radius` parameter.

To have the effect of multiple widgets using the same decoration (e.g. the curved ends appearing on the first and last widgets) use the `group=True` option.

The advantage of using the `group` option is that the group is dynamic meaning that it is drawn according to the widgets that are currently visible in the group. The group will adjust if a window becomes visible or hidden.

```
decoration_group = {
    "decorations": [
        RectDecoration(colour="#004040", radius=10, filled=True, padding_y=4,␣
↪group=True)
    ],
    "padding": 10,
}

screens = [
    Screen(
        top=Bar(
            [
                extrawidgets.CurrentLayout(**decoration_group),
                widget.Spacer(),
                extrawidgets.StatusNotifier(**decoration_group),
                extrawidgets.Mpris2(**decoration_group),
                extrawidgets.Clock(format="%H:%M:%S", **decoration_group),
                extrawidgets.ScriptExit(
                    default_text="[X]",
                    countdown_format="[{}]",
                    countdown_start=2,
                    **decoration_group
```

(continues on next page)

Fig. 5: path='forward_slash'

Fig. 6: path='back_slash'

Fig. 7: path='zig_zag'

(continued from previous page)

```
                ),
            ]
        ),
        28
    )
]
```

There are two groups in this config: Group 1 is the CurrentLayout widget. Group 2 is the StatusNotifier, Mpris2, Clock and ScriptExit widgets. The groups are separated by the Spacer widget.

When there is no active icon in the StatusNotifier, the bar looks like this:

When there is an icon, the group is expanded to include the widget:

Note the group is not broken despite the Mpris2 widget having no contents.

Groups are determined by looking for:

- widgets using the identical configuration for the decoration
- widgets in a consecutive groups

Groups can therefore be broken by changing the configuration of the group (e.g. by adding an additional parameter such as `group_id=1`) or having an undecorated separator between groups.

Setting `clip=True` will result in the widget's contents being restricted to the area covered by the decoration. This may be desirable for widgets like `ALSAWidget` and `BrightnessControl` which draw their levels in the bar.

Fig. 8: path=[(0, 0), (0.5, 0), (0.5, 0.25), (1, 0.25), (1, 0.75), (0.5, 0.75), (0.5, 1), (0, 1)]

Fig. 9: Single decoration

Fig. 10: Two decorations stacked

CHAPTER

NINE

MASKED IMAGES

9.1 ImgMask

class qtile_extras.images.**ImgMask**(**args*, *drawer: Drawer | None = None*, ***kwargs*)

Image object that uses the image source as a mask to paint the background.

Colour can be set at the moment of drawing, rather than preparing images in advance.

9.2 Loader

class qtile_extras.images.**Loader**(**directories*, *masked=True*, ***kwargs*)

Same as libqtile.images.Loader but takes an optional parameter, `masked`, to determine whether to use `ImgMask` class.

CHAPTER

TEN

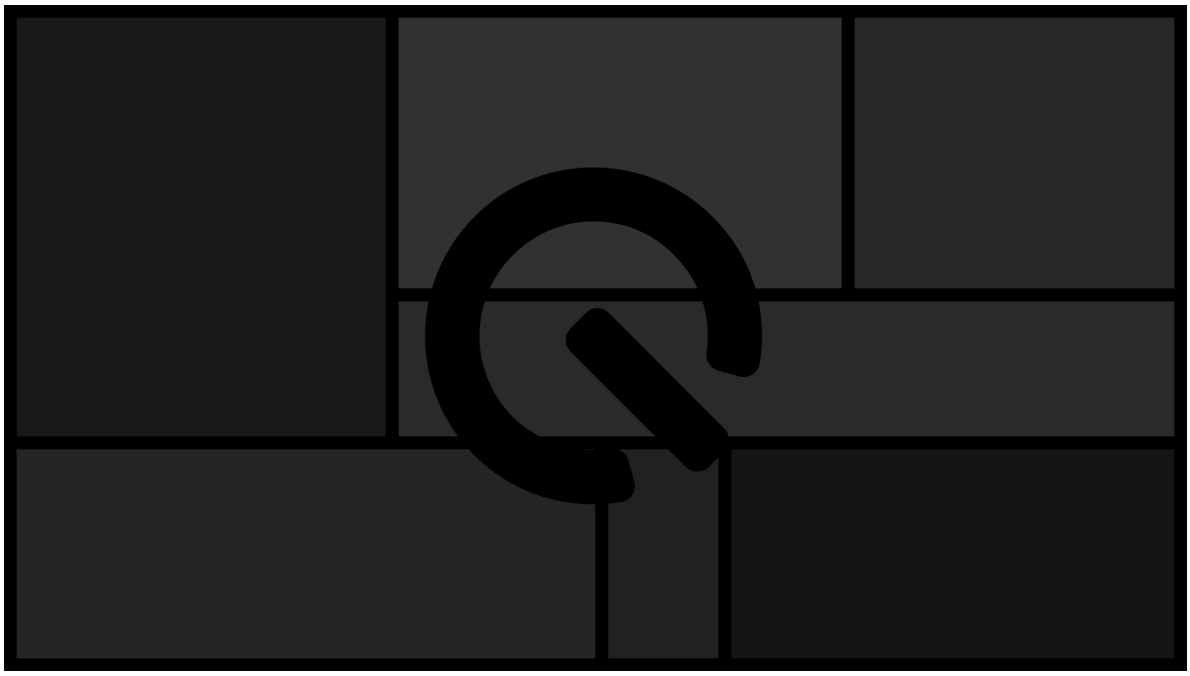
WALLPAPERS

At one point, we thought about shipping Qtile with a default wallpaper as that would be a bit more welcoming than the current black screen. The PR met with mixed reactions so I'll put my "artwork" here instead.

These can be added to your config by doing:

```
from qtile_extras.resources import wallpapers

...

screens = [
    Screen(
        top=Bar(...),
        wallpaper=wallpapers.WALLPAPER_TRIANGLES,
        wallpaper_mode="fill"
    )
]
```

WALLPAPER_TILES



WALLPAPER_TRIANGLES

WALLPAPER_TRIANGLES_ROUNDED

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

G

I

L

P

R

S

T

U

V

W