Kahakai Documentation

Nick Welch

Kahakai Documentation
by Nick Welch

Table of Contents

1. Introduction **1**
1.1. Synopsis 1
1.2. History 1
1.3. Features 1
2. Configuration Files **2**
3. `userconfig` **3**
3.1. **`import`**s 3
3.2. Macros 3
3.3. The anatomy of bindings 3
3.3.1. Events 3
3.3.2. Actions 6
3.4. **`MyConfiguration`** 8

List of Examples

3-1. Example uses of keynames 5

Chapter 1. Introduction

1.1. Synopsis

Kahakai is a progressive, moderately lightweight window manager focused on extreme customization, good looks, and well thought-out design.

1.2. History

Forked from the then-stagnating (and now seemingly dead) Waimea (http://sf.net/projects/waimea) window manager, Kahakai sprang to life in late May of 2003. A fair number of members of the #waimea IRC channel on Freenode (http://freenode.net) were tired of waiting around, and decided that it was feasible to take Waimea to bigger and better places. The main feature in mind was integrated scriptability -- not in one language, but many. This was seen as the natural evolution of Waimea's already powerful *actions* system. SWIG (http://www.swig.org) was chosen for its wide range of supported languages, and work began initally on Python. As of this writing, Ruby support is mostly complete, with some small kinks left to work out and refine.

Naturally, many other things began to change, be removed, added, and reworked. Kahakai is now very much a huge leap beyond what Waimea was, both in scripting abilities, and in all other aspects. Development continues, and at roughly 4.5 months old (at time of writing, of course :), Kahakai still has a lot of room to grow.

1.3. Features

- Python scripting for configuration
- Preliminary Ruby support
- Window/edge snapping (on move *and* resize)
- Transparent window titles and menus with anti-aliased fonts
- Dockapp support
- Pixmap styles, compatible with Blackbox/Fluxbox/Openbox & Waimea styles
- Viewports, multiple desktops
- KDE and GNOME support

Chapter 2. Configuration Files

Your Kahakai configuration files will all reside inside of your `$HOME/.kahakai` directory. If this directory does not exist upon startup, Kahakai will create it and populate it with the default configuration files.

The files in `$PREFIX/share/kahakai` that begin with "home_" are the same ones that are automatically copied to `$HOME/.kahakai` the first time you run Kahakai. When put into your home directory, the "home_" prefix is removed.

> **Note:** One thing to note is that development of Kahakai can happen at a fairly rapid pace, and some releases will require modification of your old configuration files. Generally there will be a file describing any such changes within the `doc/` subdirectory of the Kahakai source. For example, in 0.5, some things in `userconfig.py` changed, and accordingly, there is a file describing the changes in `doc/userconfig-migration-0.5`.

Chapter 3. `userconfig`

`userconfig` is your personal configuration file for all things concerning Python in Kahakai, which basically boils down to key and mouse bindings. At the time being, you'll almost definitely be using `userconfig.py`, but as Ruby support matures (and Ruby users spring up), you may very well be using `userconfig.rb`, or even another language in the future. For now the docs mostly focus on Python.

3.1. `imports`

The first thing done in `userconfig.py` (and generally all Python scripts) is importing of other modules. A Python module is just a file containing Python code; you import it by using the **`import`** Python keyword. The name of the module is the same as the name of the file, with the `.py` chopped off the end. You can learn more about how python modules work in the Python tutorial (http://www.python.org/doc/current/tut/node8.html).

So now let's take a look at what we're importing.

```
from actions import install
from macros import *
```

What this means is that we are importing the function **`install`** from `actions.py`, and importing everything from `macros.py`. Using the **`from modulename import foo`** syntax imports everything from *modulename* into the global namespace of the current file, with the exception of any names starting with an underscore. Thus, things like **`Ctrl`**, **`KeyPress()`**, and **`screen()`** all come from `macros.py`, and are available to us in `userconfig.py`.

3.2. Macros

There are many small functions which set up specific sets of key and mouse bindings for you; we call these functions *macros*. These are generally called near the top of `userconfig.py`, right after your import statements. The examples and comments contained in `userconfig.py` should do a good job of explaining it all, so have a look through it, and if you're even more curious, you can look through `macros.py`.

3.3. The anatomy of bindings

A binding is basically a pairing of two things: an *event*, and an *action*. Together, they form a coherent instruction to the window manager, along the lines of: *"when this happens, do that."*

3.3.1. Events

The *event* is generally triggered by something outside of the window manager, usually the mouse or keyboard. Some examples are **`DoubleClick`** (the familiar double-clicking of a mouse button), **`KeyPress`** (the pressing of a key on the keyboard), and **`EnterNotify`** (when the mouse pointer enters a window's area).

An event generally excepts one or two arguments. If the event involves a mouse or key press, then it will accept two arguments: the modifier, and the key/mouse button. If an event does not involve a key/mouse (e.g. **`EnterNotify`**), then it accepts only one argument; the modifier. In the former case, the modifier comes *before* the key/mouse button, and in either case, the modifier may be omitted. To use more than one modifier, simply add them together. To specifically require a modifier to *not* be part of a binding, subtract it. Here are some examples of valid events, in Python:

- Pressing Control, Alt, and A at the same time:

```
KeyPress(Ctrl+Alt, "A")
```

- Clicking the left mouse button:

```
ButtonPress("Button1")
```

- Scrolling the mouse wheel down twice quickly, while holding shift and *not* holding Super (windows key):

```
DoubleClick(-Super+Shift, "Button5")
```

- Entering a window with the mouse pointer, while holding Alt:

```
EnterNotify(Alt)
```

- Simply leaving a window with the mouse pointer:

```
LeaveNotify()
```

- Pressing Control and the keypad "plus" key at the same time, and *not* pressing Shift, Alt, or Super:

```
KeyPress(Ctrl-Shift-Alt-Super, "KP_Add")
```

3.3.1.1. Keys

Keyboard keys are specified in a string, using the standard X name for that key. To find the name of a key, use the **`xev(1)`** utility that comes with XFree86. Just start up **`xev`**, start typing in keys, and it'll tell you all sorts of information about them.

Key names are about as straightforward as you'd expect; most keys go by the name that you call them. The backspace key is **`BackSpace`**, the tab key is **`Tab`**, the letter "a" is **`a`**, and the number one is **`1`**. Some keys, such as keypad keys, and "multimedia" keys, will have somewhat less predictable names. Note that key names are *case-insensitive*!

Alternatively, you can use keycodes for keys that don't have a name. Many times, you may have a multimedia keyboard where some or all of the multimedia keys do not have names -- but every key has a keycode (well, as long as X actually realizes the key is there :). To specify a keycode, just enter the keycode itself where the key name goes, *without* quotes (i.e. as an integer, not a string). The keycode is obtained the same way as the key name: with **`xev`**.

Example 3-1. Example uses of keynames

```
KeyPress(Alt, "A")             # letter A
KeyPress(Alt, "a")             # same thing
KeyPress(Alt, "KP_Add")        # plus sign on keypad
KeyPress(Alt, "F1")            # F1 key
KeyPress(Alt, "XF86AudioPlay") # multimedia "play" key
KeyPress(Alt, 142)             # whatever key has keycode 142
KeyPress(Alt, "Control_L")     # works!
```

3.3.1.2. Mouse Buttons

Mouse buttons are even simpler, since there are relatively so few of them. The names of the buttons are in the form of **`ButtonX`**, where X is the number of the button. Your left (usually) mouse button is Button1, and your right mouse button is Button3. If you have a wheel, then pressing it should be Button2, scrolling it up should be Button4, and scrolling down should be Button5. Buttons are given in strings, just like key names. Thus, when you actually type in a button name, you would put quotes around it.

3.3.1.3. Modifiers

A modifier is different from a normal key in that it is designed to be held in conjunction with normal keys, to *modify* their meaning. The modifier that you use all of the time is the Shift key. Others include the Control key, the Alt key, and the Super AKA Windows key. These four main modifiers also have shorthand counterparts, **`C`**, **`S`**, **`A`**, and **`W`**, representing Control, Shift, Alt, and Super, respectively.

As previously mentioned, you can use normal addition and subtraction operations to specify arbitrary combinations of modifiers; thus **`-Ctrl+Alt-Super`** is perfectly valid (it would mean "alt and not control and not super"). Unlike key and mouse button names, modifiers are predefined

objects, and you can use them as such, instead of quoting modifier keynames. However -- you can also use the keynames in strings, if you need or want to. **`"Control_L"`** is just as good as **`Ctrl`**, if it's by itself. You can't, however, use the addition and subtraction operators like you would with modifier objects, since you'd just be concatenating/subtracting strings, and it would basically just bomb on you at one point or another, since it wouldn't make any sense. But if you *really* do need to use a string name for a modifier, and want to be able to add/subtract it with other modifiers, there is a way. If you want to add "Control_L" with "BobsModifier_L", you would do **`Modifiers.fromString("Control_L") + Modifiers.fromString("BobsModifier_L")`**. However, that's digging deeper at the scripting interface, so no guarantees that that will always work out of the box (although at present, it should work just fine).

3.3.2. Actions

The *action* is the actual course of action to take when the corresponding event happens. An action can be a few different things:

3.3.2.1. Builtin Actions With No Arguments

This is simply an action in the Kahakai core, that has no need for extra information to be passed to it. This action would simply be a string with the name of the (case sensitive!) action in it. An example would be **`"Close"`** (to close the focused window), or **`"{xterm}"`** (to execute an xterm - shell commands are put in curly braces).

3.3.2.2. Builtin Actions With Arguments

To call a builtin action with an argument, you enclose the two in a sequence (in Python - a tuple or list). The first item is the action name in a string, and the second item is the argument, also in a string. However, if the argument is an integer, it can be a normal integer, not enclosed in quotes. Some examples:

- ("GoToDesktop", 0)
- ("ViewportRelativeMove", "+100+200")

3.3.2.3. Functions within the scripts

You can create your own functions within your scripts, and register them to events, to be called when those events occur. The only difference is that instead of putting the action name into a string, you actually pass a reference to your function. For example, if you created a function called **`growByTwentyPx()`**, and wanted to assign it to Super+G, you would create a binding like this:

```
(KeyPress(Super, "G"), growByTwentyPx)
```

You can also pass arbitrary arguments to your own callback functions, in basically the same way you would do so with builtin actions. Say you had the following function:

```
def printSomething(window, event, action):
print action.param
```

And now you want to Ctrl+H to print "Hello, World!", and Ctrl+F to say "F U!". You would use the following bindings:

```
(KeyPress(Ctrl, "H"), (printSomething, "Hello, World!")),
(KeyPress(Ctrl, "F"), (printSomething, "F U!"))
```

Notice the arguments to **`printSomething()`**; **`window`**, **`event`**, and **`action`**: These arguments are passed to all functions that you call from bindings. The window argument is either the currently focused window, or the window involved in the event. For example, if you assigned a binding to a double click event on the root window, then the **`window`** argument would be the root window. You can do *a lot* of useful things with the window. Some useful examples:

```
def halfScreenWidth(win, ev, act):
  """Sets the window's width to ((screen width / 2) - 1)"""

  # the internal names of the parameters are irrelevant, you
  # could name them poop, pizza, and lightning, if you wanted.
  # in fact, i think you do...

  win.SetFrameWidth((screen.width/2)-1)
  win.RedrawWindow()

def moveToLeft(poop, pizza, lightning):
  """Moves windo^H^H^H^H^Hpoop to left screen edge"""
  poop.SetFrameLeft(0)
  poop.RedrawWindow()

def grow(win, ev, act):
  """Grows a window by 20 pixels in all four directions"""
  win.SetFrameLeft(win.GetFrameLeft()-20)
  win.SetFrameTop(win.GetFrameTop()-20)
  win.SetFrameWidth(win.GetFrameWidth()+40)
  win.SetFrameHeight(win.GetFrameHeight()+40)
  win.RedrawWindow()
```

The **`event`** argument is the event object representing the event which triggered the binding to be called. Generally you won't have much use for this (if you can think of one, let us know :). The **`action`** argument is the action part of the binding used to call your function, and its **`param`** attribute is the argument you (may have) passed to your function. In my examples above, the "Hello, World!" and "F U!" strings are contained in **`action.param`**.

There are some additional things to note about user-defined functions:

- Your functions should not block! If they do, then the entire window manager will be blocked, including redraws and the whole deal. Not purty.

- Threads don't work with Kahakai! Don't try to be clever and spawn a thread to do something, because it basically just won't work. Swig is probably the culprit, but no one has really looked into it. If you can figure any more out regarding threads, please let us know!

3.4. `MyConfiguration`

`MyConfiguration` is a class which contains methods (functions) that return sets of bindings. Each method corresponds to a different screen or window area. For example, **`defaultAllDecor()`** registers bindings for all window decorations, and **`root()`** registers bindings for your root window (AKA *desktop*).

The **`MyConfiguration`** class *itself* contains no such code for registering bindings; it is rather dumb. Instead, you pass an *instance* of MyConfiguration to the **`install`** function (remember him? :). This is done near the bottom of `userconfig.py`.

Some things to note about **`MyConfiguration`**:

- *Individual methods are optional*: You can remove any method(s) you want from **`MyConfiguration`**, they're all optional.
- **`MyConfiguration`** *itself* is optional; you can just delete the entire thing if you want. If you do this, make sure to also delete (or comment out) the line that says **`install(MyConfiguration())`**, *and* replace it with **`finish()`**!

Most of your bindings will probably go into either **`globalKeyBindings()`** (for global bindings), or **`defaultAllWindows()`** (for window bindings). As previously mentioned, all of these methods return sets of bindings. A set in vague terms just means a list or array of things. In Python, it would be either a list or a tuple. The default uses lists, since they are easier for beginners to use (tuples have weird syntax when they only contain zero or one items). A list is just some square brackets, with items inside, delimited by commas. A simple Python list would be [1,2,3,4].

Now, with that in mind, we're going to set up some bindings, using what we learned previously about bindings and events and actions and whatnot. We will set up the following bindings:

- Double-click on root window to bring up an aterm
- Super + M to maximize windows
- Ctrl + Alt + Z to run Mozilla
- Ctrl + Alt + G to run The Gimp

The first item is a root window binding. So what we do is have the **`root()`** method in **`MyConfiguration`** return this binding:

```
def root(self):
  """
  The stuff in these quotes is inline documentation, AKA a "docstring."
  It does not affect your code at all.  You can delete it if it isn't
```

```
  useful to you.
  """
  # this is a comment

  # now here is our double click aterm!
  return [ (DoubleClick("Button1"), "{aterm}") ]
```

Ok, that wasn't bad. Now onto the Super + M mazimize binding. This is a window binding, so it goes into **`defaultAllWindows()`**.

```
def defaultAllWindows(self):
  return [ (KeyPress(Super, "M"), "Mazimize") ]
```

Pretty similar to the last one. Now let's do two at once.

```
def globalKeyBindings(self):
return [

  # Indentation is important in Python, but once we type an opening
  # brace or parentheses, we can pretty much indent stuff however
  # we want.

  (KeyPress(Ctrl+Alt, "Z"), "{mozilla}"),  # If you have more than one binding,
                                           # always remember the commas in between!

  (KeyPress(C+A, "g"), "{gimp}")           # C is an alias for Ctrl, and A is an alias
                                           # for Alt (remember? :). capitalization of th
                                           # key names doesn't matter.  G == g

] # we are free to put the ending bracket pretty much anywhere
```