XMonad.Hooks.StatusBar

Contents

xmonad calls the logHook with every internal state update, which is useful for (among other things) outputting status information to an external status bar program such as xmobar or dzen.

This module provides a composable interface for (re)starting these status bars and logging to them, either using pipes or X properties. There's also XMonad.Hooks.StatusBar.PP which provides an abstraction and some utilities for customization what is logged to a status bar. Together, these are a modern replacement for XMonad.Hooks.DynamicLog, which is now just a compatibility wrapper.

Usage

You can use this module with the following in your `~/.xmonad/xmonad.hs`:

```
import XMonad
import XMonad.Hooks.StatusBar
import XMonad.Hooks.StatusBar.PP
```

The easiest way to use this module with xmobar, as well as any other status bar that supports property logging, is to use `statusBarProp` with `withEasySB`; these take care of the necessary plumbing:

```
mySB = statusBarProp "xmobar" (pure xmobarPP)
main = xmonad $ withEasySB mySB defToggleStrutsKey def
```

You can read more about X11 properties here or here, although you don't have to understand them in order to use the functions mentioned above.

Most users will, however, want to customize the logging and integrate it into their existing custom xmonad configuration. The `withSB` function is more appropriate in this case: it doesn't touch your keybindings, layout modifiers, or event hooks; instead, you're expected to configure XMonad.Hooks.ManageDocks yourself. Here's what that might look like:

```
mySB = statusBarProp "xmobar" (pure myPP)
main = xmonad . withSB mySB . ewmh . docks $ def {...}
```

You then have to tell your status bar to read from the `_XMONAD_LOG` property of the root window. In the case of xmobar, this is achieved by simply using the `XMonadLog` plugin instead of `StdinReader` in your `.xmobarrc`:

```
Config { ...
       , commands = [ Run XMonadLog, ... ]
       , template = "%XMonadLog% }{ ..."
       }
```

If you don't have an `.xmobarrc`, create it; the `XMonadLog` plugin is not part of the default xmobar configuration and your status bar will not show workspace information otherwise!

With `statusBarProp`, you need to use property logging. Make sure the status bar you use supports reading a property string from the root window, or use some kind of wrapper that reads the property and pipes it into the bar (e.g. `xmonadpropread | dzen2`, see `scripts/xmonadpropread.hs`). The default property is `_XMONAD_LOG`, which is conveniently saved in `xmonadDefProp`. You can use another property by using the function `statusBarPropTo`.

If your status bar does not support property-based logging, you may also try `statusBarPipe`. It can be used in the same way as `statusBarProp` above (for xmobar, you now have to use the `StdinReader` plugin in your `.xmobarrc`). Instead of writing to a property, this function opens a pipe and makes the given status bar read from that pipe. Please be aware that this kind of setup is very bug-prone and hence is discouraged: if anything goes wrong with the bar, xmonad will freeze!

Also note that `statusBarPipe` returns 'IO StatusBarConfig', so you need to evaluate it before passing it to `withSB` or `withEasySB`:

```
main = do
  mySB <- statusBarPipe "xmobar" (pure myPP)
  xmonad $ withSB mySB myConf
```

`data` **StatusBarConfig** # Source

This datatype abstracts a status bar to provide a common interface functions like `statusBarPipe` or `statusBarProp`. Once defined, a status bar can be incorporated in `XConfig` by using `withSB` or `withEasySB`, which take care of the necessary plumbing.

Constructors

StatusBarConfig

sbLogHook :: X ()	What and how to log to the status bar.
sbStartupHook :: X ()	How to start the status bar.
sbCleanupHook :: X ()	How to kill the status bar.

Instances

▷ `Semigroup StatusBarConfig` # Source

▷ `Monoid StatusBarConfig` # Source

▷ `Default StatusBarConfig` # Source — Per default, all the hooks do nothing.

withSB # Source

```
:: LayoutClass l Window
=> StatusBarConfig          The status bar config
-> XConfig l                The base config
-> XConfig l
```

Incorporates a `StatusBarConfig` into an `XConfig` by taking care of the necessary plumbing (starting, restarting and logging to it).

Using this function multiple times to combine status bars may result in only one status bar working properly. See the section on using multiple status bars for more details.

withEasySB # Source

```
:: LayoutClass l Window
=> StatusBarConfig                          The status bar config
-> (XConfig Layout -> (KeyMask, KeySym))    The key binding
-> XConfig l                                The base config
-> XConfig (ModifiedLayout AvoidStruts l)
```

Like `withSB`, but takes an extra key to toggle struts. It also applies the `avoidStruts` layout modifier and the `docks` combinator.

Using this function multiple times to combine status bars may result in only one status bar working properly. See the section on using multiple status bars for more details.

defToggleStrutsKey `:: XConfig t -> (KeyMask, KeySym)` # Source

Default `mod-b` key binding for `withEasySB`

Available Configs

statusBarProp # Source

```
:: String           The command line to launch the status bar
-> X PP             The pretty printing options
-> StatusBarConfig
```

Creates a `StatusBarConfig` that uses property logging to `_XMONAD_LOG`, which is set in `xmonadDefProp`

statusBarPropTo # Source

```
:: String           Property to write the string to
-> String           The command line to launch the status bar
-> X PP             The pretty printing options
-> StatusBarConfig
```

Like `statusBarProp`, but lets you define the property

statusBarPipe # Source

```
:: String                The command line to launch the status bar
-> X PP                  The pretty printing options
-> IO StatusBarConfig
```

Like `statusBarProp`, but uses pipe-based logging instead.

Multiple Status Bars

`StatusBarConfig` is a `Monoid`, which means that multiple status bars can be combined together using `<>` or `mconcat` and passed to `withSB`.

Here's an example of what such declarative configuration of multiple status bars may look like:

```
-- Make sure to setup the xmobar config accordingly
xmobarTop    = statusBarPropTo "_XMONAD_LOG_1" "xmobar -x 0 ~/.config/xmobar/xmobarrc_top"    (pure
xmobarBottom = statusBarPropTo "_XMONAD_LOG_2" "xmobar -x 0 ~/.config/xmobar/xmobarrc_bottom" (pure
xmobar1      = statusBarPropTo "_XMONAD_LOG_3" "xmobar -x 1 ~/.config/xmobar/xmobarrc1"       (pure

main = xmonad $ withSB (xmobarTop <> xmobarBottom <> xmobar1) myConfig
```

The above example also works if the different status bars support different logging methods: you could mix property logging and logging via pipes. One thing to keep in mind is that if multiple bars read from the same property, their content will be the same. If you want to use property-based logging with multiple bars, they should read from different properties.

Long-time xmonad users will note that the above config is equivalent to the following less robust and more verbose configuration that they might find in their old configs:

```
main = do
  -- do not use this, this is an example of a deprecated config
  xmproc0 <- spawnPipe "xmobar -x 0 ~/.config/xmobar/xmobarrc_top"
  xmproc1 <- spawnPipe "xmobar -x 0 ~/.config/xmobar/xmobarrc_bottom"
  xmproc2 <- spawnPipe "xmobar -x 1 ~/.config/xmobar/xmobarrc1"
  xmonad $ def {
    ...
    , logHook = dynamicLogWithPP ppTop { ppOutput = hPutStrLn xmproc0 }
             >> dynamicLogWithPP ppBottom { ppOutput = hPutStrLn xmproc1 }
             >> dynamicLogWithPP pp1 { ppOutput = hPutStrLn xmproc2 }
    ...
  }
```

By using the new interface, the config becomes more declarative and there's less room for errors.

Property Logging utilities

xmonadPropLog :: String -> X () # Source

Write a string to the `_XMONAD_LOG` property on the root window.

xmonadPropLog' # Source

```
:: String   Property name
-> String   Message to be written to the property
-> X ()
```

Write a string to a property on the root window. This property is of type `UTF8_STRING`. The string must have been processed by `encodeString` (`dynamicLogString` does this).

xmonadDefProp `:: String` # Source

The default property xmonad writes to. (`_XMONAD_LOG`).

Managing Status Bar Processes

spawnStatusBarAndRemember # Source

```
:: String   The command used to spawn the status bar
-> X ()
```

Spawns a status bar and saves its PID. This is useful when the status bars should be restarted with xmonad. Use this in combination with `cleanupStatusBars`.

Note: in some systems, multiple processes might start, even though one command is provided. This means the first PID, of the group leader, is saved.

cleanupStatusBars `:: X ()` # Source

Kills the status bars started with `spawnStatusBarAndRemember`, and resets the state. This could go for example at the beginning of the startupHook.

Concretely, this function sends a `sigTERM` to the saved PIDs using `signalProcessGroup` to effectively terminate all processes, regardless of how many were started by using `spawnStatusBarAndRemember`.

There is one caveat to keep in mind: to keep the implementation simple; no checks are executed before terminating the processes. This means: if the started process dies for some reason, and enough time passes for the PIDs to wrap around, this function might terminate another process that happens to have the same PID. However, this isn't a typical usage scenario.

Manual Plumbing

If you do not want to use any of the "batteries included" functions above, you can also add all of the necessary plumbing yourself (the source of `withSB` might come in handy here).

`xmonadPropLog` allows you to write a string to the `_XMONAD_LOG` property of the root window. Together with `dynamicLogString`, you can now simply set your `logHook` to the appropriate function; for instance:

```
main = xmonad $ def {
   ...
   , logHook = xmonadPropLog =<< dynamicLogString myPP
   ...
   }
```

If you want to define your own property name, use `xmonadPropLog'` instead of `xmonadPropLog`.

If you just want to use the default pretty-printing format, you can replace `myPP` with `def` in the above `logHook`.

Note that setting `logHook` only sets up xmonad's output; you are responsible for starting your own status bar program and making sure it reads from the property that xmonad writes to. To start your bar, simply put it into your `startupHook`. You will also have also have to add `docks` and `avoidStruts` to your config. Putting all of this together would look something like

```
import XMonad.Util.SpawnOnce (spawnOnce)
import XMonad.Hooks.ManageDocks (avoidStruts, docks)

main = do
    xmonad $ docks $ def {
      ...
      , logHook     = xmonadPropLog =<< dynamicLogString myPP
      , startupHook = spawnOnce "xmobar"
      , layoutHook  = avoidStruts myLayout
      ...
      }
myPP = def { ... }
myLayout = ...
```

If you want a keybinding to toggle your bar, you will also need to add this to the rest of your keybindings.

The above has the problem that xmobar will not get restarted whenever you restart xmonad (`spawnOnce` will simply prevent your chosen status bar from spawning again). Using `statusBarProp`, however, takes care of the necessary plumbing *and* keeps track of the started status bars, so they can be correctly restarted with xmonad. This is achieved using `spawnStatusBarAndRemember` to start them and `cleanupStatusBars` to kill previously started bars.

Even if you don't use a status bar, you can still use `dynamicLogString` to show on-screen notifications in response to some events. For example, to show the current layout when it changes, you could make a keybinding to cycle the layout and display the current status:

```
((mod1Mask, xK_a), sendMessage NextLayout >> (dynamicLogString myPP >>= \d -> spawn $ "xmessage " ++
```

If you use a status bar that does not support reading from a property (like dzen), and you don't want to use the `statusBar` function, you can, again, also manually add all of the required components, like this:

```
import XMonad.Util.Run (hPutStrLn, spawnPipe)

main = do
    h <- spawnPipe "dzen2 -options -foo -bar"
    xmonad $ def {
      ...
      , logHook = dynamicLogWithPP $ def { ppOutput = hPutStrLn h }
      ...
      }
```

In the above, note that if you use `spawnPipe` you need to redefine the `ppOutput` field of your pretty-printer; by default the status will be printed to stdout rather than the pipe you create. This was meant to be used together with running xmonad piped to a status bar like so: `xmonad | dzen2`, and is what the old `dynamicLog` assumes, but it isn't recommended in modern setups. Applications launched from xmonad inherit its stdout and stderr, and will print their own garbage to the status bar.