carmanaught / **mpvcontextmenu** Archived

Context Menu for mpv

MIT License

**7** stars **1** fork

Star Notifications

Code Issues Pull requests Actions Security Insights

master Go to file

**carmanaught** Update README.md ··· on 4 Jun 2018 **26**

View code

**README.md**

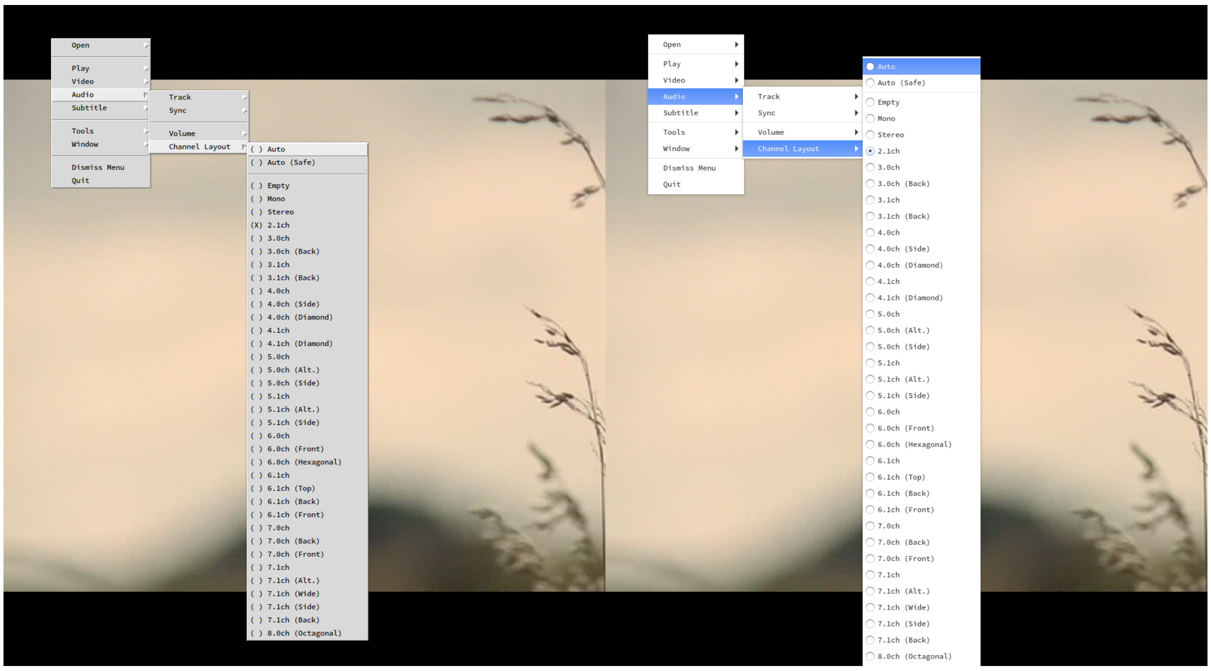
DEPRECATED: This repository has been deprecated and moved to https://gitlab.com/carmanaught/mpvcontextmenu/

# Context Menu for mpv

This is a context menu forked and fairly extensively modified from this one (credit to avih).

## Table of Contents

This is an example of what the Tk (left) and Gtk+ (right) menus look like in use (showing the audio Channel Layout sub-menu):



A lot of the code from the original menu has been rewritten. In particular, the Tk menu adds sub-menu's using the Tcl menu 'cascade' command, retaining the possibility to rebuild the menu along with sub-menu's through the use of the Tcl 'postcascade' command.

The menu layout is based on the right-click menu for Bomi, which is what I was using before switching to mpv. If you were a Bomi user be aware that not all the menu items for Bomi are implemented (particularly those around video settings) and there is no current plan to implement them at this point.

Some of the menu items reference commands that use the functions/bindings in `gui-dialogs.lua` to show dialogs. These are based on the KDialog-open-files script (credit to ntasos) and include both `zenity` and `kdialog` options.

The menu layout definitions and the menu engine have been separated. Part of this is to allow for the use of other menu builders, with the logic and interactions with builders handled by the menu engine script. This has the advantage that the menu engine script doesn't care about the menu definition file and actually allows multiple menu definition files to be used if desired, ensuring they are configured correctly (see Customization below).

While some of this may work on Windows or macOS, most of this is only tested on Linux. For macOS, some of the requirements may be available via Homebrew, though I can't provide any guarantees for things working.

## Requirements

For the menu's to work, this currently requires the following:

- **tcl**, **tk** for the Tk based menu,
- **gjs** (and it's dependancies, likely including **gtk3**) for the Gtk+ based menu,
- **kdialog** or **zenity** for the dialogs to open files/folders/URLs.

Place the files (except for `input.conf` ) in your `~/.config/mpv/scripts/` or `~/.mpv/scripts/` folder. The `input.conf` is not strictly necessary, but the key-bindings shown in the menu are based on those in the `input.conf` . **Note:** the key-bindings are not automatically detected and have been manually added as text to the menu, so you'll need to change/remove them if they don't match your own.

### Tk

You will need to install Tcl and Tk (for the Tk menu) and ensure that the `interpreter["tk"]` variable in `menu-engine.lua` is set properly. This can be set to `wish` or `tclsh` (set to `wish` by default, though `tclsh` may work smoother) and should either be accessible via the PATH environment variable or the full path should be specified.

For Windows, download your preferred Tcl package (check the Tcl software page or perhaps ActiveTcl or Tcl3D) and install/extract as needed. Ensure that the relevant wish.exe or tclsh.exe is either in the PATH, the full path has been specified, or put the necessary executable into the the mpv.exe directory. You could also use tclsh/wish from git/msys2 (mingw).

### Gtk+

The Gtk+ menu uses `gjs` for the interpreter and will need that installed along with whatever dependencies that requires. The `interpreter["gtk"]` variable in `menu-engine.lua` should be set properly as needed, depending on whether it's accessible by from the PATH environment variable, etc.

**Note:** The Gtk+ menu may be slightly slower to open than the Tk menu, which might just generally be due to how I have written the code and the manner in which the menu is built.

This hasn't been tested on Windows as I'm unsure if there is a workable port/version of gjs for Windows. For macOS, it looks like it's available in Homebrew (Gjs), though it doesn't specify dependencies, so you may need Gtk+ from there as well.

## Dialogs

For the dialogs to work, either `kdialog` or `zenity` need to be installed and accessible via the PATH or the path set up manually in `gui-dialogs.lua`. By default, neither the `kdialog` or `zenity` dialogs are enabled and either the script file should be modified or a `gui-dialogs.conf` file should be created (read configuration section below regarding the right directory for this).

If you do not wish to use any of the included dialogs you can also remove the entries in the menu items referencing them.

For `zenity`, apparently there is a port of Zenity for windows, but this has not been tested. For macOS, apparently this is available via Homebrew (Zenity). As far as I'm aware, `kdialog` is not readily available on windows or macOS.

## Fonts

The menu uses the Source Code Pro font, which can be found here (or check your repositories), however the font can be changed in the `menu-builder-tk.tcl` or `menu-builder-gtk-js` files. A mono-spaced font must be used for the menu items to appear correctly.

### Tk

To change the font for the Tk menu and to ensure that the correct name of the font is specified, run `wish` from a terminal, then from the wish prompt, enter `puts [font families]`.

```
user@hostname:~$ wish
% puts [font families]
```

This should output a list of available fonts enclosed by curly braces, which can be used to copy the name of the desired font. To exit, type `exit`.

```
% exit
```

Set the font to be used in `menu-builder-tk.tcl`, changing the line with `{Source Code Pro}` below in the Tcl file to whichever font is preferred and adjusting size as desired.

```
font create defFont -family {Source Code Pro} -size 9
```

#### Gtk+

The Gtk+ menu uses CSS to specify the font, so you should be able to use the font name as it appears to most other applications. Change the font near the end of the file in the `show_menu()` function as part of the `Gtk.CssProvider().load_from_file()` text string.

```
load_from_data(" * { font-family: Source Code Pro; font-size: 9pt; font-weight: 500; }")
```

### Scripts

Additionally the context menu uses some mpv scripts that other people have written, called via script-binding/script-message. These are currently:

- subit for the "Find Subtitle (Subit)" item under "Tools"
- playlistmanager for the "Show" item under "Tools > Playlist"
- stats for the "Playback Information" item under "Tools"

These will need to be either downloaded and set-up for these menu items to work or the commands to use them should be removed if not.

## Usage

There is no default binding for the context menu and the choice of menu needs to be specified via a binding in `input.conf` via `<BINDING> script_message <MENUTYPE>` replacing the `<BINDING>` with the desired key-binding. The `<MENUTYPE>` options are:

- `mpv_context_menu_tk` for the Tk menu,
- `mpv_context_menu_gtk` for the Gtk+ menu.

To get the Tk menu to work with the right-click button of the mouse, for instance, you would use:

```
MOUSE_BTN2 script_message mpv_context_menu_tk
```

The default bindings for the dialogs in `gui-dialogs.lua` are:

| Key | Action |
|---|---|
| Ctrl + F | Open files, replacing the playlist |
| Ctrl + G | Open a folder, replacing the playlist |
| Ctrl + Shift + F | Add/append files to the playlist |
| Ctrl + Shift + G | Add/append a folder to the playlist |
| Shift + F | Open a subtitle file (for the playing file) |

## Configuration

Both `gui-dialogs.lua` and `mpvcontextmenu.lua` will read options stored in respective config files (in a `lua-settings` directory for mpv versions below 0.29.X and a `script-opts` directory for mpv versions from 0.29.X on) in your mpv config directory (mentioned above). The files should be named the same as the Lua files but with `.conf` instead of `.lua` on the end. Check the near the top of each of the files to see which settings can be changed.

For instance, if you want to specify the use of `zenity` dialogs and change the key-bindings for the dialogs, you could create `gui-dialogs.conf` and specify the `dialogPref` and the shortcuts as such:

```
# Dialog preference
dialogPref=zenity
# Open files and open folder
addFiles=Ctrl+f
addFolder=Ctrl+g
```

Keep in mind that the shortcut style should match the `input.conf` format that mpv uses (allowing for case sensitivity) and that there should be no space between the `=` and the value after the `=` should not use quote marks.

For the context menu itself, the options listed in the top of the file specify the unit for the values used. It's important to specify the options in a conf file keeping the same unit values in mind (e.g. Audio Sync is set up for milliseconds but Seek is set up for seconds). An example for `mpvcontextmenu.conf` might be:

```
# Play > Seek - Seconds
seekSmall=10
seekMedium=60
seekLarge=600
# Audio > Volume - Percentage
```

```
audVol=1
```

Note that the changes to the values here only affect the menu, not the values for shortcuts in the `input.conf` which has to be edited separately.

## Customization

For those wishing to change the menu items or add/remove menu items, the following should help, though looking at the code directly will be best. It's best if you know some Lua to make changes here.

### Menu Layout

The menu layouts use what Lua calls tables, though you could also think of them as arrays (I certainly do).

The layout for the menus are sets of tables nested inside an over-arching table (currently called menuList).

When the pseudo-gui is in use and there is no file playing, the layout uses a select number of relevant options from the "while playing" menu allowing for a slightly different menu.

For files that are playing, the layout is wrapped inside a function that is triggered by the "file-loaded" event in mpv (registered with `mp.register_event` ). This is important as some of the mpv property values like track-list items and chapter information, etc. are not available until the file has been loaded.

For both, menus, the layout is inside a table:

```
menuList = {
  -- Menu items go here
}
```

Each menu item is itself a table.

A separator is comprised of a single item and uses the variable SEP to indicate this like so:

```
{SEP},
```

A full menu item table is comprised of at least six (6) or seven (7) items. The layout looks something like this (for all seven items):

```
{Item Type, Label, Accelerator, Command, Item State, Item Disable, Menu Rebuild},
```

Using the above as a guide, this table provides some information for what can/should be entered for each item.

| Value | Purpose |
|---|---|
| Item Type | The item type specifies whether this is a a cascade (sub-menu), command, checkbox, radio item, etc. |
| Label | The label for the menu item and is what will be seen when using the menu. |
| Accelerator | The shortcut for the menu item and will show to the right of the label when using the menu. |
| Command | The command you want to be executed when an item is clicked on. |
| Item State | For checkboxes and radio items, this is the selected/unselected state. |
| Item Disable | This is set to true or false depending on whether the menu item should be clickable/usable. |
| Menu Rebuild | Set this to true if the menu should be rebuilt and re-cascaded so that it appears in the same place with the same sub-menus open. |

For **Item Type**, this should be one of the preset variable names, CASCADE, COMMAND, CHECK and RADIO respectively.

The **Label** and **Accelerator** items become the actual text that shows for each menu item.

**Command** is either the a command to be sent directly to mpv (via `mp.command`) or a function call that will do something (better detailed further below).

The **Item State** should normally be `true` or `false` values, though there is a special case for the A/B Repeat where the receiving script will handle "a", "b" or "off" values and change the appearance of the "checkbox" based on the value.

The best way to handle this for checkboxes or radio items is to wrap a function that will return a true/false for the respective menu item in a function call (detailed below).

The **Item Disable** item is used to enable/disable menu items and can also disable cascades from functioning. This can be useful if a menu item should be disabled when certain functionality is not available with a given file. Set this to `true` to disable the item and `false` to leave it enabled.

**Menu Rebuild** is used to toggle/use a menu item, but keep the menu open to the same sub-menu. This makes it possible to have the mouse over a menu item and click the item, causing a brief flicker as the menu is essentially closed and reopened to where it was.

For the **Item Label**, **Accelerator** and **Command** items, you can use string concatenation `("Text " .. var .. " Text")` as the value is evaluated at much the same time as in-line functions returning values (detailed below).

At least six items should be entered and empty quotes used when not specifying a value. A seventh can be used when using the **Menu Rebuild** functionality.

An example item with these in mind, could be like so:

```
{Command, "Toggle Fullscreen", "F", "cycle fullscreen", "", false},
```

With all that in mind, a basic menu might look something like this (it's important to remember your commas!):

```
menuList = {
  context_menu = {
    {CASCADE, "Play", "play_menu", "", "", false},
  },

  play_menu = {
    {COMMAND, "Play/Pause", "Space", "cycle pause", "", false, true},
    {COMMAND, "Stop", "Ctrl+Space", "stop", "", false},
    {SEP},
    {COMMAND, "Previous", "<", "playlist-prev", "", false},
    {COMMAND, "Next", ">", "playlist-next", "", false},
  },
}
```

The **CASCADE** item is special in that the third value in the table for a cascade menu item is the name of the table that should cascade from that menu item.

In the above example, the cascade will have the `play_menu` table as a sub-menu coming off of it. There is a maximum of 6 total menu levels, including the base, making for up to 5 sub-menu levels deep. Any more than that will throw an error, which also prevents infinite recursion in the menu building function.

### Function call/in-line function

Apart from the separator, the menu items can make use of a function call:

```
function() return somefunction() end
```

Apart from the **Command** item, this function call is evaluated each time the menu is built (each time the script-message shortcut binding is activated). This allows you to dynamically build labels or set values at the time of creating the menu.

In the instance of the **Command** item, if there is a function call, this will be evaluated as a function only upon being clicked and will **not** send a command unless you do so within your function. If there is not a function call, the command provided will be sent to mpv.

There is also a difference between a function call like the above and an in-line function returning a value as part of building a command to send to mpv. For instance, if we had a menu item like this:

```
{Command, getLabel(), "Shift+L", "", "", getState()}
```

When the Lua code is being executed (either on initially being loaded or after a file is loaded), the getLabel() and getState() functions would be run and should return a label string and a true/false value respectively.

## Notes

One of the files included is the `langcodes.lua` which holds two tables filled with associative arrays using the 2 or 3 character long language name representations (ISO 639-1 and ISO 639-2) as property accessors to get full length language names, though these are only in English at this point.

## Disclaimer

I have tried to test this on a variety of media files and have attempted to deal with any bugs that have arisen, but I can't guarantee that this is bug-free. There may be use-cases I haven't considered or some functions may throw errors from unexpected values/input.

I also have only tested this on Linux and any information about Windows or macOS is based either on the original authors code comments or some searches I've done. I may look at trying to test other OS's, but can't guarantee anything at the moment.

## Credits

Thanks go out to the following people:

- avih for the original Tcl/Tk context menu upon which this menu has been built
- ntasos for some code and ideas from the KDialog-open-files script

### Packages

No packages published

### Contributors 2

**carmanaught** Thomas Carmichael

**avih**

### Languages

● **Lua** 87.5% ● **Tcl** 12.5%