Practical rc.d scripting in BSD

Yar Tikhiy <yar@FreeBSD.org >

Revision: 43184

Copyright © 2005, 2006, 2012 The FreeBSD Project

FreeBSD is a registered trademark of the FreeBSD Foundation.

NetBSD is a registered trademark of the NetBSD Foundation.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this document, and the FreeBSD Project was aware of the trademark claim, the designations have been followed by the "™" or the "®" symbol.

2013-11-13 by hrs.

Abstract

Beginners may find it difficult to relate the facts from the formal documentation on the BSD `rc.d` framework with the practical tasks of `rc.d` scripting. In this article, we consider a few typical cases of increasing complexity, show `rc.d` features suited for each case, and discuss how they work. Such an examination should provide reference points for further study of the design and efficient application of `rc.d`.

Table of Contents

1. Introduction 2
2. Outlining the task 3
3. A dummy script 3
4. A configurable dummy script 6
5. Startup and shutdown of a simple daemon 9
6. Startup and shutdown of an advanced daemon 10
7. Connecting a script to the rc.d framework 16
8. Giving more flexibility to an rc.d script 20
9. Further reading 22

1. Introduction

The historical BSD had a monolithic startup script, `/etc/rc`. It was invoked by init(8) at system boot time and performed all userland tasks required for multi-user operation: checking and mounting file systems, setting up the network, starting daemons, and so on. The precise list of tasks was not the same in every system; admins needed to customize it. With few exceptions, `/etc/rc` had to be modified, and true hackers liked it.

The real problem with the monolithic approach was that it provided no control over the individual components started from `/etc/rc`. For instance, `/etc/rc` could not restart a single daemon. The system admin had to find the daemon process by hand, kill it, wait until it actually exited, then browse through `/etc/rc` for the flags, and finally type the full command line to start the daemon again. The task would become even more difficult and prone to errors if the service to restart consisted of more than one daemon or demanded additional actions. In a few words, the single script failed to fulfil what scripts are for: to make the system admin's life easier.

Later there was an attempt to split out some parts of `/etc/rc` for the sake of starting the most important subsystems separately. The notorious example was `/etc/netstart` to bring up networking. It did allow for accessing the network from single-user mode, but it did not integrate well into the automatic startup process because parts of its code needed to interleave with actions essentially unrelated to networking. That was why `/etc/netstart` mutated into `/etc/rc.network`. The latter was no longer an ordinary script; it comprised of large, tangled sh(1) functions called from `/etc/rc` at different stages of system startup. However, as the startup tasks grew diverse and sophisticated, the "quasi-modular" approach became even more of a drag than the monolithic `/etc/rc` had been.

Without a clean and well-designed framework, the startup scripts had to bend over backwards to satisfy the needs of rapidly developing BSD-based operating systems. It became obvious at last that more steps are necessary on the way to a fine-grained and extensible `rc` system. Thus BSD `rc.d` was born. Its acknowledged fathers were Luke Mewburn and the NetBSD community. Later it was imported into FreeBSD. Its name refers to the location of system scripts for individual services, which is in `/etc/rc.d`. Soon we will learn about more components of the `rc.d` system and see how the individual scripts are invoked.

The basic ideas behind BSD `rc.d` are *fine modularity* and *code reuse*. *Fine modularity* means that each basic "service" such as a system daemon or primitive startup task gets its own sh(1) script able to start the service, stop it, reload it, check its status. A particular action is chosen by the command-line argument to the script. The `/etc/rc` script still drives system startup, but now it merely invokes the smaller scripts one by one with the `start` argument. It is easy to perform shutdown tasks as well by running the same set of scripts with the `stop` argument, which is done by `/etc/rc.shutdown`. Note how closely this follows the Unix way of having a set of small specialized tools, each fulfilling its task as well as possible. *Code reuse* means that common operations are implemented as sh(1) functions

and collected in `/etc/rc.subr` . Now a typical script can be just a few lines' worth of sh(1) code. Finally, an important part of the `rc.d` framework is rcorder(8), which helps `/etc/rc` to run the small scripts orderly with respect to dependencies between them. It can help `/etc/rc.shutdown` , too, because the proper order for the shutdown sequence is opposite to that of startup.

The BSD `rc.d` design is described in the original article by Luke Mewburn, and the `rc.d` components are documented in great detail in the respective manual pages. However, it might not appear obvious to an `rc.d` newbie how to tie the numerous bits and pieces together in order to create a well-styled script for a particular task. Therefore this article will try a different approach to describe `rc.d`. It will show which features should be used in a number of typical cases, and why. Note that this is not a how-to document because our aim is not at giving ready-made recipes, but at showing a few easy entrances into the `rc.d` realm. Neither is this article a replacement for the relevant manual pages. Do not hesitate to refer to them for more formal and complete documentation while reading this article.

There are prerequisites to understanding this article. First of all, you should be familiar with the sh(1) scripting language in order to master `rc.d`. In addition, you should know how the system performs userland startup and shutdown tasks, which is described in rc(8).

This article focuses on the FreeBSD branch of `rc.d`. Nevertheless, it may be useful to NetBSD developers, too, because the two branches of BSD `rc.d` not only share the same design but also stay similar in their aspects visible to script authors.

2. Outlining the task

A little consideration before starting `$EDITOR` will not hurt. In order to write a well-tempered `rc.d` script for a system service, we should be able to answer the following questions first:

- Is the service mandatory or optional?
- Will the script serve a single program, e.g., a daemon, or perform more complex actions?
- Which other services will our service depend on, and vice versa?

From the examples that follow we will see why it is important to know the answers to these questions.

3. A dummy script

The following script just emits a message each time the system boots up:

```
#!/bin/sh❶

. -/etc/rc.subr❷

name="dummy"❸
start_cmd="${name}_start"❹
stop_cmd=":"❺

dummy_start()❻
{
 echo -"Nothing started."
}

load_rc_config $name❼
run_rc_command -"$1"❽
```

Things to note are:

❶ An interpreted script should begin with the magic “shebang” line. That line specifies the interpreter program for the script. Due to the shebang line, the script can be invoked exactly like a binary program provided that it has the execute bit set. (See chmod(1).) For example, a system admin can run our script manually, from the command line:

```
# /etc/rc.d/dummy start
```

Note

In order to be properly managed by the `rc.d` framework, its scripts need to be written in the sh(1) language. If you have a service or port that uses a binary control utility or a startup routine written in another language, install that element in `/usr/sbin` (for the system) or `/usr/local/sbin` (for ports) and call it from a sh(1) script in the appropriate `rc.d` directory.

Tip

If you would like to learn the details of why `rc.d` scripts must be written in the sh(1) language, see how `/etc/rc` invokes

them by means of `run_rc_script` , then study the implementation of `run_rc_script` in `/etc/rc.subr` .

❷ In `/etc/rc.subr` , a number of sh(1) functions are defined for an `rc.d` script to use. The functions are documented in rc.subr(8). While it is theoretically possible to write an `rc.d` script without ever using rc.subr(8), its functions prove extremely handy and make the job an order of magnitude easier. So it is no surprise that everybody resorts to rc.subr(8) in `rc.d` scripts. We are not going to be an exception.

An `rc.d` script must "source" `/etc/rc.subr` (include it using ".") *before* it calls rc.subr(8) functions so that sh(1) has an opportunity to learn the functions. The preferred style is to source `/etc/rc.subr` first of all.

Note

Some useful functions related to networking are provided by another include file, `/etc/network.subr` .

❸ The mandatory variable `name` specifies the name of our script. It is required by rc.subr(8). That is, each `rc.d` script *must* set `name` before it calls rc.subr(8) functions.

Now it is the right time to choose a unique name for our script once and for all. We will use it in a number of places while developing the script. For a start, let us give the same name to the script file, too.

Note

The current style of `rc.d` scripting is to enclose values assigned to variables in double quotes. Keep in mind that it is just a style issue that may not always be applicable. You can safely omit quotes from around simple words without sh(1) metacharacters in them, while in certain cases you will need single quotes to prevent any interpretation of the value by sh(1). A programmer should be able to tell the language syntax from style conventions and use both of them wisely.

❹ The main idea behind rc.subr(8) is that an `rc.d` script provides handlers, or methods, for rc.subr(8) to invoke. In particular, `start`, `stop`, and other arguments to an `rc.d` script are handled this way. A method is a sh(1) expression stored in a variable

named *argument* _cmd, where *argument* corresponds to what can be specified on the script's command line. We will see later how rc.subr(8) provides default methods for the standard arguments.

Note

To make the code in `rc.d` more uniform, it is common to use `${name}` wherever appropriate. Thus a number of lines can be just copied from one script to another.

❺ We should keep in mind that rc.subr(8) provides default methods for the standard arguments. Consequently, we must override a standard method with a no-op sh(1) expression if we want it to do nothing.

❻ The body of a sophisticated method can be implemented as a function. It is a good idea to make the function name meaningful.

Important

It is strongly recommended to add the prefix `${name}` to the names of all functions defined in our script so they never clash with the functions from rc.subr(8) or another common include file.

❼ This call to rc.subr(8) loads rc.conf(5) variables. Our script makes no use of them yet, but it still is recommended to load rc.conf(5) because there can be rc.conf(5) variables controlling rc.subr(8) itself.

❽ Usually this is the last command in an `rc.d` script. It invokes the rc.subr(8) machinery to perform the requested action using the variables and methods our script has provided.

4. A configurable dummy script

Now let us add some controls to our dummy script. As you may know, `rc.d` scripts are controlled with rc.conf(5). Fortunately, rc.subr(8) hides all the complications from us. The following script uses rc.conf(5) via rc.subr(8) to see whether it is enabled in the first place, and to fetch a message to show at boot time. These two tasks in fact are independent. On the one hand, an `rc.d` script can just support enabling and disabling its service. On the

other hand, a mandatory `rc.d` script can have configuration variables. We will do both things in the same script though:

```
#!/bin/sh

. -/etc/rc.subr

name=dummy
rcvar=dummy_enable❶

start_cmd="${name}_start"
stop_cmd=":"

load_rc_config $name❷
: ${dummy_enable:=no} ❸
: ${dummy_msg="Nothing started."}❹

dummy_start()
{
 echo -"$dummy_msg"❺
}

run_rc_command -"$1"
```

What changed in this example?

❶ The variable `rcvar` specifies the name of the ON/OFF knob variable.

❷ Now `load_rc_config` is invoked earlier in the script, before any rc.conf(5) variables are accessed.

Note

While examining `rc.d` scripts, keep in mind that sh(1) defers the evaluation of expressions in a function until the latter is called. Therefore it is not an error to invoke `load_rc_config` as late as just before `run_rc_command` and still access rc.conf(5) variables from the method functions exported to `run_rc_command` . This is because the method functions are to be called by `run_rc_command` , which is invoked *after* `load_rc_config` .

❸ A warning will be emitted by `run_rc_command` if `rcvar` itself is set, but the indicated knob variable is unset. If your `rc.d` script is for the base system, you should add a default setting for the knob to `/etc/defaults/rc.conf` and document it in rc.conf(5). Otherwise it is your script that should provide a default setting for the knob. The canonical approach to the latter case is shown in the example.

Note

You can make rc.subr(8) act as though the knob is set to `ON`, irrespective of its current setting, by prefixing the argument to the script with `one` or `force`, as in `onestart` or `forcestop`. Keep in mind though that `force` has other dangerous effects we will touch upon below, while `one` just overrides the ON/OFF knob. E.g., assume that `dummy_enable` is `OFF`. The following command will run the `start` method in spite of the setting:

```
# /etc/rc.d/dummy onestart
```

4 Now the message to be shown at boot time is no longer hard-coded in the script. It is specified by an rc.conf(5) variable named `dummy_msg`. This is a trivial example of how rc.conf(5) variables can control an `rc.d` script.

Important

The names of all rc.conf(5) variables used exclusively by our script *must* have the same prefix: `${name}_`. For example: `dummy_mode`, `dummy_state_file`, and so on.

Note

While it is possible to use a shorter name internally, e.g., just `msg`, adding the unique prefix `${name}_` to all global names introduced by our script will save us from possible collisions with the rc.subr(8) namespace.

As a rule, `rc.d` scripts of the base system need not provide defaults for their rc.conf(5) variables because the defaults should be set in `/etc/defaults/rc.conf` instead. On the other hand, `rc.d` scripts for ports should provide the defaults as shown in the example.

❺ Here we use `dummy_msg` to actually control our script, i.e., to emit a variable message. Use of a shell function is overkill here, since it only runs a single command; an equally valid alternative is:

```
start_cmd="echo \"$dummy_msg\""
```

5. Startup and shutdown of a simple daemon

We said earlier that rc.subr(8) could provide default methods. Obviously, such defaults cannot be too general. They are suited for the common case of starting and shutting down a simple daemon program. Let us assume now that we need to write an `rc.d` script for such a daemon called `mumbled`. Here it is:

```
#!/bin/sh

. -/etc/rc.subr

name=mumbled
rcvar=mumbled_enable

command="/usr/sbin/${name}"❶

load_rc_config $name
run_rc_command -"$1"
```

Pleasingly simple, isn't it? Let us examine our little script. The only new thing to note is as follows:

❶ The `command` variable is meaningful to rc.subr(8). If it is set, rc.subr(8) will act according to the scenario of serving a conventional daemon. In particular, the default methods will be provided for such arguments: `start`, `stop`, `restart`, `poll`, and `status`.

The daemon will be started by running `$command` with command-line flags specified by `$mumbled_flags`. Thus all the input data for the default `start` method are available in the variables set by our script. Unlike `start`, other methods may require additional information about the process started. For instance, `stop` must know the PID of the process to terminate it. In the present case, rc.subr(8) will scan through the list of all processes, looking for a process with its name equal to `$procname`. The latter is another variable of meaning to rc.subr(8), and its value defaults to that of `command`. In other words, when we set `command`, `procname` is effectively set to the same value. This enables our script to kill the daemon and to check if it is running in the first place.

Note

Some programs are in fact executable scripts. The system runs such a script by starting its interpreter and passing the name of the script to it as a command-line argument. This is reflected in the list of processes, which can confuse rc.subr(8). You should additionally set `command_interpreter` to let rc.subr(8) know the actual name of the process if `$command` is a script.

For each `rc.d` script, there is an optional rc.conf(5) variable that takes precedence over `command`. Its name is constructed as follows: `${name}_program`, where `name` is the mandatory variable we discussed earlier. E.g., in this case it will be `mumbled_program`. It is rc.subr(8) that arranges `${name}_program` to override `command`.

Of course, sh(1) will permit you to set `${name}_program` from rc.conf(5) or the script itself even if `command` is unset. In that case, the special properties of `${name}_program` are lost, and it becomes an ordinary variable your script can use for its own purposes. However, the sole use of `${name}_program` is discouraged because using it together with `command` became an idiom of `rc.d` scripting.

For more detailed information on default methods, refer to rc.subr(8).

6. Startup and shutdown of an advanced daemon

Let us add some meat onto the bones of the previous script and make it more complex and featureful. The default methods can do a good job for us, but we may need some of their aspects tweaked. Now we will learn how to tune the default methods to our needs.

```
#!/bin/sh

. -/etc/rc.subr

name=mumbled
rcvar=mumbled_enable

command="/usr/sbin/${name}"
command_args="mock arguments > -/dev/null 2>&1"❶
```

```
pidfile="/var/run/${name}.pid"❷

required_files="/etc/${name}.conf -/usr/share/misc/${name}.rules"❸

sig_reload="USR1"❹

start_precmd="${name}_prestart"❺
stop_postcmd="echo Bye-bye"❻

extra_commands="reload plugh xyzzy"❼

plugh_cmd="mumbled_plugh"❽
xyzzy_cmd="echo -'Nothing happens.'"

mumbled_prestart()
{
 if checkyesno mumbled_smart; then❾
  rc_flags="-o smart ${rc_flags}"❿
 fi
 case -"$mumbled_mode" in
 foo)
  rc_flags="-frotz ${rc_flags}"
  ;;
 bar)
  rc_flags="-baz ${rc_flags}"
  ;;
 *)
  warn -"Invalid value for mumbled_mode"⓫
  return 1⓬
  ;;
 esac
 run_rc_command xyzzy⓭
 return 0
}

mumbled_plugh()⓮
{
 echo -'A hollow voice says -"plugh".'
}

load_rc_config $name
run_rc_command -"$1"
```

❶ Additional arguments to `$command` can be passed in `command_args` . They will be added to the command line after `$mumbled_flags` . Since the final command line is passed to `eval` for its actual execution, input and output redirections can be specified in `command_args` .

Note

Never include dashed options, like `-X` or `--foo`, in `command_args`. The contents of `command_args` will appear at the end of the final command line, hence they are likely to follow arguments present in `${name}_flags`; but most commands will not recognize dashed options after ordinary arguments. A better way of passing additional options to `$command` is to add them to the beginning of `${name}_flags`. Another way is to modify `rc_flags` as shown later.

❷ A good-mannered daemon should create a *pidfile* so that its process can be found more easily and reliably. The variable `pidfile`, if set, tells rc.subr(8) where it can find the pidfile for its default methods to use.

Note

In fact, rc.subr(8) will also use the pidfile to see if the daemon is already running before starting it. This check can be skipped by using the `faststart` argument.

❸ If the daemon cannot run unless certain files exist, just list them in `required_files`, and rc.subr(8) will check that those files do exist before starting the daemon. There also are `required_dirs` and `required_vars` for directories and environment variables, respectively. They all are described in detail in rc.subr(8).

Note

The default method from rc.subr(8) can be forced to skip the prerequisite checks by using `forcestart` as the argument to the script.

❹ We can customize signals to send to the daemon in case they differ from the well-known ones. In particular, `sig_reload` specifies the signal that makes the daemon reload its configuration; it is SIGHUP by default. Another signal is sent to stop

the daemon process; the default is SIGTERM, but this can be changed by setting `sig_stop` appropriately.

Note

The signal names should be specified to rc.subr(8) without the `SIG` prefix, as it is shown in the example. The FreeBSD version of kill(1) can recognize the `SIG` prefix, but the versions from other OS types may not.

5 6 Performing additional tasks before or after the default methods is easy. For each command-argument supported by our script, we can define *`argument`*`_precmd` and *`argument`*`_postcmd`. These sh(1) commands are invoked before and after the respective method, as it is evident from their names.

Note

Overriding a default method with a custom *`argument`*`_cmd` still does not prevent us from making use of *`argument`*`_precmd` or *`argument`*`_postcmd` if we need to. In particular, the former is good for checking custom, sophisticated conditions that should be met before performing the command itself. Using *`argument`*`_precmd` along with *`argument`*`_cmd` lets us logically separate the checks from the action.

Do not forget that you can cram any valid sh(1) expressions into the methods, pre-, and post-commands you define. Just invoking a function that makes the real job is a good style in most cases, but never let style limit your understanding of what is going on behind the curtain.

7 If we would like to implement custom arguments, which can also be thought of as *commands* to our script, we need to list them in `extra_commands` and provide methods to handle them.

Note

The `reload` command is special. On the one hand, it has a preset method in rc.subr(8). On the other hand, `reload` is not offered by default. The reason is that not all daemons use the same reload mechanism and some have nothing to reload at all. So we need to ask explicitly that the builtin functionality be provided. We can do so via `extra_commands` .

What do we get from the default method for `reload`? Quite often daemons reload their configuration upon reception of a signal — typically, SIGHUP. Therefore rc.subr(8) attempts to reload the daemon by sending a signal to it. The signal is preset to SIGHUP but can be customized via `sig_reload` if necessary.

❽❿ Our script supports two non-standard commands, `plugh` and `xyzzy`. We saw them listed in `extra_commands` , and now it is time to provide methods for them. The method for `xyzzy` is just inlined while that for `plugh` is implemented as the `mumbled_plugh` function.

Non-standard commands are not invoked during startup or shutdown. Usually they are for the system admin's convenience. They can also be used from other subsystems, e.g., devd(8) if specified in devd.conf(5).

The full list of available commands can be found in the usage line printed by rc.subr(8) when the script is invoked without arguments. For example, here is the usage line from the script under study:

```
# /etc/rc.d/mumbled
Usage: -/etc/rc.d/mumbled [fast|force|one](start|stop|restart|
rcvar|reload|plugh|xyzzy|status|poll)
```

⓭ A script can invoke its own standard or non-standard commands if needed. This may look similar to calling functions, but we know that commands and shell functions are not always the same thing. For instance, `xyzzy` is not implemented as a function here. In addition, there can be a pre-command and post-command, which should be invoked orderly. So the proper way for a script to run its own command is by means of rc.subr(8), as shown in the example.

❾ A handy function named `checkyesno` is provided by rc.subr(8). It takes a variable name as its argument and returns a zero exit code if and only if the variable is set to `YES`, or `TRUE`, or `ON`, or `1`, case insensitive; a non-zero exit code is returned otherwise. In the latter case, the function tests the variable for being set to `NO`, `FALSE`, `OFF`, or `0`,

case insensitive; it prints a warning message if the variable contains anything else, i.e., junk.

Keep in mind that for sh(1) a zero exit code means true and a non-zero exit code means false.

Important

The `checkyesno` function takes a *variable name*. Do not pass the expanded *value* of a variable to it; it will not work as expected.

The following is the correct usage of `checkyesno`:

```
if checkyesno mumbled_enable; then
        foo
fi
```

On the contrary, calling `checkyesno` as shown below will not work — at least not as expected:

```
if checkyesno -"${mumbled_enable}"; then
        foo
fi
```

10. We can affect the flags to be passed to `$command` by modifying `rc_flags` in `$start_precmd`.
11. In certain cases we may need to emit an important message that should go to syslog as well. This can be done easily with the following rc.subr(8) functions: `debug`, `info`, `warn`, and `err`. The latter function then exits the script with the code specified.
12. The exit codes from methods and their pre-commands are not just ignored by default. If *`argument`*`_precmd` returns a non-zero exit code, the main method will not be performed. In turn, *`argument`*`_postcmd` will not be invoked unless the main method returns a zero exit code.

Note

However, rc.subr(8) can be instructed from the command line to ignore those exit codes and invoke all commands anyway by prefixing an argument with `force`, as in `forcestart`.

7. Connecting a script to the rc.d framework

After a script has been written, it needs to be integrated into `rc.d`. The crucial step is to install the script in `/etc/rc.d` (for the base system) or `/usr/local/etc/rc.d` (for ports). Both `<bsd.prog.mk>` and `<bsd.port.mk>` provide convenient hooks for that, and usually you do not have to worry about the proper ownership and mode. System scripts should be installed from `src/etc/rc.d` through the `Makefile` found there. Port scripts can be installed using `USE_RC_SUBR` as described in the Porter's Handbook.

However, we should consider beforehand the place of our script in the system startup sequence. The service handled by our script is likely to depend on other services. For instance, a network daemon cannot function without the network interfaces and routing up and running. Even if a service seems to demand nothing, it can hardly start before the basic filesystems have been checked and mounted.

We mentioned rcorder(8) already. Now it is time to have a close look at it. In a nutshell, rcorder(8) takes a set of files, examines their contents, and prints a dependency-ordered list of files from the set to `stdout`. The point is to keep dependency information *inside* the files so that each file can speak for itself only. A file can specify the following information:

- the names of the "conditions" (which means services to us) it *provides*;
- the names of the "conditions" it *requires*;
- the names of the "conditions" this file should run *before*;
- additional *keywords* that can be used to select a subset from the whole set of files (rcorder(8) can be instructed via options to include or omit the files having particular keywords listed.)

It is no surprise that rcorder(8) can handle only text files with a syntax close to that of sh(1). That is, special lines understood by rcorder(8) look like sh(1) comments. The syntax of such special lines is rather rigid to simplify their processing. See rcorder(8) for details.

Besides using rcorder(8) special lines, a script can insist on its dependency upon another service by just starting it forcibly. This can be needed when the other service is optional and will not start by itself because the system admin has disabled it mistakenly in rc.conf(5).

With this general knowledge in mind, let us consider the simple daemon script enhanced with dependency stuff:

```
#!/bin/sh

# PROVIDE: mumbled oldmumble ❶
# REQUIRE: DAEMON cleanvar frotz❷
# BEFORE:  LOGIN❸
# KEYWORD: nojail shutdown❹
```

```
. -/etc/rc.subr

name=mumbled
rcvar=mumbled_enable

command="/usr/sbin/${name}"
start_precmd="${name}_prestart"

mumbled_prestart()
{
 if -! checkyesno frotz_enable && \
     -! -/etc/rc.d/frotz forcestatus 1>/dev/null 2>&1; then
  force_depend frotz -|| return 1❺
 fi
 return 0
}

load_rc_config $name
run_rc_command -"$1"
```

As before, detailed analysis follows:

❶ That line declares the names of "conditions" our script provides. Now other scripts can record a dependency on our script by those names.

Note

Usually a script specifies a single condition provided. However, nothing prevents us from listing several conditions there, e.g., for compatibility reasons.

In any case, the name of the main, or the only, `PROVIDE:` condition should be the same as `${name}`.

❷❸ So our script indicates which "conditions" provided by other scripts it depends on. According to the lines, our script asks rcorder(8) to put it after the script(s) providing `DAEMON` and `cleanvar`, but before that providing `LOGIN`.

Note

The `BEFORE:` line should not be abused to work around an incomplete dependency list in the other script. The appropriate case for using `BEFORE:` is when the other script does not care about ours, but our script can do its task better if run before the other one. A typical real-life example is the network inter-

faces vs. the firewall: While the interfaces do not depend on the firewall in doing their job, the system security will benefit from the firewall being ready before there is any network traffic.

Besides conditions corresponding to a single service each, there are meta-conditions and their "placeholder" scripts used to ensure that certain groups of operations are performed before others. These are denoted by UPPERCASE names. Their list and purposes can be found in rc(8).

Keep in mind that putting a service name in the `REQUIRE:` line does not guarantee that the service will actually be running by the time our script starts. The required service may fail to start or just be disabled in rc.conf(5). Obviously, rcorder(8) cannot track such details, and rc(8) will not do that either. Consequently, the application started by our script should be able to cope with any required services being unavailable. In certain cases, we can help it as discussed below.

As we remember from the above text, rcorder(8) keywords can be used to select or leave out some scripts. Namely any rcorder(8) consumer can specify through `-k` and `-s` options which keywords are on the "keep list" and "skip list", respectively. From all the files to be dependency sorted, rcorder(8) will pick only those having a keyword from the keep list (unless empty) and not having a keyword from the skip list.

In FreeBSD, rcorder(8) is used by `/etc/rc` and `/etc/rc.shutdown` . These two scripts define the standard list of FreeBSD `rc.d` keywords and their meanings as follows:

`nojail`
: The service is not for jail(8) environment. The automatic startup and shutdown procedures will ignore the script if inside a jail.

`nostart`
: The service is to be started manually or not started at all. The automatic startup procedure will ignore the script. In conjunction with the `shutdown` keyword, this can be used to write scripts that do something only at system shutdown.

`shutdown`
: This keyword is to be listed *explicitly* if the service needs to be stopped before system shutdown.

Note

When the system is going to shut down, `/etc/rc.shutdown` runs. It assumes that most `rc.d` scripts have nothing to do at that time. Therefore `/etc/rc.shutdown` selectively invokes `rc.d` scripts with the `shutdown` keyword, effectively ignoring the rest of the scripts. For even faster shutdown, `/etc/rc.shutdown` passes the `faststop` command to the scripts it runs so that they skip preliminary checks, e.g., the pidfile check. As dependent services should be stopped before their prerequisites, `/etc/rc.shutdown` runs the scripts in reverse dependency order.

If writing a real `rc.d` script, you should consider whether it is relevant at system shutdown time. E.g., if your script does its work in response to the `start` command only, then you need not include this keyword. However, if your script manages a service, it is probably a good idea to stop it before the system proceeds to the final stage of its shutdown sequence described in halt(8). In particular, a service should be stopped explicitly if it needs considerable time or special actions to shut down cleanly. A typical example of such a service is a database engine.

❺ To begin with, `force_depend` should be used with much care. It is generally better to revise the hierarchy of configuration variables for your `rc.d` scripts if they are interdependent.

If you still cannot do without `force_depend`, the example offers an idiom of how to invoke it conditionally. In the example, our `mumbled` daemon requires that another one, `frotz`, be started in advance. However, `frotz` is optional, too; and rcorder(8) knows nothing about such details. Fortunately, our script has access to all rc.conf(5) variables. If `frotz_enable` is true, we hope for the best and rely on `rc.d` to have started `frotz`. Otherwise we forcibly check the status of `frotz`. Finally, we enforce our dependency on `frotz` if it is found to be not running. A warning message will be emitted by `force_depend` because it should be invoked only if a misconfiguration has been detected.

8. Giving more flexibility to an rc.d script

When invoked during startup or shutdown, an `rc.d` script is supposed to act on the entire subsystem it is responsible for. E.g., `/etc/rc.d/netif` should start or stop all network interfaces described by rc.conf(5). Either task can be uniquely indicated by a single command argument such as `start` or `stop`. Between startup and shutdown, `rc.d` scripts help the admin to control the running system, and it is when the need for more flexibility and precision arises. For instance, the admin may want to add the settings of a new network interface to rc.conf(5) and then to start it without interfering with the operation of the existing interfaces. Next time the admin may need to shut down a single network interface. In the spirit of the command line, the respective `rc.d` script calls for an extra argument, the interface name.

Fortunately, rc.subr(8) allows for passing any number of arguments to script's methods (within the system limits). Due to that, the changes in the script itself can be minimal.

How can rc.subr(8) gain access to the extra command-line arguments. Should it just grab them directly? Not by any means. Firstly, an sh(1) function has no access to the positional parameters of its caller, but rc.subr(8) is just a sack of such functions. Secondly, the good manner of `rc.d` dictates that it is for the main script to decide which arguments are to be passed to its methods.

So the approach adopted by rc.subr(8) is as follows: `run_rc_command` passes on all its arguments but the first one to the respective method verbatim. The first, omitted, argument is the name of the method itself: `start`, `stop`, etc. It will be shifted out by `run_rc_command` , so what is `$2` in the original command line will be presented as `$1` to the method, and so on.

To illustrate this opportunity, let us modify the primitive dummy script so that its messages depend on the additional arguments supplied. Here we go:

```
#!/bin/sh

. -/etc/rc.subr

name="dummy"
start_cmd="${name}_start"
stop_cmd=":"
kiss_cmd="${name}_kiss"
extra_commands="kiss"

dummy_start()
{
        if [ $# --gt 0 -]; then❶
                echo -"Greeting message: $*"
        else
                echo -"Nothing started."
        fi
```

```
}

dummy_kiss()
{
        echo --n -"A ghost gives you a kiss"
        if [ $# --gt 0 -]; then❷
                echo --n -" and whispers: $*"
        fi
        case -"$*" in
        *[.!?])
                echo
                -;;
        *)
                echo -.
                -;;
        esac
}

load_rc_config $name
run_rc_command -"$@"❸
```

What essential changes can we notice in the script?

❶ All arguments you type after `start` can end up as positional parameters to the respective method. We can use them in any way according to our task, skills, and fancy. In the current example, we just pass all of them to echo(1) as one string in the next line — note `$*` within the double quotes. Here is how the script can be invoked now:

```
# /etc/rc.d/dummy start
Nothing started.
# /etc/rc.d/dummy start Hello world!
Greeting message: Hello world!
```

❷ The same applies to any method our script provides, not only to a standard one. We have added a custom method named `kiss`, and it can take advantage of the extra arguments not less than `start` does. E.g.:

```
# /etc/rc.d/dummy kiss
A ghost gives you a kiss.
# /etc/rc.d/dummy kiss Once I was Etaoin Shrdlu...
A ghost gives you a kiss and whispers: Once I was Etaoin ↺
Shrdlu...
```

❸ If we want just to pass all extra arguments to any method, we can merely substitute `"$@"` for `"$1"` in the last line of our script, where we invoke `run_rc_command .`

Important

An sh(1) programmer ought to understand the subtle difference between `$*` and `$@` as the ways to designate all positional parameters. For its in-depth discussion, refer to a good hand-

book on sh(1) scripting. *Do not* use the expressions until you fully understand them because their misuse will result in buggy and insecure scripts.

Note

Currently `run_rc_command` may have a bug that prevents it from keeping the original boundaries between arguments. That is, arguments with embedded whitespace may not be processed correctly. The bug stems from `$*` misuse.

9. Further reading

The original article by Luke Mewburn offers a general overview of `rc.d` and detailed rationale for its design decisions. It provides insight on the whole `rc.d` framework and its place in a modern BSD operating system.

The manual pages rc(8), rc.subr(8), and rcorder(8) document the `rc.d` components in great detail. You cannot fully use the `rc.d` power without studying the manual pages and referring to them while writing your own scripts.

The major source of working, real-life examples is `/etc/rc.d` in a live system. Its contents are easy and pleasant to read because most rough corners are hidden deep in rc.subr(8). Keep in mind though that the `/etc/rc.d` scripts were not written by angels, so they might suffer from bugs and suboptimal design decisions. Now you can improve them!