Zsh Plugin Standard

Table of Contents

What Is A Zsh Plugin?

Historically, Zsh plugins were first defined by Oh My Zsh. They provide for a way to package together files that extend or configure the shell's functionality in a particular way.

At a simple level, a plugin:

1. Has its directory added to `$fpath` (Zsh documentation). This is being done either by a plugin

manager or by the plugin itself (see 5th section for more information).

2. Has its first `*.plugin.zsh` file sourced (or `*.zsh`, `init.zsh`, `*.sh`, these are non-standard).

The first point allows plugins to provide completions and functions that are loaded via Zsh's `autoload` mechanism (a single function per-file).

From a more broad perspective, a plugin consists of:

1. A directory containing various files (main script, autoload functions, completions, Makefiles, backend programs, documentation).
2. A sourcable script that obtains the path to its directory via `$0` (see the next section for a related enhancement proposal).
3. A Github (or other site) repository identified by two components **username/pluginname**.
4. A software package containing any type of command line artifacts – when used with advanced plugin managers that have hooks, can run Makefiles, add directories to `$PATH`.

Below follow proposed enhancements and codifications of the definition of a "Zsh plugin" and the actions of plugin managers – the proposed standardization. They cover the information of how to write a Zsh plugin.

---

1. Standardized `$0` Handling

To get the plugin's location, plugins should do:

```
0="${${ZERO:-${0:#$ZSH_ARGZERO}}:-${(%):-%N}}"
0="${${(M)0:#/*}:-$PWD/$0}"

# Then ${0:h} to get plugin's directory
```

The one-line code above will:

1. Be backwards-compatible with normal `$0` setting and usage.
2. Use `ZERO` if it's not empty,
   - the plugin manager will be easily able to alter effective `$0` before loading a plugin,
   - this allows for e.g. `eval "$(<plugin)"`, which can be faster than `source` (comparison note that it's not for a compiled script).
3. Use `$0` if it doesn't contain the path to the Zsh binary,
   - plugin manager will still be able to set `$0`, although more difficultly (requires `unsetopt function_argzero` before sourcing plugin script, and `0=⋯` assignment),
   - `unsetopt function_argzero` will be detected (it causes `$0` not to contain plugin-script path, but path to Zsh binary, if not overwritten by a `0=⋯` assignment),

    - `setopt posix_argzero` will be detected (as above).
4. Use `%N` prompt expansion flag, which always gives absolute path to script,
    - plugin manager cannot alter this (no advanced loading of plugin is possible), but simple plugin-file sourcing (without a plugin manager) will be saved from breaking caused by the mentioned `*_argzero` options, so this is a very good last-resort fallback.
5. Finally, in the second line, it will ensure that `$0` contains an absolute path by prepending it with `$PWD` if necessary.

The goal is flexibility, with essential motivation to support `eval "$(<plugin)"` and definitely solve `setopt no_function_argzero` and `setopt posix_argzero` cases.

A plugin manager will be even able to convert a plugin to a function (author implemented such proof of concept functionality, it's fully possible – also in an automatic fashion), but performance differences of this are yet unclear. It might however provide a use case.

The last, 5th point also allows to use the `$0` handling in scripts (i.e. runnables with the hashbang `#!⋯`) to get the directory in which the script file resides.

The assignment uses quoting to make it resilient to combination of `GLOB_SUBST` and `GLOB_ASSIGN` options. It's a standard snippet of code, so it has to be always working. When you'll set e.g.: the `zsh` emulation in a function, you in general don't have to quote assignments.

Adoption Status

1. Plugin managers: Zinit, Zpm, Zgen (after and if the PR will be merged)
2. Plugins: GitHub search

2. Functions Directory

Despite that the current-standard plugins have their main directory added to `$fpath`, a more clean approach is being proposed: that the plugins use a subdirectory called `functions` to store their completions and autoload functions. This will allow a much cleaner design of plugins. For example, zdharma/zflai suffers from this issue – it has all of its autoload functions in the main directory of the plugin.

The plugin manager should add such directory to `$fpath`. The lack of support of the current plugin managers can be easily resolved via the indicator:

```
if [[ ${zsh_loaded_plugins[-1]} != */kalc && -z ${fpath[(r)${0:h}/functions]} ]] {
    fpath+=( "${0:h}/functions" )
}
```

or, via use of the `PMSPEC` parameter:

```
if [[ $PMSPEC != *f* ]] {
    fpath+=( "${0:h}/functions" )
}
```

Above snippet added to the `plugin.zsh` file will add the directory to the `$fpath` with the compatibiliy with any new plugin managers preserved.

Adoption Status

1. Plugin managers: Zpm

3. Unload Function

If a plugin is named e.g. `kalc` (and is available via `an-user/kalc` plugin-ID), then it can provide a function, `kalc_plugin_unload`, that can be called by a plugin manager to undo the effects of loading that plugin.

A plugin manager can implement its own tracking of changes made by a plugin so this is in general optional. However, to properly unload e.g. a prompt, dedicated tracking (easy to do for the plugin creator) can provide better, predictable results. Any special, uncommon effects of loading a plugin are possible to undo only by a dedicated function.

However, an interesting compromise approach is available – to withdraw only the special effects of loading a plugin via the dedicated, plugin-provided function and leave the rest to the plugin manager. The value of such approach is that maintaining of such function (if it is to withdraw **all** plugin side-effects) can be a daunting task requiring constant monitoring of it during the plugin develoment process.

Note that the unload function should contain `unfunction $0` (or better `unfunction kalc_plugin_unload` etc., for copatibility with the `*_argzero` options), to also delete the function itself.

Adoption Status

1. One plugin manager, Zinit, implements plugin unloading and calls the function.
2. Multiple plugins:
    - GitHub search,
    - `romkatv/powerlevel10k`, is using the function to execute a specific task: shutdown of the binary, background gitstatus demon, with a very good results,
    - `agkozak/agkozak-zsh-prompt` is using the function to completely unload the prompt,
    - `agkozak/zsh-z` is using the function to completly unload the plugin,
    - `agkozak/zhooks` is using the function to completely unload the plugin.

4. `@zsh-plugin-run-on-unload` Call

The plugin manager can provide a function `@zsh-plugin-run-on-unload` which has the following call syntax:

```
@zsh-plugin-run-on-unload "{code-snippet-1}" "{code-snippet-2}" ⋯
```

The function registers pieces of code to be run by the plugin manager **on unload of the plugin**. The execution of the code should be done by the `eval` builtin in the same order as they are passed to the call.

The code should be executed in the plugin's directory, in the current shell.

The mechanism thus provides another way, side to the unload function, for the plugin to participate in the process of unloading it.

Adoption Status

It's a recent addition to the standard and only one plugin manager, Zinit, implements it.

5. `@zsh-plugin-run-on-update` Call

The plugin manager can provide a function `@zsh-plugin-run-on-update` which has the following call syntax:

```
@zsh-plugin-run-on-update "{code-snippet-1}" "{code-snippet-2}" ⋯
```

The function registers pieces of code to be run by the plugin manager on update of the plugin. The execution of the code should be done by the `eval` builtin in the same order as they are passed to the call.

The code should be executed in the plugin's directory, possibly in a subshell **after downloading any new commits** to the repository.

Adoption Status

It's a recent addition to the standard and only one plugin manager, Zinit, implements it.

6. Plugin Manager Activity Indicator

Plugin managers should set the `$zsh_loaded_plugins` array to contain all previously loaded plugins and the plugin currently being loaded (as the last element). This will allow any plugin to:

1. Check which plugins are already loaded.
2. Check if it is being loaded by a plugin manager (i.e. not just sourced).

The first item allows a plugin to e.g. issue a notice about missing dependencies. Instead of issuing a notice, it may be able to satisfy the dependencies from resources it provides. For example, `pure` prompt provides `zsh-async` dependency library within its source tree, which is normally a separate project. Consequently, the prompt can decide to source its private copy of `zsh-async`, having also reliable `$0` defined by previous section (note: `pure` doesn't normally do this).

The second item allows a plugin to e.g. set up `$fpath`, knowing that plugin manager will not handle this:

```
if [[ ${zsh_loaded_plugins[-1]} != */kalc && -z ${fpath[(r)${0:h}]} ]] {
    fpath+=( "${0:h}" )
}
```

This will allow user to reliably source the plugin without using a plugin manager. The code uses the wrapping braces around variables (i.e.: e.g.: `${fpath⋯}`) to make it compatible with the `KSH_ARRAYS` option and the quoting around `${0:h}` to make it compatible with the `SH_WORD_SPLIT` option.

Adoption Status

1. Plugin managers: Zinit, Zpm, Zgen (after and if the PR will be merged)
2. Plugins: GitHub search

7. Global Parameter With PREFIX For Make, Configure, Etc.

Plugin managers may export the parameter `$ZPFX` which should contain a path to a directory dedicated for user-land software, i.e. for directories `$ZPFX/bin`, `$ZPFX/lib`, `$ZPFX/share`, etc. Suggested name of the directory is `polaris` (e.g.: Zinit uses this name and places this directory at `~/.zinit/polaris` by default).

User can then configure hooks (feature of e.g. zplug and Zinit) to invoke e.g. `make PREFIX=$ZPFX install` at clone & update of the plugin to install software like e.g. tj/git-extras. This is a developing role of Zsh plugin managers as package managers, where `.zshrc` has a similar role to Chef or Puppet configuration and allows to **declare** system state, and have the same state on different accounts / machines.

No-narration facts-list related to `$ZPFX`:

1. `export ZPFX="$HOME/polaris"` (or e.g. `$HOME/.zinit/polaris`)
2. `make PREFIX=$ZPFX install`
3. `./configure --prefix=$ZPFX`
4. `cmake -DCMAKE_INSTALL_PREFIX=$ZPFX .`
5. `zinit ice make"PREFIX=$ZPFX install"`
6. `zplug ⋯ hook-build:"make PREFIX=$PFX install"`

Adoption Status

1. Plugin managers: Zinit, Zpm

8. Global Parameter holding the plugin-manager's capabilities

The above paragraphs of the standard spec each constitute a capability, a feature of the plugin manager. It would make sense that the capabilities are somehow discoverable. To address this, a global parameter called `PMSPEC` (from *plugin-manager specification*) is proposed. It can hold the following latin letters each informing the plugin, that the plugin manager has support for a given feature:

- `0` – the plugin manager provides the `ZERO` parameter,
- `f` - … supports the `functions` subdirectory,
- `u` - … the unload function,
- `U` - … the `@zsh-plugin-run-on-unload` call,
- `p` – … the `@zsh-plugin-run-on-update` call,
- `i` – … the `zsh_loaded_plugins` activity indicator,
- `P` – … the `ZPFX` global parameter,
- `s` – … the `PMSPEC` global parameter itself (i.e.: should be always present).

The contents of the parameter describing a fully-compliant plugin manager should be: `0fuUpiPs`. The plugin can then verify the support by, e.g.:

```
if [[ $PMSPEC != *f* ]] {
    fpath+=( "${0:h}/functions" )
}
```

Adoption Status

1. Plugin managers: Zinit, Zpm

Zsh Plugin-Programming Best Practices

The document is to define a **Zsh-plugin** but also to serve as an information source for plugin creators. Therefore, it covers also a best practices information in this section.

Use Of `add-zsh-hook` To Install Hooks

Zsh ships with a function `add-zsh-hook`. It has the following invocation syntax:

```
add-zsh-hook [ -L | -dD ] [ -Uzk ] hook function
```

The command installs a `function` as one of the supported zsh `hook` entries. which are one of: `chpwd`, `periodic`, `precmd`, `preexec`, `zshaddhistory`, `zshexit`, `zsh_directory_name`. For their meaning refer to the Zsh documentation.

Use Of `add-zle-hook-widget` To Install Zle Hooks

Zle editor is the part of the Zsh that is responsible for receiving the text from the user. It can be said that it's based on widgets, which are nothing more than Zsh functions that are allowed to be ran in Zle context, i.e. from the Zle editor (plus a few minor differences, like e.g.: the `$WIDGET` parameter that's automatically set by the Zle editor).

The syntax of the call is:

```
add-zle-hook-widget [ -L | -dD ] [ -Uzk ] hook widgetname
```

The call resembles the syntax of the `add-zsh-hook` function. The only difference is that it takes a `widgetname`, not a function name, and that the `hook` is being one of: `isearch-exit`, `isearch-update`, `line-pre-redraw`, `line-init`, `line-finish`, `history-line-set`, or `keymap-select`. Their meaning is explained in the Zsh documentation.

The use of this function is recommended because it allows to install **multiple** hooks per each `hook` entry. Before introducing the `add-zle-hook-widget` function the "normal" way to install a hook was to define widget with the name of one of the special widgets. Now, after the function has been introduced in Zsh `5.3` it should be used instead.

Standard Parameter Naming

There's a convention already present in the Zsh world – to name array variables lowercase and scalars uppercase. It's being followed by e.g.: the Zsh manual and the Zshell itself (e.g.: `REPLY` scalar and `reply` array, etc.). The requirement for the scalars to be uppercase should be, in my opinion, kept only for the global parameters. I.e.: it's fine to name local parameters inside a function lowercase even when they are scalars, not only arrays.

An extension to the convention is being proposed: to name associative arrays (i.e.: hashes) capitalized, i.e.: with only first letter uppercase and the remaining letters lowercase. See the next section for an example of such hash. In case of the name consisting of multiple words each of them should be capitalized, e.g.: `typeset -A MyHash`.

This convention will increase code readibility and bring order to it.

Standard `Plugins` Hash

The plugin often has to declare global parameters that should live throughout a Zsh session. Following the namespace pollution prevention the plugin could use a hash to store the different

values. Additionally, the plugins could use a single hash parameter – called `Plugins` – to prevent the pollution even more:

```
...
typeset -gA Plugins
# An example value needed by the plugin
Plugins[MY_PLUGIN_REPO_DIR]="${0:h}"
```

This way all the data of all plugins will be kept in a single parameter, available for easy examination and overview (via e.g.: `vared Plugins`) and also not polluting the namespace.

Standard Recommended Options

The following code snippet is recommended to be included at the beginning of each of the main functions provided by the plugin:

```
emulate -L zsh
setopt extended_glob warn_create_global typeset_silent \
       no_short_loops rc_quotes no_auto_pushd
```

It resets all the options to their default state according to the `zsh` emulation mode, with use of the `local_options` option – so the options will be restored to their previous state when leaving the function.

It then alters the emulation by `6` different options:

- `extended_glob` – enables one of the main Zshell features – the advanced, built-in regex-like globing mechanism,
- `warn_create_global` – enables warnings to be printed each time a (global) variable is defined without being explicitly defined by a `typeset`, `local`, `declare`, etc. call; it allows to catch typos and missing localizations of the variables and thus prevents from writing a bad code,
- `typeset_silent` – it allows to call `typeset`, `local`, etc. multiple times on the same variable; without it the second call causes the variable contents to be printed first; using this option allows to declare variables inside loops, near the place of their use, which sometimes helps to write a more readable code,
- `no_short_loops` – disables the short-loops syntax; this is done because when the syntax is enabled it limits the parser's ability to detect errors (see this zsh-workers post for the details),
- `rc_quotes` – adds useful ability to insert apostrophes into an apostrophe-quoted string, by use of `''` inside it, e.g.: `'a string''s example'` will yield the string `a string's example`,
- `no_auto_pushd` - disables the automatic push of the directory passed to `cd` builtin onto the directory stack; this is useful, because otherwise the internal directory changes done by the plugin will pollute the global directory stack.

Standard Recommended Variables

It's good to localize the following variables at the entry of the main function of a plugin:

```
local MATCH REPLY; integer MBEGIN MEND
local -a match mbegin mend reply
```

The variables starting with `m` and `M` are being used by the substitutions utilizing `(#b)` and `(#m)` flags, respectively. They should not leak to the global scope. Also, their automatic creation would trigger the warning from the `warn_create_global` option.

The `reply` and `REPLY` parameters are being normally used to return an array or a scalar from a function, respectively – it's the standard way of passing values from functions. Their use is naturally limited to the functions called from the main function of a plugin – they should not be used to pass data around e.g.: in between prompts, thus it's natural to localize them in the main function.

Standard Function Name-Space Prefixes

The recommendation is purely subjective opinion of the author. It can evolve – if you have any remarks, don't hesitate to fill them.

The Problems Solved By The Proposition

However when adopted, the proposition will solve the following issues:

1. Using the underscore `_` to namespace functions – this isn't the right thing to do because the prefix is being already used by the completion functions, so the namespace is already filled up greatly and the plugin functions get lost in it.
2. Not using a prefix at all – this is also an unwanted practice as it pollutes the command namespace (an example of such issue appearing).
3. It would allow to quickly discriminate between function types – e.g.: seeing the `:` prefix informs the user that it's a hook-type function, while seeing the `@` prefix informs the user that it's an API-like function, etc.
4. It also provides an improvement during programming, by allowing to quickly limit the number of completions offered by the editor, e.g.: for Vim's `Ctrl-P` completing, when entering `+<Ctrl-P>`, then only a subset of the functions is being completed (see below for the type of the functions). **Note:** the editor has to be configured so that it accepts such special characters as part of keywords, for Vim it's: `:set isk+=@-@,.,+,/,:` for all of the proposed prefixes.

The Proposed Function-Name Prefixes

The proposition of the standard prefixes is as follows:

1. `.`: for regular private functions. Example function: `.prompt_zinc_get_value`.
2. `→`: for hook-like functions, so it should be used e.g.: for the Zsh hooks and the Zle hooks, but also

for any other custom hook-like mechanism in the plugin . Example function name: `→prompt_zinc_precmd`.

- previous version of the document recommended colon (`:`) for the prefix, however, it was problematic, because Windows doesn't allow colons in file names, so it wasn't possible to name an autoload function this way,
- the arrow has a rationale behind – it denotes the execution **coming back** to the function at a later time, after it has been registered as a callback or a handler,
- the arrow is easy to type on most keyboard layouts – it is `Right-Alt+I`; in case of problems with typing the character can be always copied – handler functions do occur in the code rarely,
- Zsh supports absolutely any string as a function name, because absolutely any string can be a **file** name – if there would be an exception in the name of the callables, then how would it be possible to run a script called "→abcd"? There are **no** exceptions, the function can be called even as a sequence of null bytes:

```
 $'\0'() { print hello }
 $'\0'
hello
```

3. `+`: for output functions, i.e.: for functions that print to the standard output and error or to a log, etc. Example function name: `+prompt_zinc_output_segment`.
4. `/`: for debug functions, i.e: for functions that output debug messages to the screen or to a log or e.g.: gather some debug data. **Note:** the slash makes it impossible for such functions to be auto-loaded via the `autoload` mechanism. It is somewhat risky to assume, that this will never be needed for the functions, however the limited number of available ASCII characters justifies such allocation. Example function name: `/prompt_zinc_dmsg`.
5. `@`: for API-like functions, i.e: for functions that are on a boundary to a subsystem and expose its functionality through a well-defined, in general fixed interface. For example this plugin standard defines the function `@zsh-plugin-run-on-update`, which is exposing a plugin manager's functionality in a well-defined way.

Example Code Utilizing The Prefixes

```
.zinc_register_hooks() {
    add-zsh-hook precmd :zinc_precmd
    /zinc_dmsg "Installed precmd hook with result: $?"
    @zsh-plugin-run-on-unload "add-zsh-hook -d precmd :zinc_precmd"
    +zinc_print "Zinc initialization complete"
}
```

Preventing Function Pollution

When writing a larger autoload function, it very often is the case that the function contains definitions of other functions. When the main function finishes executing, the functions are being

left defined. This might be undesired, e.g.: because of the command namespace pollution. The following snippet of code, when added at the beginning of the main function will automatically unset the sub-functions when leaving the main function:

```
# Don't leak any functions
typeset -g prjef
prjef=( ${(k)functions} )
trap "unset -f -- \"\${(k)functions[@]:|prjef}\" &>/dev/null; unset prjef" EXIT
trap "unset -f -- \"\${(k)functions[@]:|prjef}\" &>/dev/null; unset prjef; return 1"
INT
```

Replace the `prj*` prefix with your project name, e.g.: `rustef` for a `rust`-related plugin. The `*ef` stands for "entry functions". The snippet works as follows:

1. The line `prjef=( ${(k)functions} )` remembers all the functions that are currently defined – which means that the list excludes the functions that are to be yet defined by the body of the main function.
2. The code `unset -f␣-␣"${(k)functions[@]:|prjef}"` first does an subtraction of array contents – the `:|` substitution operator – of the functions that are defined at the moment of leaving of the function (the `trap`-s invoke the code in this moment) with the list of functions from the start of the main function – the ones stored in the variables `$prjef`.
3. It then unsets the resulting list of the functions – being only the newly defined functions in the main function – by passing it to `unset -f ⋯`.

This way the functions defined by the body of the main (most often an autoload) function will be only set during the execution of the function.

Preventing Parameter Pollution

When writing a plugin one often needs to keep a state during the Zsh session. To do this it is natural to use global parameters. However, when the number of the parameters grows one might want to limit it.

With the following method, only a single global parameter per plugin can be sufficient:

```
typeset -A PlgMap
typeset -A SomeMap
typeset -a some_array

# Use
PlgMap[state]=1
SomeMap[state]=1
some_array[1]=state
```

can be converted into:

```
typeset -A PlgMap

# Use
PlgMap[state]=1
PlgMap[SomeMap__state]=1
PlgMap[some_array__1]=state
```

The use of this method is very unproblematic. The author reduced the number of global parameters in one of projects by 21 by using an automatic conversion with Vim substitution patterns with back references without any problems.

Following the Standard Plugins Hash section, the plugin could even use a common hash name – `Plugins` – to lower the pollution even more.

Appendix A: Revision History (History Of Updates To The Document)

v1.1, 21/02/2020: Changed the handler-function prefix character to `→`
v1.09, 01/29/2020: 1/ Added `Standard Parameter Naming` section
v1.09, 01/29/2020: 2/ Added `Standard Plugins Hash` section
v1.08, 01/29/2020: Added the `PMSPEC` section
v1.07, 01/29/2020: Added the `functions`-directory section
v1.05, 11/22/2019: Restored the quoting to the `$0` assignments + justification
v1.0, 11/22/2019: Removed quoting from the `$0` assignments
v0.99, 10/26/2019: Added `Adoption Status` sub-sections
v0.98, 10/25/2019: 1/ Added `Standard Recommended Variables` section
v0.98, 10/25/2019: 2/ Added `Standard Function Name-Space Prefixes` section
v0.98, 10/25/2019: 3/ Added `Preventing Function Pollution` section
v0.98, 10/25/2019: 4/ Added `Preventing Parameter Pollution` section
v0.97, 10/23/2019: Added `Standard Recommended Options` section
v0.96, 10/23/2019: Added `@zsh-plugin-run-on-unload` and `@zsh-plugin-run-on-update` calls
v0.95, 07/31/2019: Plugin unload function `*_unload_plugin` -→ `*_plugin_unload`
v0.94, 07/20/2019: Add initial version of the best practices section
v0.93, 07/20/2019: 1/ Add the second line to the `$0` handling.
v0.93, 07/20/2019: 2/ Reformat to 80 columns
v0.92, 07/14/2019: 1/ Rename LOADED_PLUGINS to zsh_loaded_plugins.
v0.92, 07/14/2019: 2/ Suggest that $ZPFX is optional.
v0.91, 06/02/2018: Fix the link to the PDF for Github.
v0.9, 12/12/2017: Remove ZERO references (wrong design), add TOC.

Reminder: The date format that uses slashes is `MM/DD/YYYY`.