DPM Specification Files

To build a package, you'll need a specification file.

A `spec file` is a file that dpm is told to create a package with. Think of it as a configuration file for the build process for DPM packages.

For an example, imagine you're building a package. You've populated your directory tree and you're ready to make the package. You'd run something like this:

```
dpm buildpkg --spec=./MyApp.SPEC --contents-path=./contents
```

This is telling DPM to pass the spec file path and pre-populated contents tree to the `buildpkg` module. The package creation module would read the spec file's contents and generate a package based on those contents. This allows repeatable package builds.

On Source Compilation: DPM doesn't do it.

Bear in mind that the use of the word `build` here is not meant to convey source compilation. The compilation of binaries and the packaging of pre-compiled binaries are wholly separate processes that never should have been merged into a single unit of work, and DPM solves this problem by not attempting to include source compilation in any of its features — it assumes that the packager has already done this.

There is room in the design for support for a `SDPM` file format, but this is not a priority for the system as a whole — merely allowance in the design for the capability of handling this, which will be elaborated on in a dedicated section for SDPM compilation.

DPM Spec File Structure

The DPM spec file is a structured file with headers for each section. Each section is intended to be used by DPM during the buildpkg process to populate:

- the package metadata
- the optional signature archive
- the DPM hooks contents

Syntax

Syntax is very simple. The file is read line by line and contains three types of data:

- Comments
- Section Start/Stop declarations
- Section Content

Sections have a reserved name that match the corresponding filename in the DPM Package specification.

Comments

Commented lines start with a # symbol.

Sections

Each field represents a data value that the package will need.

To start a section, the maintainer would have:

```
%section metadata.NAME
```

To likewise end a section:

```
%section_end metadata.NAME
```

Whatever is between these values will become the value in the resulting packages’ metadata. In the event that a section is multiline but the value is a single-line string, only the first line will be read.

Whitespace between sections will not be read.

Environment Variables

Environment variables can be interpolated in DPM Spec files. Their notation is a preceding `'[['` and proceeding `']]'` with a single space as padding around the variable name.

So, if you set an environment variable prior to reading it DPM will interpolate it in the SPEC file:

```
yourshellprompt# PKG_NAME="Apache2"

... spec file ...
...
%section metadata.NAME
[[ PKG_NAME ]]
%section_end metadata.NAME
```

Then `[[ PKG_NAME ]]` will be replaced with the string `Apache2`.

If you would like to avoid negation of a variable name for a section declaration for whatever reason, such as when working with layers of templating engines, you can wrap literal statements with `[%literal]` and `[%/literal]` indicators.

```
%section hooks.POST-INSTALL
echo "adding [%literal] [[ PKG_NAME ]] to %section hooks.POST-INSTALL [%/literal]"
%section_end hooks.POST-INSTALL
```

In this example, the echoed text will be:

```
adding [[ PKG_NAME ]] to %section hooks.POST-INSTALL
```

This allows the maintainer to do layered templating approaches of large sets of packages.

If a variable is referenced but is empty or not defined, DPM will refuse to build the package and cite usage of an uninitialized variable as the reason.

Reserved Names

Reserved names for sections are:

hooks

1. hooks.POST-INSTALL
2. hooks.POST-INSTALL_ROLLBACK
3. hooks.POST-REMOVE
4. hooks.POST-REMOVE_ROLLBACK
5. hooks.POST-UPDATE
6. hooks.POST-UPDATE_ROLLBACK
7. hooks.POST-INSTALL
8. hooks.POST-INSTALL_ROLLBACK
9. hooks.PRE-REMOVE
10. hooks.PRE-REMOVE_ROLLBACK
11. hooks.PRE-UPDATE
12. hooks.PRE-UPDATE_ROLLBACK

metadata

1. metadata.ARCHITECTURE
2. metadata.AUTHOR
3. metadata.MAINTAINER
4. metadata.DEPENDENCIES
5. metadata.DESCRIPTION
6. metadata.CONTENTS_MANIFEST_DIGEST
7. metadata.LICENSE
8. metadata.NAME
9. metadata.PROVIDES
10. metadata.REPLACES
11. metadata.SOURCE
12. metadata.CHANGELOG
13. metadata.VERSION

On Signatures

You will likely immediately notice that there is no mention of signatures. That’s because building a package and signing a package are different things, and signing is handled by a different DPM module prior to zipping up the package after the metadata is generated by the SPEC file. Once generated, the resulting directory structure is able to be finalized into a packaged, or optionally signed first and then finalized.

Example DPM Spec File

```
# This is an example DPM Spec file.
# This is a comment.
# Created by Chris Punches on 2025-02-22.

# Whitespace is ignored between sections.
# Comments are processed as ignored between sections. Otherwise, they're included in the section data.

# The name of the software we're creating.
%section metadata.NAME
MyApp
%section_end metadata.NAME

# The version of the software.
%section metadata.VERSION
0.1.0
%section_end metadata.VERSION

# The author of the software.
%section metadata.AUTHOR
Chris Punches <chris.punches@silogroup.org>
%section_end metadata.AUTHOR

# Source of the software.  Usually a URL where an archive
# of the source code can be downloaded.
%section metadata.SOURCE
https://source.silogroup.org/Dark-Horse-Linux/pyrois
%section_end metadata.SOURCE

# The package maintainer.
%section metadata.MAINTAINER
Chris Punches <chris.punches@silogroup.org>
%section_end metadata.MAINTAINER

# Changelog - the changelog for THIS PACKAGE
```

```
%section metadata.CHANGELOG
*2025-02-23 Chris Punches chris.punches@silogroup.org
First draft.
%section_end metadata.CHANGELOG

# The license (used for compliance tracking).
%section metadata.LICENSE
A/GPL 3.0
%section_end metadata.LICENSE

# The architecture.  For this example, we'll supply it as an
# environment variable.
%section metadata.ARCHITECTURE
[[ target_architecture ]]
%section_end metadata.ARCHITECTURE

# Dependencies -- this is multiline.
# These are packages, and their versions, that must be present
# in the DPMDB in order for this package to meet its required
# dependencies.
%section metadata.DEPENDENCIES
glibc >= 2.21
glibc <= 2.39
libstdc++ > 0
%section_end metadata.DEPENDENCIES

# Provides -- these are package aliases for meeting
# interchangeable dependencies.  Left blank here.
%section metadata.PROVIDES
%section_end metadata.PROVIDES

# Replaces -- these are packages this package is meant to
replace.
%section metadata.REPLACES
%section_end metadata.REPLACES

# hooks -- these are triggered when the maintainer populates
them and these are assumed to be shell scripts (/bin/sh not
necessarily /bin/bash).
%section hooks.POST-INSTALL
%section_end hooks.POST-INSTALL
```

```
%section hooks.POST-INSTALL_ROLLBACK
%section_end hooks.POST-INSTALL_ROLLBACK

%section hooks.POST-REMOVE
%section_end hooks.POST-REMOVE
%section hooks.POST-REMOVE_ROLLBACK
%section_end hooks.POST-REMOVE_ROLLBACK

%section hooks.POST-UPDATE
systemctl start MyAppD
%section_end hooks.POST-UPDATE
%section hooks.POST-UPDATE_ROLLBACK
%section_end hooks.POST-UPDATE_ROLLBACK

%section hooks.PRE-INSTALL
%section_end hooks.PRE-INSTALL
%section hooks.PRE-INSTALL_ROLLBACK
%section_end hooks.PRE-INSTALL_ROLLBACK

%section hooks.PRE-REMOVE
systemctl stop MyAppD
%section_end hooks.PRE-REMOVE
%section hooks.PRE-REMOVE_ROLLBACK
systemctl start MyAppD
%section_end hooks.PRE-REMOVE_ROLLBACK

%section hooks.PRE-UPDATE
systemctl stop MyAppD
%section_end hooks.PRE-UPDATE
%section hooks.PRE-UPDATE_ROLLBACK
systemctl start MyAppD
%section_end hooks.PRE-UPDATE_ROLLBACK
```

The spec file format is simple and easy for a human to edit, and this can be populated with a web form in just a few seconds, or generated from a template.