6. Package maintainer scripts and installation procedure — Debian Policy Manual v4.7.4.1

Archived: 2026-04-05 18:24:14 UTC

6.1. Introduction to package maintainer scripts

It is possible to supply scripts as part of a package which the package management system will run for you when your package is installed, upgraded or removed.

These scripts are the package metadata files `preinst`, `postinst`, `prerm` and `postrm`. They must be proper executable files; if they are scripts (which is recommended), they must start with the usual `#!` convention. They should be readable and executable by anyone, and must not be world-writable.

The package management system looks at the exit status from these scripts. It is important that they exit with a non-zero status if there is an error, so that the package management system can stop its processing. For shell scripts this means that you *almost always* need to use `set -e` (this is usually true when writing shell scripts, in fact). It is also important, of course, that they exit with a zero status if everything went well.

Additionally, packages interacting with users using `debconf` in the `postinst` script should install a `config` script as a package metadata file. See Prompting in maintainer scripts for details.

When a package is upgraded a combination of the scripts from the old and new packages is called during the upgrade procedure. If your scripts are going to be at all complicated you need to be aware of this, and may need to check the arguments to your scripts.

Broadly speaking the `preinst` is called before (a particular version of) a package is unpacked, and the `postinst` afterwards; the `prerm` before (a version of) a package is removed and the `postrm` afterwards.

Programs called from maintainer scripts should not normally have a path prepended to them. Before installation is started, the package management system checks to see if the programs `ldconfig`, `start-stop-daemon`, and `update-rc.d` can be found via the `PATH` environment variable. Those programs, and any other program that one would expect to be in the `PATH`, should thus be invoked without an absolute pathname. Maintainer scripts should also not reset the `PATH`, though they might choose to modify it by prepending or appending package-specific directories. These considerations really apply to all shell scripts.

6.2. Maintainer scripts idempotency

It is necessary for the error recovery procedures that the scripts be idempotent. This means that if it is run successfully, and then it is called again, it doesn't bomb out or cause any harm, but just ensures that everything is the way it ought to be. If the first call failed, or aborted half way through for some reason, the second call should merely do the things that were left undone the first time, if any, and exit with a success status if everything is OK. [1]

6.3. Controlling terminal for maintainer scripts

Maintainer scripts are not guaranteed to run with a controlling terminal and may not be able to interact with the user. They must be able to fall back to noninteractive behavior if no controlling terminal is available. Maintainer scripts that prompt via a program conforming to the Debian Configuration Management Specification (see Prompting in maintainer scripts) may assume that program will handle falling back to noninteractive behavior.

For high-priority prompts without a reasonable default answer, maintainer scripts may abort if there is no controlling terminal. However, this situation should be avoided if at all possible, since it prevents automated or unattended installs. In most cases, users will consider this to be a bug in the package.

6.4. Exit status

Each script must return a zero exit status for success, or a nonzero one for failure, since the package management system looks for the exit status of these scripts and determines what action to take next based on that datum.

6.5. Summary of ways maintainer scripts are called

What follows is a summary of all the ways in which maintainer scripts may be called along with what facilities those scripts may rely on being available at that time. Script names preceded by new- are the scripts from the new version of a package being installed, upgraded to, or downgraded to. Script names preceded by old- are the scripts from the old version of a package that is being upgraded from or downgraded from.

The `preinst` script may be called in the following ways:

`new-preinst` install

`new-preinst` install *old-version new-version*

`new-preinst` upgrade *old-version new-version*

> The package will not yet be unpacked, so the `preinst` script cannot rely on any files included in its package. Only essential packages and pre-dependencies (`Pre-Depends`) may be assumed to be available. Pre-dependencies will have been configured at least once, but at the time the `preinst` is called they may only be in an “Unpacked” or “Half-Configured” state if a previous version of the pre-dependency was completely configured and has not been removed since then.

`old-preinst` abort-upgrade *new-version*

> Called during error handling of an upgrade that failed after unpacking the new package because the `postrm upgrade` action failed. The unpacked files may be partly from the new version or partly missing, so the script cannot rely on files included in the package. Package dependencies may not be available. Pre-dependencies will be at least “Unpacked” following the same rules as above, except they may be only “Half-Installed” if an upgrade of the pre-dependency failed. [2]

The `postinst` script may be called in the following ways:

`postinst` configure *most-recently-configured-version*

> The files contained in the package will be unpacked. All package dependencies will at least be "Unpacked". If there are no circular dependencies involved, all package dependencies will be configured. For behavior in the case of circular dependencies, see the discussion in Binary Dependencies - Depends, Recommends, Suggests, Enhances, Pre-Depends.

`old-postinst` abort-upgrade *new-version*

`conflictor's-postinst` abort-remove in-favour *package new-version*

`postinst` abort-remove

`deconfigured's-postinst` abort-deconfigure in-favour *failed-install-package version* [removing conflicting-package version]

> The files contained in the package will be unpacked. All package dependencies will at least be "Half-Installed" and will have previously been configured and not removed. However, dependencies may not be configured or even fully unpacked in some error situations. [3] The `postinst` should still attempt any actions for which its dependencies are required, since they will normally be available, but consider the correct error handling approach if those actions fail. Aborting the `postinst` action if commands or facilities from the package dependencies are not available is often the best approach.

The `prerm` script may be called in the following ways:

`prerm` remove

`old-prerm` upgrade *new-version*

`conflictor's-prerm` remove in-favour package *new-version*

`deconfigured's-prerm` deconfigure in-favour *package-being-installed version* [removing conflicting-package version]

> The package whose `prerm` is being called will be at least "Half-Installed". All package dependencies will at least be "Half-Installed" and will have previously been configured and not removed. If there was no error, all dependencies will at least be "Unpacked", but these actions may be called in various error states where dependencies are only "Half-Installed" due to a partial upgrade.

`new-prerm` failed-upgrade *old-version new-version*

> Called during error handling when `prerm upgrade` fails. The new package will not yet be unpacked, and all the same constraints as for `preinst upgrade` apply.

The `postrm` script may be called in the following ways:

`postrm` remove

`postrm` purge

`old-postrm` upgrade *new-version*

`disappearer's-postrm` disappear overwriter *overwriter-version*

> The `postrm` script is called after the package's files have been removed or replaced. The package whose `postrm` is being called may have previously been deconfigured and only be "Unpacked", at which point subsequent package changes do not consider its dependencies. Therefore, all `postrm` actions must only rely on essential packages and must gracefully skip any actions that require the package's dependencies if those dependencies are unavailable. [4]

`new-postrm` failed-upgrade *old-version new-version*

> Called when the old `postrm upgrade` action fails. The new package will be unpacked, but only essential packages and pre-dependencies can be relied on. Pre-dependencies will either be configured or will be "Unpacked" or "Half-Configured" but previously had been configured and was never removed.

`new-postrm` abort-install

`new-postrm` abort-install *old-version new-version*

`new-postrm` abort-upgrade *old-version new-version*

> Called before unpacking the new package as part of the error handling of `preinst` failures. May assume the same state as `preinst` can assume.

6.6. Details of unpack phase of installation or upgrade

The procedure on installation/upgrade/overwrite/disappear (i.e., when running `dpkg --unpack`, or the unpack stage of `dpkg --install`) is as follows. [5] In each case, if a major error occurs (unless listed below) the actions are, in general, run backwards - this means that the maintainer scripts are run with different arguments in reverse order. These are the "error unwind" calls listed below.

1. Notify the currently installed package:
    1. If a version of the package is already "Installed", call

        ```
        old-prerm upgrade new-version
        ```

    2. If the script runs but exits with a non-zero exit status, `dpkg` will attempt:

        ```
        new-prerm failed-upgrade old-version new-version
        ```

        If this works, the upgrade continues. If this does not work, the error unwind:

        ```
        old-postinst abort-upgrade new-version
        ```

If this works, then the *old-version* is “Installed”, if not, the old version is in a “Half-Configured” state.

2. If a “conflicting” package is being removed at the same time, or if any package will be broken (due to `Breaks`):

    1. If `--auto-deconfigure` is specified, call, for each package to be deconfigured due to `Breaks`:

        ```
        deconfigured's-prerm deconfigure \
            in-favour package-being-installed version
        ```

        Error unwind:

        ```
        deconfigured's-postinst abort-deconfigure \
            in-favour package-being-installed-but-failed version
        ```

        The deconfigured packages are marked as requiring configuration, so that if `--install` is used they will be configured again if possible.

    2. If any packages depended on a conflicting package being removed and `--auto-deconfigure` is specified, call, for each such package:

        ```
        deconfigured's-prerm deconfigure \
           in-favour package-being-installed version \
           removing conflicting-package version
        ```

        Error unwind:

        ```
        deconfigured's-postinst abort-deconfigure \
            in-favour package-being-installed-but-failed version \
            removing conflicting-package version
        ```

        The deconfigured packages are marked as requiring configuration, so that if `--install` is used they will be configured again if possible.

    3. To prepare for removal of each conflicting package, call:

        ```
        conflictor's-prerm remove \
            in-favour package new-version
        ```

        Error unwind:

```
conflictor's-postinst abort-remove \
    in-favour package new-version
```

3. Run the `preinst` of the new package:

    1. If the package is being upgraded, call:

    ```
    new-preinst upgrade old-version new-version
    ```

    If this fails, we call:

    ```
    new-postrm abort-upgrade old-version new-version
    ```

        1. If that works, then

        ```
        old-postinst abort-upgrade new-version
        ```

        is called. If this works, then the old version is in an “Installed” state, or else it is left in an “Unpacked” state.

        2. If it fails, then the old version is left in an “Half-Installed” state.

    2. Otherwise, if the package had some configuration files from a previous version installed (i.e., it is in the “Config-Files” state):

    ```
    new-preinst install old-version new-version
    ```

    Error unwind:

    ```
    new-postrm abort-install old-version new-version
    ```

    If this fails, the package is left in a “Half-Installed” state, which requires a reinstall. If it works, the packages is left in a “Config-Files” state.

    3. Otherwise (i.e., the package was completely purged):

    ```
    new-preinst install
    ```

    Error unwind:

    ```
    new-postrm abort-install
    ```

If the error-unwind fails, the package is in a “Half-Installed” phase, and requires a reinstall. If the error unwind works, the package is in the “Not-Installed” state.

4. The new package’s files are unpacked, overwriting any that may be on the system already, for example any from the old version of the same package or from another package. Backups of the old files are kept temporarily, and if anything goes wrong the package management system will attempt to put them back as part of the error unwind.

   It is an error for a package to contain files which are on the system in another package, unless `Replaces` is used (see Overwriting files and replacing packages - Replaces).

   It is a more serious error for a package to contain a plain file or other kind of non-directory where another package has a directory (again, unless `Replaces` is used). This error can be overridden if desired using `--force-overwrite-dir`, but this is not advisable.

   Packages which overwrite each other’s files produce behavior which, though deterministic, is hard for the system administrator to understand. It can easily lead to “missing” programs if, for example, a package is unpacked which overwrites a file from another package, and is then removed again. [6]

   A directory will never be replaced by a symbolic link to a directory or vice versa; instead, the existing state (symlink or not) will be left alone and `dpkg` will follow the symlink if there is one.

5. If the package is being upgraded:

   1. Call:

      ```
      old-postrm upgrade new-version
      ```

   2. If this fails, `dpkg` will attempt:

      ```
      new-postrm failed-upgrade old-version new-version
      ```

      If this works, installation continues. If not, Error unwind:

      ```
      old-preinst abort-upgrade new-version
      ```

      If this fails, the old version is left in a “Half-Installed” state. If it works, dpkg now calls:

      ```
      new-postrm abort-upgrade old-version new-version
      ```

      If this fails, the old version is left in a “Half-Installed” state. If it works, dpkg now calls:

      ```
      old-postinst abort-upgrade new-version
      ```

If this fails, the old version is in an “Unpacked” state.

This is the point of no return. If `dpkg` gets this far, it won’t back off past this point if an error occurs. This will leave the package in a fairly bad state, which will require a successful re-installation to clear up, but it’s when `dpkg` starts doing things that are irreversible.

6. Any files which were in the old version of the package but not in the new are removed.
7. The new file list replaces the old.
8. The new maintainer scripts replace the old.
9. Any packages all of whose files have been overwritten during the installation, and which aren’t required for dependencies, are considered to have been removed. For each such package
    1. `dpkg` calls:

        ```
        disappearer's-postrm disappear \
            overwriter overwriter-version
        ```

    2. The package’s maintainer scripts are removed.
    3. It is noted in the status database as being in a sane state, namely “Not-Installed” (any conffiles it may have are ignored, rather than being removed by `dpkg`). Note that disappearing packages do not have their prerm called, because `dpkg` doesn’t know in advance that the package is going to vanish.
10. Any files in the package we’re unpacking that are also listed in the file lists of other packages are removed from those lists. (This will lobotomize the file list of the “conflicting” package if there is one.)
11. The backup files made during installation, above, are deleted.
12. The new package’s status is now sane, and recorded as “Unpacked”.

    Here is another point of no return: if the conflicting package’s removal fails we do not unwind the rest of the installation. The conflicting package is left in a half-removed limbo.
13. If there was a conflicting package we go and do the removal actions (described below), starting with the removal of the conflicting package’s files (any that are also in the package being unpacked have already been removed from the conflicting package’s file list, and so do not get removed now).

6.7. Details of configuration

When we configure a package (this happens with `dpkg --install` and `dpkg --configure`), we first update any `conffile`s and then call:

```
postinst configure most-recently-configured-version
```

No attempt is made to unwind after errors during configuration. If the configuration fails, the package is in a “Half-Configured” state, and an error message is generated.

If there is no most recently configured version `dpkg` will pass a null argument. [7]

6.8. Details of removal and/or configuration purging

1. ```
   prerm remove
   ```

   If prerm fails during replacement due to conflict

   ```
   conflictor's-postinst abort-remove \
       in-favour package new-version
   ```

   Or else we call:

   ```
   postinst abort-remove
   ```

   If this fails, the package is in a “Half-Configured” state, or else it remains “Installed”.

2. The package’s files are removed (except `conffile` s).

3. ```
   postrm remove
   ```

   If it fails, there’s no error unwind, and the package is in an “Half-Installed” state.

4. All the maintainer scripts except the `postrm` are removed.

   If we aren’t purging the package we stop here. Note that packages which have no `postrm` and no `conffile` s are automatically purged when removed, as there is no difference except for the `dpkg` status.

5. The `conffile` s and any backup files (`~` -files, `#*#` files, `%` -files, `.dpkg-{old,new,tmp}` , etc.) are removed.

6. ```
   postrm purge
   ```

   If this fails, the package remains in a “Config-Files” state.

7. The package’s file list is removed.
   ```
