OpenWrt Wiki

Configuration in scripts

OpenWrt offers a set of standard shell procedures to interface with UCI in order to efficiently read and process configuration files from within shell scripts. This is most likely useful for writing startup scripts in `/etc/init.d/`.

Loading

To be able to load UCI configuration files, you need to include the common functions with:

```
. /lib/functions.sh
```

Then you can use `config_load` *`name`* to load config files.

The function first checks for *`name`* as absolute filename and falls back to loading it from `/etc/config/` (which is the most common way of using it).

Callbacks

The first method for parsing an UCI config file is through **callbacks**, which will be called for each `section`, `option` and `list` encountered during parsing.

You can define three callbacks: `config_cb()`, `option_cb()`, and `list_cb()`. You need to define these functions before calling `config_load`, but after including `/lib/functions.sh`.

Additionally, you may call `reset_cb()` to reset all three callbacks to no-op functions.

"config_cb" callback

The `config_cb` procedure is called every time a UCI section heading is encountered during parsing. Also an extra call to `config_cb` (without any argument) is generated after `config_load` is done. This may be useful if you accumulate something while parsing options (see below) and want to do something with the accumulated values (write them to a file, …) when a section has been entirely parsed.

When called, the procedure receives two arguments:

1. Type, the section type, e.g. `interface` for `config interface wan`
2. Name, the section name, e.g. `wan` for `config interface wan` or an autogenerated ID like `cfg13abef` for anonymous sections like `config route`

```
config_cb() { local type="$1" local name="$2" # commands to be run for every section }
```

Also an extra call to `config_cb` (without a new section) is generated after config_load is done - this allows you to process sections both before and after all options were processed.

"option_cb" callback

Similar to `config_cb`, the `option_cb` procedure is called each time a UCI option is encountered.

When called, the procedure receives two arguments:

1. Name, the option name, e.g. `ifname` for `option ifname eth0`
2. Value, the option value, e.g. `eth0` for `option ifname eth0`

```
option_cb() { local name="$1" local value="$2" # commands to be run for every option }
```

Within the callback, the ID of the current section is accessible via the `$CONFIG_SECTION` variable.

You can define `option_cb` inside `config_cb`: this allows to define different `option_cb` callbacks based on the section type. This allows you to process every single config section based on its type individually.

"list_cb" callback

`list_cb` works exactly like `option_cb` above, but gets called each time a `list` item is encountered.

When called, the procedure receives two arguments:

1. Name, the list name, e.g. `server` for `list server 192.168.42.1`
2. Value, the list value, e.g. `192.168.42.1` for `list server 192.168.42.1`

```
list_cb() { local name="$1" local value="$2" # commands to be run for every list item }
```

Each item of a given list generates a new call to `list_cb`.

Iterating

An alternative approach to callback based parsing is iterating the configuration sections with the `config_foreach` procedure.

The `config_foreach` procedure takes at least one argument:

1. Function, name of a previously defined procedure called for each encountered section
2. Type (optional), only iterate sections of the given type, skip others
3. Additional arguments (optional), all following arguments are passed to the callback procedure as-is

In the example below, the `handle_interface` procedure is called for each `config interface` section in `/etc/config/network`. The string `test` is passed as second argument on each invocation.

```
handle_interface() { local config="$1" local custom="$2" # run commands for every interface section
} config_load network config_foreach handle_interface interface test
```

It is possible to abort the iteration from within the callback by returning a non-zero value (`return 1`).

Within the per-section callback, the `config_get` or `config_set` procedures may be used to read or set values belonging to the currently processed section.

Reading options

The `config_get` procedure takes at least three arguments:

1. Name of a variable to store the retrieved value in
2. ID of the section to read the value from
3. Name of the option to read the value from
4. Default (optional), value to return instead if option is unset

```
… # read the value of "option ifname" into the "iface" variable # $config contains the ID of the
current section local iface config_get iface "$config" ifname echo "Interface name is $iface" …
```

Setting options

The `config_set` procedure takes three arguments:

1. ID of the section to set the option in
2. Name of the option to assign the value to
3. Value to assign

```
… # set the value of "option auto" to "0" # $config contains the ID of the current section
config_set "$config" auto 0 …
```

Note that values changed with `config_set` are only kept in memory. Subsequent calls to `config_get` will return the updated values but the underlying configuration files are *not* altered. If you want to alter values, use the uci_* functions from /lib/config/uci.sh which are automatically included by /etc/functions.sh.

Direct access

If the name of a configuration section is known in advance (it is named), options can be read directly without using a section iterator callback.

The example below reads "option proto" from the "config interface wan" section.

```
… local proto config_get proto wan proto echo "Current WAN protocol is $proto" …
```

Reading lists

Some UCI configurations may contain `list` options in the form:

```
… list network lan list network wifi …
```

Calling `config_get` on the `network` list will return the list values separated by space, `lan wifi` in this example.

However, this behaviour might break values if the list items itself contain spaces like illustrated below:

```
… list animal 'White Elephant' list animal 'Mighty Unicorn' …
```

The `config_get` approach would return the values in the form `White Elephant Mighty Unicorn` and the original list items are not clearly separated anymore.

To circumvent this problem, the `config_list_foreach` iterator can be used. It works similar to `config_foreach` but operates on list values instead of config sections.

The `config_list_foreach` procedure takes at least three arguments:

1. ID of the section to read the list from
2. Name of the list option to read items from
3. Procedure to call for each list item
4. Additional arguments (optional), all following arguments are passed to the callback procedure as-is

```
# handle list items in a callback # $config contains the ID of the section handle_animal() { local
value="$1" # do something with $value } config_list_foreach "$config" animal handle_animal
```

Note: for editing value, use the uci command line tool.

Reading booleans

Boolean options may contain various values to signify a *true* value like `on`, `true`, `enabled` or `1`.

The `config_get_bool` procedure simplifies the process of reading a boolean option and casting it to a plain integer value (`1` for *true* and `0` for *false*).

At least three arguments are expected by the procedure:

1. Name of a variable to store the retrieved value in
2. ID of the section to read the value from
3. Name of the option to read the value from
4. Default (optional), boolean value to return if option is unset

Should I use callbacks or explicit access to options?

It depends on what you want to parse.

Use case and example for callbacks

If, for a given section, all options should be treated in the same way (say, write them to a config file), whatever their order or name, then callbacks are a good option. It allows you to parse all options without having to know their name in advance.

For instance, consider this script:

```
config_cb() {
    local type="$1"
    local name="$2"
    if [ "$type" = "mysection" ]
    then
        option_cb() {
            local option="$1"
            local value="$2"
            echo "${option//_/-} $value" >> /var/etc/myfile.conf
        }
    else {
        option_cb() { return; }
    }
}
```

This would parse a configuration like this one:

```
config mysection foo option link_quality auto option rxcost 256 option hello_inverval 4
```

And generate a file containing:

```
link-quality auto rxcost 256 hello-interval 4
```

Use case and example for direct access

On the other hand, if different options play a very different role, then it may be more convenient to pick each option explicitly.

For instance, babeld [http://www.pps.univ-paris-diderot.fr/~jch/software/babel/] has the following syntax for filters:

```
{in|out|redistribute} selector* [allow|deny|metric n]
```

where a selector can be one of:

```
local
ip prefix
proto p
if interface
...
```

At most one instance of each selector type is allowed in a filter. The UCI configuration looks like this:

```
config filter
    option ip '172.22.0.0/15'
    option local 'true'
    option type 'redistribute'
    option action 'metric 128'
```

The result of this section should be:

```
redistribute ip 172.22.0.0/15 local metric 128
```

That is, the type should come first, then selectors (with a special case for `local`, which doesn't take an argument), and finally the action. The basic parsing method is the following:

```
parse_filter() {
    local section="$1"
    local _buffer
    local _value

    config_get _value "$section" "type"
    append _buffer "$_value"

    config_get _value "$section" "ip"
    append _buffer "ip $_value"
    config_get _value "$section" "proto"
    append _buffer "proto $_value"
    ...

    config_get_bool _value "$section" "local" 0
    [ "$_value" -eq 1 ] && append _buffer "local"

    config_get _value "$section" "action"
    append _buffer "$_value"

    echo "$_buffer" >> /var/etc/babeld.conf
}
config_load babeld
config_foreach parse_filter filter
```

Mixing the two styles

Of course, it is possible to mix the two styles of parsing, for instance parsing one section with callbacks and another section explicitly. Don't forget to reset or unset the callbacks (`reset_cb` or `option_cb() { return; }`) for sections you don't want to parse with callbacks.

doc/devel/config-scripting.txt · Last modified: 2015/12/10 17:13 by b2ag