Red Hat Enterprise Linux 8

Packaging and distributing software

A guide to packaging and distributing software in Red Hat Enterprise Linux 8

Last Updated: 2022-01-26

Red Hat Enterprise Linux 8 Packaging and distributing software

A guide to packaging and distributing software in Red Hat Enterprise Linux 8

Legal Notice

Copyright © 2022 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This document describes how to package software into an RPM. It also shows how to prepare source code for packaging, and explains selected advanced packaging scenarios, such as packaging Python projects or RubyGems into RPM.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE 5

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION 6

CHAPTER 1. GETTING STARTED WITH RPM PACKAGING 7

CHAPTER 2. PREPARING SOFTWARE FOR RPM PACKAGING 8
2.1. WHAT SOURCE CODE IS 8
2.2. HOW PROGRAMS ARE MADE 9
2.2.1. Natively Compiled Code 9
2.2.2. Interpreted Code 9
2.2.2.1. Raw-interpreted programs 9
2.2.2.2. Byte-compiled programs 10
2.3. BUILDING SOFTWARE FROM SOURCE 10
2.4. BUILDING SOFTWARE FROM NATIVELY COMPILED CODE 10
2.4.1. Manual building 10
2.4.2. Automated building 11
2.5. INTERPRETING CODE 11
2.5.1. Byte-compiling code 12
2.5.2. Raw-interpreting code 13
2.6. PATCHING SOFTWARE 13
2.7. ARBITRARY ARTIFACTS 15
2.8. PLACING ARBITRARY ARTIFACTS IN THE SYSTEM USING THE INSTALL COMMAND 15
2.9. PLACING ARBITRARY ARTIFACTS IN THE SYSTEM USING THE MAKE INSTALL COMMAND 16
2.10. PREPARING SOURCE CODE FOR PACKAGING 17
2.11. PUTTING SOURCE CODE INTO TARBALL 17

CHAPTER 3. PACKAGING SOFTWARE 21
3.1. RPM PACKAGES 21
Types of RPM packages 21
3.2. LISTING RPM PACKAGING TOOL'S UTILITIES 21
3.3. SETTING UP RPM PACKAGING WORKSPACE 21
3.4. WHAT A SPEC FILE IS 22
3.4.1. Preamble Items 22
3.4.2. Body Items 24
3.4.3. Advanced items 25
3.5. BUILDROOTS 25
3.6. RPM MACROS 26
3.7. WORKING WITH SPEC FILES 26
3.8. CREATING A NEW SPEC FILE WITH RPMDEV-NEWSPEC 27
3.9. MODIFYING AN ORIGINAL SPEC FILE FOR CREATING RPMS 28
3.10. AN EXAMPLE SPEC FILE FOR A PROGRAM WRITTEN IN BASH 30
3.11. AN EXAMPLE SPEC FILE FOR A PROGRAM WRITTEN IN PYTHON 31
3.12. AN EXAMPLE SPEC FILE FOR A PROGRAM WRITTEN IN C 33
3.13. BUILDING RPMS 34
3.14. BUILDING SOURCE RPMS 34
3.15. REBUILDING A BINARY RPM FROM A SOURCE RPM 35
3.16. BUILDING A BINARY RPM FROM THE SPEC FILE 36
3.17. BUILDING BINARY RPMS FROM SOURCE RPMS 36
3.18. CHECKING RPMS FOR SANITY 37
3.19. CHECKING BELLO FOR SANITY 37
3.19.1. Checking the bello SPEC File 37

3.19.2. Checking the bello binary RPM 38
3.20. CHECKING PELLO FOR SANITY 38
3.20.1. Checking the pello SPEC File 38
3.20.2. Checking the pello binary RPM 39
3.21. CHECKING CELLO FOR SANITY 40
3.21.1. Checking the cello SPEC File 40
3.21.2. Checking the cello binary RPM 40
3.22. LOGGING RPM ACTIVITY TO SYSLOG 41
3.23. EXTRACTING RPM CONTENT 41

CHAPTER 4. ADVANCED TOPICS **43**
4.1. SIGNING PACKAGES 43
4.1.1. Creating a GPG key 43
4.1.2. Adding a signature to an already existing package 43
4.1.3. Checking the signatures of a package with multiple signatures 44
4.1.4. A practical example of adding a signature to an already existing package 44
4.1.5. Replacing the signature on an already existing package 44
4.1.6. Signing a package at build-time 45
4.2. MORE ON MACROS 45
4.2.1. Defining your own macros 46
4.2.2. Using the %setup macro 46
4.2.2.1. Using the %setup -q macro 47
4.2.2.2. Using the %setup -n macro 47
4.2.2.3. Using the %setup -c macro 47
4.2.2.4. Using the %setup -D and %setup -T macros 48
4.2.2.5. Using the %setup -a and %setup -b macros 48
4.2.3. Common RPM macros in the %files section 48
4.2.4. Displaying the built-in macros 49
4.2.5. RPM distribution macros 49
4.2.5.1. Creating custom macros 50
4.3. EPOCH, SCRIPTLETS AND TRIGGERS 50
4.3.1. The Epoch directive 51
4.3.2. Scriptlets 51
4.3.2.1. Scriptlets directives 51
4.3.2.2. Turning off a scriptlet execution 52
4.3.2.3. Scriptlets macros 52
4.3.3. The Triggers directives 53
4.3.4. Using non-shell scripts in a SPEC file 54
4.4. RPM CONDITIONALS 55
4.4.1. RPM conditionals syntax 55
4.4.2. RPM conditionals examples 55
4.4.2.1. The %if conditionals 56
4.4.2.2. Specialized variants of %if conditionals 56
4.4.2.2.1. The %ifarch conditional 56
4.4.2.2.2. The %ifnarch conditional 56
4.4.2.2.3. The %ifos conditional 57
4.5. PACKAGING PYTHON 3 RPMS 57
4.5.1. SPEC file description for a Python package 57
4.5.2. Common macros for Python 3 RPMs 59
4.5.3. Automatic provides for Python RPMs 60
4.6. HANDLING INTERPRETER DIRECTIVES IN PYTHON SCRIPTS 60
4.6.1. Modifying interpreter directives in Python scripts 60
4.6.2. Changing /usr/bin/python3 interpreter directives in your custom packages 61

4.7. RUBYGEMS PACKAGES 61
4.7.1. What RubyGems are 62
4.7.2. How RubyGems relate to RPM 62
4.7.3. Creating RPM packages from RubyGems packages 62
4.7.3.1. RubyGems SPEC file conventions 62
Macros 63
4.7.3.2. RubyGems SPEC file example 64
4.7.3.3. Converting RubyGems packages to RPM SPEC files with gem2rpm 65
4.7.3.3.1. Installing gem2rpm 65
4.7.3.3.2. Displaying all options of gem2rpm 65
4.7.3.3.3. Using gem2rpm to covert RubyGems packages to RPM SPEC files 65
4.7.3.3.4. Editing gem2rpm templates 66
4.8. HOW TO HANDLE RPM PACKAGES WITH PERLS SCRIPTS 67
4.8.1. Common Perl-related dependencies 67
4.8.2. Using a specific Perl module 67
4.8.3. Limiting a package to a specific Perl version 68
4.8.4. Ensuring that a package uses the correct Perl interpreter 68

CHAPTER 5. NEW FEATURES IN RHEL 8 69
5.1. SUPPORT FOR WEAK DEPENDENCIES 69
5.1.1. Introduction to Weak dependencies policy 69
5.1.1.1. Weak dependencies 69
Conditions of use 69
Use cases 69
5.1.1.2. Hints 70
Package Preference 70
5.1.1.3. Forward and Backward dependencies 70
5.2. SUPPORT FOR BOOLEAN DEPENDENCIES 71
5.2.1. Boolean dependencies syntax 71
5.2.2. Boolean operators 71
5.2.3. Nesting 72
5.2.4. Semantics 73
5.2.4.1. Understanding the output of the if operator 73
5.3. SUPPORT FOR FILE TRIGGERS 74
5.3.1. File triggers syntax 74
5.3.2. Examples of File triggers syntax 75
5.3.3. File triggers types 75
5.3.3.1. Executed once per package File triggers 75
%filetriggerin 75
%filetriggerun 75
%filetriggerpostun 75
5.3.3.2. Executed once per transaction File triggers 76
%transfiletriggerin 76
%transfiletriggerun 76
%transfiletriggerpostun 76
5.3.4. Example use of File triggers in glibc 76
5.4. STRICTER SPEC PARSER 77
5.5. SUPPORT FOR FILES ABOVE 4 GB 77
5.5.1. 64-bit RPM tags 77
5.5.1.1. Using 64-bit tags on command line 77
5.6. OTHER FEATURES 77

CHAPTER 6. ADDITIONAL RESOURCES ABOUT RPM PACKAGING 79

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your input on our documentation. Please let us know how we could make it better. To do so:

- For simple comments on specific passages:
  1. Make sure you are viewing the documentation in the *Multi-page HTML* format. In addition, ensure you see the **Feedback** button in the upper right corner of the document.
  2. Use your mouse cursor to highlight the part of text that you want to comment on.
  3. Click the **Add Feedback** pop-up that appears below the highlighted text.
  4. Follow the displayed instructions.
- For submitting more complex feedback, create a Bugzilla ticket:
  1. Go to the Bugzilla website.
  2. As the Component, use **Documentation**.
  3. Fill in the **Description** field with your suggestion for improvement. Include a link to the relevant part(s) of documentation.
  4. Click **Submit Bug**.

CHAPTER 1. GETTING STARTED WITH RPM PACKAGING

The RPM Package Manager (RPM) is a package management system that runs on Red Hat Enterprise Linux, CentOS, and Fedora. You can use RPM to distribute, manage, and update software that you create for any of the operating systems mentioned above.

The RPM package management system brings several advantages over distribution of software in conventional archive files.

RPM enables you to:

- Install, reinstall, remove, upgrade and verify packages with standard package management tools, such as YUM or PackageKit.
- Use a database of installed packages to query and verify packages.
- Use metadata to describe packages, their installation instructions, and other package parameters.
- Package software sources, patches and complete build instructions into source and binary packages.
- Add packages to YUM repositories.
- Digitally sign your packages by using GNU Privacy Guard (GPG) signing keys.

CHAPTER 2. PREPARING SOFTWARE FOR RPM PACKAGING

This section explains how to prepare software for RPM packaging. To do so, knowing how to code is not necessary. However, you need to understand the basic concepts, such as What source code is and How programs are made.

2.1. WHAT SOURCE CODE IS

This part explains what source code is and shows example source codes of a program written in three different programming languages.

Source code is human-readable instructions to the computer, which describe how to perform a computation. Source code is expressed using a programming language.

This document features three versions of the **Hello World** program written in three different programming languages:

- Hello World written in bash
- Hello World written in Python
- Hello World written in C

Each version is packaged differently.

These versions of the **Hello World** program cover the three major use cases of an RPM packager.

Example 2.1. Hello World written in bash

The *bello* project implements **Hello World** in bash. The implementation only contains the **bello** shell script. The purpose of the program is to output **Hello World** on the command line.

The **bello** file has the following syntax:

```
#!/bin/bash

printf "Hello World\n"
```

Example 2.2. Hello World written in Python

The *pello* project implements **Hello World** in Python. The implementation only contains the **pello.py** program. The purpose of the program is to output **Hello World** on the command line.

The **pello.py** file has the following syntax:

```
#!/usr/bin/python3

print("Hello World")
```

Example 2.3. Hello World written in C

> The *cello* project implements **Hello World** in C. The implementation only contains the **cello.c** and the **Makefile** files, so the resulting **tar.gz** archive will have two files apart from the **LICENSE** file.
>
> The purpose of the program is to output **Hello World** on the command line.
>
> The **cello.c** file has the following syntax:
>
> ```
> #include <stdio.h>
>
> int main(void) {
>  printf("Hello World\n");
>  return 0;
> }
> ```

2.2. HOW PROGRAMS ARE MADE

Methods of conversion from human-readable source code to machine code (instructions that the computer follows to execute the program) include the following:

- The program is natively compiled.
- The program is interpreted by raw interpreting.
- The program is interpreted by byte compiling.

2.2.1. Natively Compiled Code

Natively compiled software is software written in a programming language that compiles to machine code with a resulting binary executable file. Such software can be run stand-alone.

RPM packages built this way are architecture-specific.

If you compile such software on a computer that uses a 64-bit (x86_64) AMD or Intel processor, it does not execute on a 32-bit (x86) AMD or Intel processor. The resulting package has architecture specified in its name.

2.2.2. Interpreted Code

Some programming languages, such as bash or Python, do not compile to machine code. Instead, their programs' source code is executed step by step, without prior transformations, by a Language Interpreter or a Language Virtual Machine.

Software written entirely in interpreted programming languages is not architecture-specific. Hence, the resulting RPM Package has the **noarch** string in its name.

Interpreted languages are either Raw-interpreted programs or Byte-compiled programs. These two types differ in program build process and in packaging procedure.

2.2.2.1. Raw-interpreted programs

Raw-interpreted language programs do not need to be compiled and are directly executed by the interpreter.

2.2.2.2. Byte-compiled programs

Byte-compiled languages need to be compiled into byte code, which is then executed by the language virtual machine.

NOTE

Some languages offer a choice: they can be raw-interpreted or byte-compiled.

2.3. BUILDING SOFTWARE FROM SOURCE

For software written in compiled languages, the source code goes through a build process, producing machine code. This process, commonly called compiling or translating, varies for different languages. The resulting built software can be run, which makes the computer perform the task specified by the programmer.

For software written in raw interpreted languages, the source code is not built, but executed directly.

For software written in byte-compiled interpreted languages, the source code is compiled into byte code, which is then executed by the language virtual machine.

The following subchapters describe how to build software from source code.

2.4. BUILDING SOFTWARE FROM NATIVELY COMPILED CODE

This section shows how to build the **cello.c** program written in the C language into an executable.

cello.c

```
#include <stdio.h>

int main(void) {
    printf("Hello World\n");
    return 0;
}
```

2.4.1. Manual building

If you want to build the **cello.c** program manually, use this procedure:

Procedure

1. Invoke the C compiler from the GNU Compiler Collection to compile the source code into binary:

```
gcc -g -o cello cello.c
```

2. Execute the resulting output binary **cello**:

```
$ ./cello
Hello World
```

2.4.2. Automated building

Large-scale software commonly uses automated building that is done by creating the **Makefile** file and then running the GNU **make** utility.

If you want to use the automated building to build the **cello.c** program, use this procedure:

Procedure

1. To set up automated building, create the **Makefile** file with the following content in the same directory as **cello.c**.
   Makefile

   ```
   cello:
    gcc -g -o cello cello.c
   clean:
    rm cello
   ```

   Note that the lines under **cello:** and **clean:** must begin with a tab space.

2. To build the software, run the **make** command:

   ```
   $ make
   make: 'cello' is up to date.
   ```

3. Since there is already a build available, run the **make clean** command, and after run the **make** command again:

   ```
   $ make clean
   rm cello

   $ make
   gcc -g -o cello cello.c
   ```

   NOTE

   Trying to build the program after another build has no effect.

   ```
   $ make
   make: 'cello' is up to date.
   ```

4. Execute the program:

   ```
   $ ./cello
   Hello World
   ```

You have now compiled a program both manually and using a build tool.

2.5. INTERPRETING CODE

This section shows how to byte-compile a program written in Python and raw-interpret a program written in bash.

NOTE

In the two examples below, the **#!** line at the top of the file is known as a **shebang**, and is not part of the programming language source code.

The **shebang** enables using a text file as an executable: the system program loader parses the line containing the **shebang** to get a path to the binary executable, which is then used as the programming language interpreter. The functionality requires the text file to be marked as executable.

2.5.1. Byte-compiling code

This section shows how to compile the **pello.py** program written in Python into byte code, which is then executed by the Python language virtual machine.

Python source code can also be raw-interpreted, but the byte-compiled version is faster. Hence, RPM Packagers prefer to package the byte-compiled version for distribution to end users.

pello.py

```
#!/usr/bin/python3

print("Hello World")
```

Procedure for byte-compiling programs varies depending on the following factors:

- Programming language
- Language's virtual machine
- Tools and processes used with that language

NOTE

Python is often byte-compiled, but not in the way described here. The following procedure aims not to conform to the community standards, but to be simple. For real-world Python guidelines, see Software Packaging and Distribution.

Use this procedure to compile **pello.py** into byte code:

Procedure

1. Byte-compile the **pello.py** file:

```
$ python -m compileall pello.py

$ file pello.pyc
pello.pyc: python 2.7 byte-compiled
```

2. Execute the byte code in **pello.pyc**:

```
$ python pello.pyc
Hello World
```

2.5.2. Raw-interpreting code

This section shows how to raw-interpret the **bello** program written in the bash shell built-in language.

bello

```
#!/bin/bash

printf "Hello World\n"
```

Programs written in shell scripting languages, like *bash*, are raw-interpreted.

Procedure

- Make the file with source code executable and run it:

```
$ chmod +x bello
$ ./bello
Hello World
```

2.6. PATCHING SOFTWARE

In RPM packaging, instead of modifying the original source code, we keep it, and use patches on it.

A patch is a source code that updates other source code. It is formatted as a *diff*, because it represents what is different between two versions of the text. A *diff* is created using the **diff** utility, which is then applied to the source code using the patch utility.

NOTE

Software developers often use Version Control Systems such as git to manage their code base. Such tools provide their own methods of creating diffs or patching software.

This section explains how to patch the software.

The following example shows how to create a patch from the original source code using **diff**, and how to apply the patch using **patch**. Patching is used in a later section when creating an RPM.

This procedure shows how to create a patch from the original source code for **cello.c**.

Procedure

1. Preserve the original source code:

```
$ cp -p cello.c cello.c.orig
```

   The **-p** option is used to preserve mode, ownership, and timestamps.

2. Modify **cello.c** as needed:

```
#include <stdio.h>

int main(void) {
```

```
    printf("Hello World from my very first patch!\n");
    return 0;
}
```

3. Generate a patch using the **diff** utility:

```
$ diff -Naur cello.c.orig cello.c
--- cello.c.orig        2016-05-26 17:21:30.478523360 -0500
+ cello.c     2016-05-27 14:53:20.668588245 -0500
@@ -1,6 +1,6 @@
 #include<stdio.h>

 int main(void){
-    printf("Hello World!\n");
+    printf("Hello World from my very first patch!\n");
     return 0;
 }
\ No newline at end of file
```

Lines starting with a **-** are removed from the original source code and replaced with the lines that start with **+**.

Using the **Naur** options with the **diff** command is recommended because it fits the majority of usual use cases. However, in this particular case, only the **-u** option is necessary. Particular options ensure the following:

- **-N** (or **--new-file**) - Handles absent files as if they were empty files.
- **-a** (or **--text**) - Treats all files as text. As a result, the files that **diff** classifies as binaries are not ignored.
- **-u** (or **-U NUM** or **--unified[=NUM]**) - Returns output in the form of output NUM (default 3) lines of unified context. This is an easily readable format that allows fuzzy matching when applying the patch to a changed source tree.
- **-r** (or **--recursive**) - Recursively compares any subdirectories that are found.
  For more information on common arguments for the **diff** utility, see the **diff** manual page.

4. Save the patch to a file:

```
$ diff -Naur cello.c.orig cello.c > cello-output-first-patch.patch
```

5. Restore the original **cello.c**:

```
$ cp cello.c.orig cello.c
```

The original **cello.c** must be retained, because when an RPM is built, the original file is used, not the modified one. For more information, see Working with SPEC files.

The following procedure shows how to patch **cello.c** using **cello-output-first-patch.patch**, build the patched program, and run it.

Procedure

1. Redirect the patch file to the **patch** command:

```
$ patch < cello-output-first-patch.patch
patching file cello.c
```

2. Check that the contents of **cello.c** now reflect the patch:

```
$ cat cello.c
#include<stdio.h>

int main(void){
    printf("Hello World from my very first patch!\n");
    return 1;
}
```

3. Build and run the patched **cello.c**:

```
$ make clean
rm cello

$ make
gcc -g -o cello cello.c

$ ./cello
Hello World from my very first patch!
```

2.7. ARBITRARY ARTIFACTS

Unix-like systems use the Filesystem Hierarchy Standard (FHS) to specify a directory suitable for a particular file.

Files installed from the RPM packages are placed according to FHS. For example, an executable file should go into a directory that is in the system **$PATH** variable.

In the context of this documentation, an *Arbitrary Artifact* is anything installed from an RPM to the system. For RPM and for the system it can be a script, a binary compiled from the package's source code, a pre-compiled binary, or any other file.

The following sections describe two common ways of placing *Arbitrary Artifacts* in the system:

- Using the install command
- Using the make install command

2.8. PLACING ARBITRARY ARTIFACTS IN THE SYSTEM USING THE INSTALL COMMAND

Packagers often use the **install** command in cases when build automation tooling such as GNU make is not optimal; for example if the packaged program does not need extra overhead.

The **install** command is provided to the system by coreutils, which places the artifact to the specified directory in the file system with a specified set of permissions.

The following procedure uses the **bello** file that was previously created as the arbitrary artifact as a subject to this installation method.

Procedure

1. Run the **install** command to place the **bello** file into the **/usr/bin** directory with permissions common for executable scripts:

```
$ sudo install -m 0755 bello /usr/bin/bello
```

   As a result, **bello** is now located in the directory that is listed in the **$PATH** variable.

2. Execute **bello** from any directory without specifying its full path:

```
$ cd ~

$ bello
Hello World
```

2.9. PLACING ARBITRARY ARTIFACTS IN THE SYSTEM USING THE MAKE INSTALL COMMAND

Using the **make install** command is an automated way to install built software to the system. In this case, you need to specify how to install the arbitrary artifacts to the system in the **Makefile** that is usually written by the developer.

This procedure shows how to install a build artifact into a chosen location on the system.

Procedure

1. Add the **install** section to the **Makefile**:
   Makefile

```
cello:
 gcc -g -o cello cello.c

clean:
 rm cello

install:
 mkdir -p $(DESTDIR)/usr/bin
 install -m 0755 cello $(DESTDIR)/usr/bin/cello
```

   Note that the lines under **cello:**, **clean:**, and **install:** must begin with a tab space.

   NOTE

   The $(DESTDIR) variable is a GNU make built-in and is commonly used to specify installation to a directory different than the root directory.

   Now you can use **Makefile** not only to build software, but also to install it to the target system.

2. Build and install the **cello.c** program:

```
$ make
gcc -g -o cello cello.c
```

```
$ sudo make install
install -m 0755 cello /usr/bin/cello
```

As a result, **cello** is now located in the directory that is listed in the **$PATH** variable.

3. Execute **cello** from any directory without specifying its full path:

```
$ cd ~

$ cello
Hello World
```

2.10. PREPARING SOURCE CODE FOR PACKAGING

Developers often distribute software as compressed archives of source code, which are then used to create packages. RPM packagers work with a ready source code archive.

Software should be distributed with a software license.

This procedure uses the GPLv3 license text as an example content of the **LICENSE** file.

Procedure

1. Create a **LICENSE** file, and make sure that it includes the following content:

```
$ cat /tmp/LICENSE
This program is free software: you can redistribute it and/or modify it under the terms of the
GNU General Public License as published by the Free Software Foundation, either version 3
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;
without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this
program. If not, see http://www.gnu.org/licenses/.
```

Additional resources

- The code created in this section

2.11. PUTTING SOURCE CODE INTO TARBALL

This section describes how to put each of the three **Hello World** programs introduced in Section 2.1, "What source code is" into a gzip-compressed tarball, which is a common way to release the software to be later packaged for distribution.

Example 2.4. Putting the bello project into tarball

The *bello* project implements **Hello World** in bash. The implementation only contains the **bello** shell script, so the resulting **tar.gz** archive will have only one file apart from the **LICENSE** file.

This procedure shows how to prepare the *bello* project for distribution.

Prerequisites

Considering that this is version **0.1** of the program.

Procedure

1. Put all required files into a single directory:

```
$ mkdir /tmp/bello-0.1

$ mv ~/bello /tmp/bello-0.1/

$ cp /tmp/LICENSE /tmp/bello-0.1/
```

2. Create the archive for distribution and move it to the **~/rpmbuild/SOURCES/** directory, which is the default directory where the **rpmbuild** command stores the files for building packages:

```
$ cd /tmp/

$ tar -cvzf bello-0.1.tar.gz bello-0.1
bello-0.1/
bello-0.1/LICENSE
bello-0.1/bello

$ mv /tmp/bello-0.1.tar.gz ~/rpmbuild/SOURCES/
```

For more information about the example source code written in bash, see Hello World written in bash.

Example 2.5. Putting the pello project into tarball

The *pello* project implements **Hello World** in Python. The implementation only contains the **pello.py** program, so the resulting **tar.gz** archive will have only one file apart from the **LICENSE** file.

This procedure shows how to prepare the *pello* project for distribution.

Prerequisites

Considering that this is version **0.1.1** of the program.

Procedure

1. Put all required files into a single directory:

```
$ mkdir /tmp/pello-0.1.2

$ mv ~/pello.py /tmp/pello-0.1.2/

$ cp /tmp/LICENSE /tmp/pello-0.1.2/
```

2. Create the archive for distribution and move it to the **~/rpmbuild/SOURCES/** directory, which is the default directory where the **rpmbuild** command stores the files for building packages:

```
$ cd /tmp/
```

```
$ tar -cvzf pello-0.1.2.tar.gz pello-0.1.2
pello-0.1.2/
pello-0.1.2/LICENSE
pello-0.1.2/pello.py

$ mv /tmp/pello-0.1.2.tar.gz ~/rpmbuild/SOURCES/
```

For more information about the example source code written in Python, see Hello World written in Python.

Example 2.6. Putting the cello project into tarball

The *cello* project implements **Hello World** in C. The implementation only contains the **cello.c** and the **Makefile** files, so the resulting **tar.gz** archive will have two files apart from the **LICENSE** file.

Note that the **patch** file is not distributed in the archive with the program. The RPM Packager applies the patch when the RPM is built. The patch will be placed into the **~/rpmbuild/SOURCES/** directory alongside the **.tar.gz** archive.

This procedure shows how to prepare the *cello* project for distribution.

Prerequisites

Considering that this is version **1.0** of the program.

Procedure

1. Put all required files into a single directory:

```
$ mkdir /tmp/cello-1.0

$ mv ~/cello.c /tmp/cello-1.0/

$ mv ~/Makefile /tmp/cello-1.0/

$ cp /tmp/LICENSE /tmp/cello-1.0/
```

2. Create the archive for distribution and move it to the **~/rpmbuild/SOURCES/** directory, which is the default directory where the **rpmbuild** command stores the files for building packages:

```
$ cd /tmp/

$ tar -cvzf cello-1.0.tar.gz cello-1.0
cello-1.0/
cello-1.0/Makefile
cello-1.0/cello.c
cello-1.0/LICENSE

$ mv /tmp/cello-1.0.tar.gz ~/rpmbuild/SOURCES/
```

3. Add the patch:

```
$ mv ~/cello-output-first-patch.patch ~/rpmbuild/SOURCES/
```

For more information about the example source code written in C, see Hello World written in C.

CHAPTER 3. PACKAGING SOFTWARE

This section explains the basics of RPM packaging.

3.1. RPM PACKAGES

An RPM package is a file containing other files and their metadata (information about the files that are needed by the system).

Specifically, an RPM package consists of the **cpio** archive.

The **cpio** archive contains:

- Files
- RPM header (package metadata)
  The **rpm** package manager uses this metadata to determine dependencies, where to install files, and other information.

Types of RPM packages

There are two types of RPM packages. Both types share the file format and tooling, but have different contents and serve different purposes:

- Source RPM (SRPM)
  An SRPM contains source code and a SPEC file, which describes how to build the source code into a binary RPM. Optionally, the patches to source code are included as well.
- Binary RPM
  A binary RPM contains the binaries built from the sources and patches.

3.2. LISTING RPM PACKAGING TOOL'S UTILITIES

The following procedure describes how to list the utilities provided by the **rpmdevtools** package.

Prerequisites

- You installed the **rpmdevtools** package, which provides several utilities for packaging RPMs:

```
# yum install rpmdevtools
```

Procedure

- List RPM packaging tool's utilities:

```
$ rpm -ql rpmdevtools | grep bin
```

For more information on the above utilities, see their manual pages or help dialogs.

3.3. SETTING UP RPM PACKAGING WORKSPACE

This section describes how to set up a directory layout that is the RPM packaging workspace by using the **rpmdev-setuptree** utility.

Prerequisites

- You installed the **rpmdevtools** package, which provides several utilities for packaging RPMs:

```
# yum install rpmdevtools
```

Procedure

- Run the **rpmdev-setuptree** utility:

```
$ rpmdev-setuptree

$ tree ~/rpmbuild/
/home/user/rpmbuild/
|-- BUILD
|-- RPMS
|-- SOURCES
|-- SPECS
`-- SRPMS

5 directories, 0 files
```

The created directories serve the following purposes:

Directory	Purpose
BUILD	When packages are built, various **%buildroot** directories are created here. This is useful for investigating a failed build if the logs output do not provide enough information.
RPMS	Binary RPMs are created here, in subdirectories for different architectures, for example in subdirectories **x86_64** and **noarch**.
SOURCES	Here, the packager puts compressed source code archives and patches. The **rpmbuild** command looks for them here.
SPECS	The packager puts SPEC files here.
SRPMS	When **rpmbuild** is used to build an SRPM instead of a binary RPM, the resulting SRPM is created here.

3.4. WHAT A SPEC FILE IS

You can understand a SPEC file as a recipe that the **rpmbuild** utility uses to build an RPM. A SPEC file provides necessary information to the build system by defining instructions in a series of sections. The sections are defined in the *Preamble* and the *Body* part. The *Preamble* part contains a series of metadata items that are used in the *Body* part. The *Body* part represents the main part of the instructions.

3.4.1. Preamble Items

The table below presents some of the directives that are used frequently in the *Preamble* section of the RPM SPEC file.

Table 3.1. Items used in the *Preamble* section of the RPM SPEC file

SPEC Directive	Definition
Name	The base name of the package, which should match the SPEC file name.
Version	The upstream version number of the software.
Release	The number of times this version of the software was released. Normally, set the initial value to 1%{?dist}, and increment it with each new release of the package. Reset to 1 when a new **Version** of the software is built.
Summary	A brief, one-line summary of the package.
License	The license of the software being packaged.
URL	The full URL for more information about the program. Most often this is the upstream project website for the software being packaged.
Source0	Path or URL to the compressed archive of the upstream source code (unpatched, patches are handled elsewhere). This should point to an accessible and reliable storage of the archive, for example, the upstream page and not the packager's local storage. If needed, more SourceX directives can be added, incrementing the number each time, for example: Source1, Source2, Source3, and so on.
Patch	The name of the first patch to apply to the source code if necessary. The directive can be applied in two ways: with or without numbers at the end of Patch. If no number is given, one is assigned to the entry internally. It is also possible to give the numbers explicitly using Patch0, Patch1, Patch2, Patch3, and so on. These patches can be applied one by one using the %patch0, %patch1, %patch2 macro and so on. The macros are applied within the %prep directive in the *Body* section of the RPM SPEC file. Alternatively, you can use the %autopatch macro which automatically applies all patches in the order they are given in the SPEC file.
BuildArch	If the package is not architecture dependent, for example, if written entirely in an interpreted programming language, set this to **BuildArch: noarch**. If not set, the package automatically inherits the Architecture of the machine on which it is built, for example **x86_64**.
BuildRequires	A comma or whitespace-separated list of packages required for building the program written in a compiled language. There can be multiple entries of **BuildRequires**, each on its own line in the SPEC file.

SPEC Directive	Definition
Requires	A comma- or whitespace-separated list of packages required by the software to run once installed. There can be multiple entries of **Requires**, each on its own line in the SPEC file.
ExcludeArch	If a piece of software can not operate on a specific processor architecture, you can exclude that architecture here.
Conflicts	**Conflicts** are inverse to **Requires**. If there is a package matching **Conflicts**, the package cannot be installed independently on whether the **Conflict** tag is on the package that has already been installed or on a package that is going to be installed.
Obsoletes	This directive alters the way updates work depending on whether the **rpm** command is used directly on the command line or the update is performed by an updates or dependency solver. When used on a command line, RPM removes all packages matching obsoletes of packages being installed. When using an update or dependency resolver, packages containing matching **Obsoletes:** are added as updates and replace the matching packages.
Provides	If **Provides** is added to a package, the package can be referred to by dependencies other than its name.

The **Name**, **Version**, and **Release** directives comprise the file name of the RPM package. RPM package maintainers and system administrators often call these three directives **N-V-R** or **NVR**, because RPM package filenames have the **NAME-VERSION-RELEASE** format.

The following example shows how to obtain the **NVR** information for a specific package by querying the **rpm** command.

Example 3.1. Querying rpm to provide the NVR information for the bash package

```
# rpm -q bash
bash-4.4.19-7.el8.x86_64
```

Here, **bash** is the package name, **4.4.19** is the version, and **7.el8** is the release. The final marker is **x86_64**, which signals the architecture. Unlike the **NVR**, the architecture marker is not under direct control of the RPM packager, but is defined by the **rpmbuild** build environment. The exception to this is the architecture-independent **noarch** package.

3.4.2. Body Items

The items used in the **Body section** of the RPM SPEC file are listed in the table below.

Table 3.2. Items used in the Body section of the RPM SPEC file

SPEC Directive	Definition
%description	A full description of the software packaged in the RPM. This description can span multiple lines and can be broken into paragraphs.
%prep	Command or series of commands to prepare the software to be built, for example, unpacking the archive in **Source0**. This directive can contain a shell script.
%build	Command or series of commands for building the software into machine code (for compiled languages) or byte code (for some interpreted languages).
%install	Command or series of commands for copying the desired build artifacts from the **%builddir** (where the build happens) to the **%buildroot** directory (which contains the directory structure with the files to be packaged). This usually means copying files from **~/rpmbuild/BUILD** to **~/rpmbuild/BUILDROOT** and creating the necessary directories in **~/rpmbuild/BUILDROOT**. This is only run when creating a package, not when the end-user installs the package. See Working with SPEC filesfor details.
%check	Command or series of commands to test the software. This normally includes things such as unit tests.
%files	The list of files that will be insstalled in the end user's system.
%changelog	A record of changes that have happened to the package between different **Version** or **Release** builds.

3.4.3. Advanced items

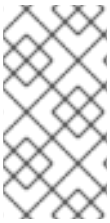
The SPEC file can also contain advanced items, such as Scriptlets or Triggers.

They take effect at different points during the installation process on the end user's system, not the build process.

3.5. BUILDROOTS

In the context of RPM packaging, **buildroot** is a chroot environment. This means that the build artifacts are placed here using the same file system hierarchy as the future hierarchy in the end user's system, with **buildroot** acting as the root directory. The placement of build artifacts should comply with the file system hierarchy standard of the end user's system.

The files in **buildroot** are later put into a **cpio** archive, which becomes the main part of the RPM. When RPM is installed on the end user's system, these files are extracted in the **root** directory, preserving the correct hierarchy.



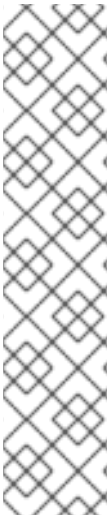
NOTE

Starting from Red Hat Enterprise Linux 6, the **rpmbuild** program has its own defaults. Overriding these defaults leads to several problems; hence, Red Hat does not recommend to define your own value of this macro. You can use the **%{buildroot}** macro with the defaults from the **rpmbuild** directory.

3.6. RPM MACROS

An rpm macro is a straight text substitution that can be conditionally assigned based on the optional evaluation of a statement when certain built-in functionality is used. Hence, RPM can perform text substitutions for you.

An example use is referencing the packaged software *Version* multiple times in a SPEC file. You define *Version* only once in the **%{version}** macro, and use this macro throughout the SPEC file. Every occurrence will be automatically substituted by *Version* that you defined previously.



NOTE

If you see an unfamiliar macro, you can evaluate it with the following command:

```
$ rpm --eval %{_MACRO}
```

Evaluating the %{_bindir} and the %{_libexecdir} macros

```
$ rpm --eval %{_bindir}
/usr/bin

$ rpm --eval %{_libexecdir}
/usr/libexec
```

One of the commonly-used macros is the **%{?dist}** macro, which signals which distribution is used for the build (distribution tag).

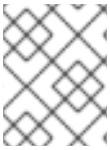
```
# On a RHEL 8.x machine
$ rpm --eval %{?dist}
.el8
```

3.7. WORKING WITH SPEC FILES

To package new software, you need to create a new SPEC file.

There are two ways to achieve this:

- Writing the new SPEC file manually from scratch
- Use the **rpmdev-newspec** utility
  This utility creates an unpopulated SPEC file, and you fill in the necessary directives and fields.



NOTE

Some programmer-focused text editors pre-populate a new **.spec** file with their own SPEC template. The **rpmdev-newspec** utility provides an editor-agnostic method.

The following sections use the three example implementations of the **Hello World!** program that were described in What source code is.

Each of the programs is also fully described in the table below.

Software Name	Explanation of example
bello	A program written in a raw interpreted programming language. It demonstrates when the source code does not need to be built, but only needs to be installed. If a pre-compiled binary needs to be packaged, you can also use this method since the binary would also just be a file.
pello	A program written in a byte-compiled interpreted programming language. It demonstrates byte-compiling the source code and installating the bytecode - the resulting pre-optimized files.
cello	A program written in a natively compiled programming language. It demonstrates a common process of compiling the source code into machine code and installing the resulting executables.

The implementations of **Hello World!** are:

- bello-0.1.tar.gz
- pello-0.1.2.tar.gz
- cello-1.0.tar.gz
  - cello-output-first-patch.patch

As a prerequisite, these implementations need to be placed into the **~/rpmbuild/SOURCES** directory.

3.8. CREATING A NEW SPEC FILE WITH RPMDEV-NEWSPEC

The following procedure shows how to create a SPEC file for each of the three aforementioned **Hello World!** programs using the **rpmdev-newspec** utility.

Procedure

1. Change to the **~/rpmbuild/SPECS** directory and use the **rpmdev-newspec** utility:

```
$ cd ~/rpmbuild/SPECS

$ rpmdev-newspec bello
bello.spec created; type minimal, rpm version >= 4.11.

$ rpmdev-newspec cello
cello.spec created; type minimal, rpm version >= 4.11.

$ rpmdev-newspec pello
pello.spec created; type minimal, rpm version >= 4.11.
```

   The **~/rpmbuild/SPECS/** directory now contains three SPEC files named **bello.spec**, **cello.spec**, and **pello.spec**.

2. Examine the files:

The directives in the files represent those described in What a SPEC file is. In the following sections, you will populate particular section in the output files of **rpmdev-newspec**.

NOTE

The **rpmdev-newspec** utility does not use guidelines or conventions specific to any particular Linux distribution. However, this document targets Red Hat Enterprise Linux, so the **%{buildroot}** notation is preferred over the **$RPM_BUILD_ROOT** notation when referencing RPM's Buildroot for consistency with all other defined or provided macros throughout the SPEC file.

3.9. MODIFYING AN ORIGINAL SPEC FILE FOR CREATING RPMS

The following procedure shows how to modify the output SPEC file provided by **rpmdev-newspec** for creating the RPMs.

Prerequisites

- The source code of the particular program has been placed into the **~/rpmbuild/SOURCES/** directory.
- The unpopulated SPEC file **~/rpmbuild/SPECS/<name>.spec** file has been created by the **rpmdev-newspec** utility.

Procedure

1. Open the output template of the **~/rpmbuild/SPECS/<name>.spec** file provided by the **rpmdev-newspec** utility:
2. Populate the first section of the SPEC file:
   The first section includes these directives that **rpmdev-newspec** grouped together:

   Name

   The **Name** was already specified as an argument to **rpmdev-newspec**.

   Version

   Set the **Version** to match the upstream release version of the source code.

   Release

   The **Release** is automatically set to **1%{?dist}**, which is initially **1**. Increment the initial value whenever updating the package without a change in the upstream release **Version** - such as when including a patch. Reset **Release** to **1** when a new upstream release happens.

   Summary

   The **Summary** is a short, one-line explanation of what this software is.
3. Populate the **License**, **URL**, and **Source0** directives:
   The **License** field is the Software License associated with the source code from the upstream release. The exact format for how to label the **License** in your SPEC file will vary depending on which specific RPM based Linux distribution guidelines you are following.

   For example, you can use GPLv3+.

   The **URL** field provides URL to the upstream software website. For consistency, utilize the RPM macro variable of **%{name}**, and use **https://example.com/%{name}**.

The **Source0** field provides URL to the upstream software source code. It should link directly to the specific version of software that is being packaged. Note that the example URLs given in this documentation include hard-coded values that are possible subject to change in the future. Similarly, the release version can change as well. To simplify these potential future changes, use the **%{name}** and **%{version}** macros. By using these, you need to update only one field in the SPEC file.

4. Populate the **BuildRequires**, **Requires** and **BuildArch** directives:
   BuildRequires specifies build-time dependencies for the package.

   Requires specifies run-time dependencies for the package.

   This is a software written in an interpreted programming language with no natively compiled extensions. Hence, add the **BuildArch** directive with the **noarch** value. This tells RPM that this package does not need to be bound to the processor architecture on which it is built.

5. Populate the **%description**, **%prep**, **%build**, **%install**, **%files**, and **%license** directives:
   These directives can be thought of as section headings, because they are directives that can define multi-line, multi-instruction, or scripted tasks to occur.

   The **%description** is a longer, fuller description of the software than **Summary**, containing one or more paragraphs.

   The **%prep** section specifies how to prepare the build environment. This usually involves expansion of compressed archives of the source code, application of patches, and, potentially, parsing of information provided in the source code for use in a later portion of the SPEC file. In this section you can use the built-in **%setup -q** macro.

   The **%build** section specifies how to build the software.

   The **%install** section contains instructions for **rpmbuild** on how to install the software, once it has been built, into the **BUILDROOT** directory.

   This directory is an empty chroot base directory, which resembles the end user's root directory. Here you can create any directories that will contain the installed files. To create such directories, you can use the RPM macros without having to hardcode the paths.

   The **%files** section specifies the list of files provided by this RPM and their full path location on the end user's system.

   Within this section, you can indicate the role of various files using built-in macros. This is useful for querying the package file manifest metadata using the **rpm** command. For example, to indicate that the LICENSE file is a software license file, use the **%license** macro.

6. The last section, **%changelog**, is a list of datestamped entries for each Version-Release of the package. They log packaging changes, not software changes. Examples of packaging changes: adding a patch, changing the build procedure in the **%build** section.
   Follow this format for the first line:

   Start with an ***** character followed by **Day-of-Week Month Day Year Name Surname <email> - Version-Release**

   Follow this format for the actual change entry:

   - Each change entry can contain multiple items, one for each change.
   - Each item starts on a new line.

- Each item begins with a **-** character.

You have now written an entire SPEC file for the required program.

Additional resources

- An example SPEC file for a program written in bash
- An example SPEC file for a program written in Python
- An example SPEC file for a program written in C
- Building RPMs

3.10. AN EXAMPLE SPEC FILE FOR A PROGRAM WRITTEN IN BASH

This section shows an example SPEC file for the **bello** program that was written in bash.

An example SPEC file for the bello program written in bash

```
Name:       bello
Version:    0.1
Release:    1%{?dist}
Summary:    Hello World example implemented in bash script

License:    GPLv3+
URL:        https://www.example.com/%{name}
Source0:    https://www.example.com/%{name}/releases/%{name}-%{version}.tar.gz

Requires:   bash

BuildArch:  noarch

%description
The long-tail description for our Hello World Example implemented in
bash script.

%prep
%setup -q

%build

%install

mkdir -p %{buildroot}/%{_bindir}

install -m 0755 %{name} %{buildroot}/%{_bindir}/%{name}

%files
%license LICENSE
%{_bindir}/%{name}

%changelog
```

```
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1-1
- First bello package
- Example second item in the changelog for version-release 0.1-1
```

The **BuildRequires** directive, which specifies build-time dependencies for the package, was deleted because there is no building step for **bello**. Bash is a raw interpreted programming language, and the files are just installed to their location on the system.

The **Requires** directive, which specifies run-time dependencies for the package, include only **bash**, because the **bello** script requires only the **bash** shell environment to execute.

The **%build** section, which specifies how to build the software, is blank, because a **bash** does not need to be built.

For installing **bello** you only need to create the destination directory and install the executable **bash** script file there. Hence, you can use the **install** command in the **%install** section. RPM macros allow to do this without hardcoding paths.

Additional resources

- What source code is

3.11. AN EXAMPLE SPEC FILE FOR A PROGRAM WRITTEN IN PYTHON

This section shows an example SPEC file for the **pello** program written in the Python programming language.

An example SPEC file for the pello program written in Python

```
Name:          pello
Version:       0.1.1
Release:       1%{?dist}
Summary:       Hello World example implemented in Python

License:       GPLv3+
URL:           https://www.example.com/%{name}
Source0:       https://www.example.com/%{name}/releases/%{name}-%{version}.tar.gz

BuildRequires:  python
Requires:      python
Requires:      bash

BuildArch:     noarch

%description
The long-tail description for our Hello World Example implemented in Python.

%prep
%setup -q

%build

python -m compileall %{name}.py

%install
```

```
mkdir -p %{buildroot}/%{_bindir}
mkdir -p %{buildroot}/usr/lib/%{name}

cat > %{buildroot}/%{_bindir}/%{name} <<EOF
#!/bin/bash
/usr/bin/python /usr/lib/%{name}/%{name}.pyc
EOF

chmod 0755 %{buildroot}/%{_bindir}/%{name}

install -m 0644 %{name}.py* %{buildroot}/usr/lib/%{name}/

%files
%license LICENSE
%dir /usr/lib/%{name}/
%{_bindir}/%{name}
/usr/lib/%{name}/%{name}.py*

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1.1-1
  - First pello package
```

IMPORTANT

The **pello** program is written in a byte-compiled interpreted language. Hence, the shebang is not applicable because the resulting file does not contain the entry.

Because the shebang is not applicable, you may want to apply one of the following approaches:

- Create a non-byte-compiled shell script that will call the executable.
- Provide a small bit of the Python code that is not byte-compiled as the entry point into the program's execution.

These approaches are useful especially for large software projects with many thousands of lines of code, where the performance increase of pre-byte-compiled code is sizeable.

The **Requires** directive, which specifies run-time dependencies for the package, includes two packages:

- The **python** package is needed to execute the byte-compiled code at runtime
- The **bash** package is needed to execute the small entry-point script

The **BuildRequires** directive, which specifies build-time dependencies for the package, includes only the **python** package. The **pello** program requires the **python** package to perform the byte-compile build process.

The **%build** section, which specifies how to build the software, corresponds to the fact that the software is byte-compiled.

To install **pello**, you need to create a wrapper script because the shebang is not applicable in byte-compiled languages. There are multiple options to accomplish this, such as:

- Making a separate script and using that as a separate **SourceX** directive.

- Creating the file in-line in the SPEC file.

This example shows creating a wrapper script in-line in the SPEC file to demonstrate that the SPEC file itself is scriptable. This wrapper script will execute the Python byte-compiled code by using a **here** document.

The **%install** section in this example also corresponds to the fact that you will need to install the byte-compiled file into a library directory on the system such that it can be accessed.

Additional resources

- What source code is

3.12. AN EXAMPLE SPEC FILE FOR A PROGRAM WRITTEN IN C

This section shows an example SPEC file for the **cello** program that was written in the C programming language.

An example SPEC file for the cello program written in C

```
Name:          cello
Version:       1.0
Release:       1%{?dist}
Summary:       Hello World example implemented in C

License:       GPLv3+
URL:           https://www.example.com/%{name}
Source0:       https://www.example.com/%{name}/releases/%{name}-%{version}.tar.gz

Patch0:        cello-output-first-patch.patch

BuildRequires:  gcc
BuildRequires:  make

%description
The long-tail description for our Hello World Example implemented in
C.

%prep
%setup -q

%patch0

%build
make %{?_smp_mflags}

%install
%make_install

%files
%license LICENSE
%{_bindir}/%{name}
```

```
%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 1.0-1
- First cello package
```

The **BuildRequires** directive, which specifies build-time dependencies for the package, includes two packages that are needed to perform the compilation build process:

- The **gcc** package
- The **make** package

The **Requires** directive, which specifies run-time dependencies for the package, is omitted in this example. All runtime requirements are handled by **rpmbuild**, and the **cello** program does not require anything outside of the core C standard libraries.

The **%build** section reflects the fact that in this example a **Makefile** for the **cello** program was written, hence the GNU make command provided by the **rpmdev-newspec** utility can be used. However, you need to remove the call to **%configure** because you did not provide a configure script.

The installation of the **cello** program can be accomplished by using the **%make_install** macro that was provided by the **rpmdev-newspec** command. This is possible because the **Makefile** for the **cello** program is available.

Additional resources

- What source code is

3.13. BUILDING RPMS

RPMs are built with the **rpmbuild** command. This command expects a certain directory and file structure, which is the same as the structure that was set up by the **rpmdev-setuptree** utility.

Different use cases and desired outcomes require different combinations of arguments to the **rpmbuild** command. The two main use cases are:

- Building source RPMs
- Building binary RPM
  - Rebuilding a binary RPM from a source RPM
  - Building a binary RPM from the SPEC file
  - Building a binary RPM from a source RPM

This following sections describe how to build an RPM after a SPEC file for a program has been created.

3.14. BUILDING SOURCE RPMS

The following procedure describes how to build a source RPM.

Prerequisites

- A SPEC file for the program that we want to package must already exist.

Procedure

Procedure

- Run the **rpmbuild** command with the specified SPEC file:

  ```
  $ rpmbuild -bs SPECFILE
  ```

  Substitute *SPECFILE* with the SPEC file. The **-bs** option stands for the build source.

The following example shows building source RPMs for the **bello**, **pello**, and **cello** projects.

Building source RPMs for bello, pello, and cello.

```
$ cd ~/rpmbuild/SPECS/

8$ rpmbuild -bs bello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm

$ rpmbuild -bs pello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm

$ rpmbuild -bs cello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/cello-1.0-1.el8.src.rpm
```

Verification steps

- Make sure that the **rpmbuild/SRPMS** directory includes the resulting source RPMs. The directory is a part of the structure expected by **rpmbuild**.

Additional resources

- Working with SPEC files.
- Creating a new SPEC file with rpmdev-newspec
- Modifying an original SPEC file for creating RPMs

3.15. REBUILDING A BINARY RPM FROM A SOURCE RPM

The following procedure shows how to rebuild a binary RPM from a source RPM (SRPM).

Procedure

- To rebuild **bello**, **pello**, and **cello** from their SRPMs, run:

  ```
  $ rpmbuild --rebuild ~/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm
  [output truncated]

  $ rpmbuild --rebuild ~/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm
  [output truncated]

  $ rpmbuild --rebuild ~/rpmbuild/SRPMS/cello-1.0-1.el8.src.rpm
  [output truncated]
  ```

NOTE

Invoking **rpmbuild --rebuild** involves:

- Installing the contents of the SRPM - the SPEC file and the source code - into the **~/rpmbuild/** directory.
- Building using the installed contents.
- Removing the SPEC file and the source code.

To retain the SPEC file and the source code after building, you can:

- When building, use the **rpmbuild** command with the **--recompile** option instead of the **--rebuild** option.
- Install the SRPMs using these commands:

```
$ rpm -Uvh ~/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm
Updating / installing…
   1:bello-0.1-1.el8              [100%]

$ rpm -Uvh ~/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm
Updating / installing…
…1:pello-0.1.2-1.el8              [100%]

$ rpm -Uvh ~/rpmbuild/SRPMS/cello-1.0-1.el8.src.rpm
Updating / installing…
…1:cello-1.0-1.el8            [100%]
```

The output generated when creating a binary RPM is verbose, which is helpful for debugging. The output varies for different examples and corresponds to their SPEC files.

The resulting binary RPMs are in the **~/rpmbuild/RPMS/YOURARCH** directory where **YOURARCH** is your architecture or in the **~/rpmbuild/RPMS/noarch/** directory, if the package is not architecture-specific.

3.16. BUILDING A BINARY RPM FROM THE SPEC FILE

The following procedure shows how to build **bello**, **pello**, and **cello** binary RPMs from their SPEC files.

Procedure

- Run the **rpmbuild** command with the **bb** option:

```
$ rpmbuild -bb ~/rpmbuild/SPECS/bello.spec

$ rpmbuild -bb ~/rpmbuild/SPECS/pello.spec

$ rpmbuild -bb ~/rpmbuild/SPECS/cello.spec
```

3.17. BUILDING BINARY RPMS FROM SOURCE RPMS

It is also possible to build any kind of RPM from a source RPM. To do so, use the following procedure.

Procedure

- Run the **rpmbuild** command with one of the below options and with the source package specified:

```
# rpmbuild {-ra|-rb|-rp|-rc|-ri|-rl|-rs} [rpmbuild-options] SOURCEPACKAGE
```

Additional resources

- **rpmbuild(8)** man page

3.18. CHECKING RPMS FOR SANITY

After creating a package, it is necessary to check the quality of the package.

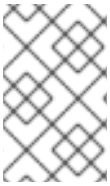
The main tool for checking package quality is rpmlint.

The **rpmlint** tool does the following:

- Improves RPM maintainability.
- Enables sanity checking by performing static analysis of the RPM.
- Enables error checking by performing static analysis of the RPM.

The **rpmlint** tool can check binary RPMs, source RPMs (SRPMs), and SPEC files, so it is useful for all stages of packaging, as shown in the following sections.

Note that **rpmlint** has very strict guidelines; hence it is sometimes acceptable to skip some of its errors and warnings, as shown in the following examples.



NOTE

In the examples described in the following sections, **rpmlint** is run without any options, which produces a non-verbose output. For detailed explanations of each error or warning, you can run **rpmlint -i** instead.

3.19. CHECKING BELLO FOR SANITY

This section shows possible warnings and errors that can occur when checking RPM sanity on the example of the bello SPEC file and bello binary RPM.

3.19.1. Checking the bello SPEC File

Example 3.2. Output of running the rpmlint command on the SPEC file for bello

```
$ rpmlint bello.spec
bello.spec: W: invalid-url Source0: https://www.example.com/bello/releases/bello-0.1.tar.gz HTTP
Error 404: Not Found
0 packages and 1 specfiles checked; 0 errors, 1 warnings.
```

For **bello.spec**, there is only one warning, which says that the URL listed in the **Source0** directive is unreachable. This is expected, because the specified **example.com** URL does not exist. Presuming that we expect this URL to work in the future, we can ignore this warning.

Example 3.3. Output of running the rpmlint command on the SRPM for bello

```
$ rpmlint ~/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm
bello.src: W: invalid-url URL: https://www.example.com/bello HTTP Error 404: Not Found
bello.src: W: invalid-url Source0: https://www.example.com/bello/releases/bello-0.1.tar.gz HTTP
Error 404: Not Found
1 packages and 0 specfiles checked; 0 errors, 2 warnings.
```

For the **bello** SRPM, there is a new warning, which says that the URL specified in the **URL** directive is unreachable. Assuming the link will be working in the future, we can ignore this warning.

3.19.2. Checking the bello binary RPM

When checking binary RPMs, **rpmlint** checks for the following items:

- Documentation
- Manual pages
- Consistent use of the filesystem hierarchy standard

Example 3.4. Output of running the rpmlint command on the binary RPM for bello

```
$ rpmlint ~/rpmbuild/RPMS/noarch/bello-0.1-1.el8.noarch.rpm
bello.noarch: W: invalid-url URL: https://www.example.com/bello HTTP Error 404: Not Found
bello.noarch: W: no-documentation
bello.noarch: W: no-manual-page-for-binary bello
1 packages and 0 specfiles checked; 0 errors, 3 warnings.
```

The **no-documentation** and **no-manual-page-for-binary** warnings say that he RPM has no documentation or manual pages, because we did not provide any. Apart from the above warnings, the RPM passed **rpmlint** checks.

3.20. CHECKING PELLO FOR SANITY

This section shows possible warnings and errors that can occur when checking RPM sanity on the example of the pello SPEC file and pello binary RPM.

3.20.1. Checking the pello SPEC File

Example 3.5. Output of running the rpmlint command on the SPEC file for pello

```
$ rpmlint pello.spec
pello.spec:30: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}
pello.spec:34: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.pyc
pello.spec:39: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}/
pello.spec:43: E: hardcoded-library-path in /usr/lib/%{name}/
```

```
pello.spec:45: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.py*
pello.spec: W: invalid-url Source0: https://www.example.com/pello/releases/pello-0.1.2.tar.gz
HTTP Error 404: Not Found
0 packages and 1 specfiles checked; 5 errors, 1 warnings.
```

The **invalid-url Source0** warning says that the URL listed in the **Source0** directive is unreachable. This is expected, because the specified **example.com** URL does not exist. Presuming that this URL will work in the future, you can ignore this warning.

The **hardcoded-library-path** errors suggest to use the **%{_libdir}** macro instead of hard-coding the library path. For the sake of this example, you can safely ignore these errors. However, for packages going into production make sure to check all errors carefully.

Example 3.6. Output of running the rpmlint command on the SRPM for pello

```
$ rpmlint ~/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm
pello.src: W: invalid-url URL: https://www.example.com/pello HTTP Error 404: Not Found
pello.src:30: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}
pello.src:34: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.pyc
pello.src:39: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}/
pello.src:43: E: hardcoded-library-path in /usr/lib/%{name}/
pello.src:45: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.py*
pello.src: W: invalid-url Source0: https://www.example.com/pello/releases/pello-0.1.2.tar.gz HTTP
Error 404: Not Found
1 packages and 0 specfiles checked; 5 errors, 2 warnings.
```

The new **invalid-url URL** error here is about the **URL** directive, which is unreachable. Assuming that the URL will be valid in the future, you can safely ignore this error.

3.20.2. Checking the pello binary RPM

When checking binary RPMs, **rpmlint** checks for the following items:

- Documentation
- Manual pages
- Consistent use of the Filesystem Hierarchy Standard

Example 3.7. Output of running the rpmlint command on the binary RPM for pello

```
$ rpmlint ~/rpmbuild/RPMS/noarch/pello-0.1.2-1.el8.noarch.rpm
pello.noarch: W: invalid-url URL: https://www.example.com/pello HTTP Error 404: Not Found
pello.noarch: W: only-non-binary-in-usr-lib
pello.noarch: W: no-documentation
pello.noarch: E: non-executable-script /usr/lib/pello/pello.py 0644L /usr/bin/env
pello.noarch: W: no-manual-page-for-binary pello
1 packages and 0 specfiles checked; 1 errors, 4 warnings.
```

The **no-documentation** and **no-manual-page-for-binary** warnings say that the RPM has no documentation or manual pages, because you did not provide any.

The **only-non-binary-in-usr-lib** warning says that you provided only non-binary artifacts in **/usr/lib/**. This directory is normally reserved for shared object files, which are binary files. Therefore, **rpmlint** expects at least one or more files in **/usr/lib/** directory to be binary.

This is an example of an **rpmlint** check for compliance with Filesystem Hierarchy Standard. Normally, use RPM macros to ensure the correct placement of files. For the sake of this example, you can safely ignore this warning.

The **non-executable-script** error warns that the **/usr/lib/pello/pello.py** file has no execute permissions. The **rpmlint** tool expects the file to be executable, because the file contains the shebang. For the purpose of this example, you can leave this file without execute permissions and ignore this error.

Apart from the above warnings and errors, the RPM passed **rpmlint** checks.

3.21. CHECKING CELLO FOR SANITY

This section shows possible warnings and errors that can occur when checking RPM sanity on the example of the cello SPEC file and pello binary RPM.

3.21.1. Checking the cello SPEC File

Example 3.8. Output of running the rpmlint command on the SPEC file for cello

```
$ rpmlint ~/rpmbuild/SPECS/cello.spec
/home/admiller/rpmbuild/SPECS/cello.spec: W: invalid-url Source0:
https://www.example.com/cello/releases/cello-1.0.tar.gz HTTP Error 404: Not Found
0 packages and 1 specfiles checked; 0 errors, 1 warnings.
```

For **cello.spec**, there is only one warning, which says that the URL listed in the **Source0** directive is unreachable. This is expected, because the specified **example.com** URL does not exist. Presuming that this URL will work in the future, you can ignore this warning.

Example 3.9. Output of running the rpmlint command on the SRPM for cello

```
$ rpmlint ~/rpmbuild/SRPMS/cello-1.0-1.el8.src.rpm
cello.src: W: invalid-url URL: https://www.example.com/cello HTTP Error 404: Not Found
cello.src: W: invalid-url Source0: https://www.example.com/cello/releases/cello-1.0.tar.gz HTTP
Error 404: Not Found
1 packages and 0 specfiles checked; 0 errors, 2 warnings.
```

For the **cello** SRPM, there is a new warning, which says that the URL specified in the **URL** directive is unreachable. Assuming the link will be working in the future, you can ignore this warning.

3.21.2. Checking the cello binary RPM

When checking binary RPMs, **rpmlint** checks for the following items:

- Documentation
- Manual pages

- Consistent use of the filesystem hierarchy standard

Example 3.10. Output of running the rpmlint command on the binary RPM for cello

```
$ rpmlint ~/rpmbuild/RPMS/x86_64/cello-1.0-1.el8.x86_64.rpm
cello.x86_64: W: invalid-url URL: https://www.example.com/cello HTTP Error 404: Not Found
cello.x86_64: W: no-documentation
cello.x86_64: W: no-manual-page-for-binary cello
1 packages and 0 specfiles checked; 0 errors, 3 warnings.
```

The **no-documentation** and **no-manual-page-for-binary** warnings say that he RPM has no documentation or manual pages, because you did not provide any. Apart from the above warnings, the RPM passed **rpmlint** checks.

3.22. LOGGING RPM ACTIVITY TO SYSLOG

Any RPM activity or transaction can be logged by the System Logging protocol (syslog).

Prerequisites

- To enable the logging of RPM transactions to syslog, make sure that the **syslog** plug-in is installed on the system:

```
# yum install rpm-plugin-syslog
```

> **NOTE**
>
> The default location for the syslog messages is the **/var/log/messages** file. However, you can configure syslog to use another location to store the messages.

To see the updates on RPM activity, follow the described procedure.

Procedure

1. Open the file that you configured to store the syslog messages, or if you use the default syslog configuration, open the **/var/log/messages** file.
2. Search for new lines including the **[RPM]** string.

3.23. EXTRACTING RPM CONTENT

In particular cases, for example, if a package required by RPM is damaged, it is necessary to extract the content of the package. In such cases, if an RPM installation is still working despite the damage, you can use the **rpm2archive** utility to convert an **.rpm** file to a tar archive to use the content of the package.

> **NOTE**
>
> If the RPM installation is severely damaged, you can use the **rpm2cpio** utility to convert the RPM package file to a cpio archive.

The following procedure describes how to convert an rpm payload to a tar archive using the **rpm2archive** utility.

Procedure

- Run the following command:

```
$ rpm2archive filename.rpm
```

  Replace *filename* with the name of the **.rpm** file.

  The resulting file has the **.tgz** suffix. For example, to archive the **bash** package:

```
$ rpm2archive bash-4.4.19-6.el8.x86_64.rpm
$ bash-4.4.19-6.el8.x86_64.rpm.tgz
bash-4.4.19-6.el8.x86_64.rpm.tgz
```

CHAPTER 4. ADVANCED TOPICS

This section covers topics that are beyond the scope of the introductory tutorial but are useful in real-world RPM packaging.

4.1. SIGNING PACKAGES

Packages are signed to make sure no third party can alter their content. A user can add an additional layer of security by using the HTTPS protocol when downloading the package.

There are three ways to sign a package:

- Section 4.1.2, "Adding a signature to an already existing package" .
- Section 4.1.5, "Replacing the signature on an already existing package" .
- Section 4.1.6, "Signing a package at build-time" .

To be able to sign a package, you need to create a GNU Privacy Guard (GPG) key as described in Section 4.1.1, "Creating a GPG key" .

4.1.1. Creating a GPG key

Procedure

1. Generate a GNU Privacy Guard (GPG) key pair:

```
# gpg --gen-key
```

2. Confirm and see the generated key:

```
# gpg --list-keys
```

3. Export the public key:

```
# gpg --export -a '<Key_name>' > RPM-GPG-KEY-pmanager
```

NOTE

Include the real name that you have selected for the key instead of <Key_name>.

4. Import the exported public key into an RPM database:

```
# rpm --import RPM-GPG-KEY-pmanager
```

4.1.2. Adding a signature to an already existing package

This section describes the most usual case when a package is built without a signature. The signature is added just before the release of the package.

To add a signature to a package, use the **--addsign** option provided by the **rpm-sign** package.

Having more than one signature enables to record the package's path of ownership from the package builder to the end-user.

Procedure

- Add a signature to a package:

```
$ rpm --addsign blather-7.9-1.x86_64.rpm
```

NOTE

You are supposed to enter the password to unlock the secret key for the signature.

4.1.3. Checking the signatures of a package with multiple signatures

Procedure

- To check the signatures of a package with multiple signatures, run the following:

```
$ rpm --checksig blather-7.9-1.x86_64.rpm
blather-7.9-1.x86_64.rpm: size pgp pgp md5 OK
```

The two **pgp** strings in the output of the **rpm --checksig** command show that the package has been signed twice.

4.1.4. A practical example of adding a signature to an already existing package

This section describes an example situation where adding a signature to an already existing package might be useful.

A division of a company creates a package and signs it with the division's key. The company's headquarters then checks the package's signature and adds the corporate signature to the package, stating that the signed package is authentic.

With two signatures, the package makes its way to a retailer. The retailer checks the signatures and, if they match, adds their signature as well.

The package now makes its way to a company that wants to deploy the package. After checking every signature on the package, they know that it is an authentic copy. Depending on the deploying company's internal controls, they may choose to add their own signature, to inform their employees that the package has received their corporate approval

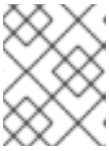
4.1.5. Replacing the signature on an already existing package

This procedure describes how to change the public key without having to rebuild each package.

Procedure

- To change the public key, run the following:

```
$ rpm --resign blather-7.9-1.x86_64.rpm
```



NOTE

You are supposed to enter the password to unlock the secret key for the signature.

The **--resign** option also enables you to change the public key for multiple packages, as shown in the following procedure.

Procedure

- To change the public key for multiple packages, execute:

```
$ rpm --resign b*.rpm
```

NOTE

You are supposed to enter the password to unlock the secret key for the signature.

4.1.6. Signing a package at build-time

Procedure

1. Build the package with the **rpmbuild** command:

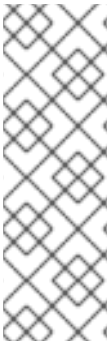
```
$ rpmbuild blather-7.9.spec
```

2. Sign the package with the **rpmsign** command using the **--addsign** option:

```
$ rpmsign --addsign blather-7.9-1.x86_64.rpm
```

3. Optionally, verify the signature of a package:

```
$ rpm --checksig blather-7.9-1.x86_64.rpm
blather-7.9-1.x86_64.rpm: size pgp md5 OK
```



NOTE

When building and signing multiple packages, use the following syntax to avoid entering the Pretty Good Privacy (PGP) passphrase multiple times.

```
$ rpmbuild -ba --sign b*.spec
```

Note that you are supposed to enter the password to unlock the secret key for the signature.

4.2. MORE ON MACROS

This section covers selected built-in RPM Macros. For an exhaustive list of such macros, see RPM Documentation.

4.2.1. Defining your own macros

The following section describes how to create a custom macro.

Procedure

- Include the following line in the RPM SPEC file:

```
%global <name>[(opts)] <body>
```

All whitespace surrounding **<body>** is removed. Name may be composed of alphanumeric characters, and the character _ and must be at least 3 characters in length. Inclusion of the **(opts)** field is optional:

- **Simple** macros do not contain the **(opts)** field. In this case, only recursive macro expansion is performed.
- **Parametrized** macros contain the **(opts)** field. The **opts** string between parentheses is passed to **getopt(3)** for **argc**/**argv** processing at the beginning of a macro invocation.

NOTE

Older RPM SPEC files use the **%define <name> <body>** macro pattern instead. The differences between **%define** and **%global** macros are as follows:

- **%define** has local scope. It applies to a specific part of a SPEC file. The body of a **%define** macro is expanded when used.
- **%global** has global scope. It applies to an entire SPEC file. The body of a **%global** macro is expanded at definition time.

IMPORTANT

Macros are evaluated even if they are commented out or the name of the macro is given into the **%changelog** section of the SPEC file. To comment out a macro, use **%%**. For example: **%%global**.

Additional resources

For comprehensive information on macros capabilities, see RPM Documentation.

4.2.2. Using the %setup macro

This section describes how to build packages with source code tarballs using different variants of the **%setup** macro. Note that the macro variants can be combined The **rpmbuild** output illustrates standard behavior of the **%setup** macro. At the beginning of each phase, the macro outputs **Executing(%...)**, as shown in the below example.

Example 4.1. Example %setup macro output

```
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.DhddsG
```

The shell output is set with **set -x** enabled. To see the content of **/var/tmp/rpm-tmp.DhddsG**, use the **--debug** option because **rpmbuild** deletes temporary files after a successful build. This displays the setup of environment variables followed by for example:

```
cd '/builddir/build/BUILD'
rm -rf 'cello-1.0'
/usr/bin/gzip -dc '/builddir/build/SOURCES/cello-1.0.tar.gz' | /usr/bin/tar -xof -
STATUS=$?
if [ $STATUS -ne 0 ]; then
  exit $STATUS
fi
cd 'cello-1.0'
/usr/bin/chmod -Rf a+rX,u+w,g-w,o-w .
```

The **%setup** macro:

- Ensures that we are working in the correct directory.
- Removes residues of previous builds.
- Unpacks the source tarball.
- Sets up some default privileges.

4.2.2.1. Using the %setup -q macro

The **-q** option limits the verbosity of the **%setup** macro. Only **tar -xof** is executed instead of **tar -xvvof**. Use this option as the first option.

4.2.2.2. Using the %setup -n macro

The **-n** option is used to specify the name of the directory from expanded tarball.

This is used in cases when the directory from expanded tarball has a different name from what is expected (**%{name}-%{version}**), which can lead to an error of the **%setup** macro.

For example, if the package name is **cello**, but the source code is archived in **hello-1.0.tgz** and contains the **hello**/ directory, the SPEC file content needs to be as follows:

```
Name: cello
Source0: https://example.com/%{name}/release/hello-%{version}.tar.gz
…
%prep
%setup -n hello
```

4.2.2.3. Using the %setup -c macro

The **-c** option is used if the source code tarball does not contain any subdirectories and after unpacking, files from an archive fills the current directory.

The **-c** option then creates the directory and steps into the archive expansion as shown below:

```
/usr/bin/mkdir -p cello-1.0
cd 'cello-1.0'
```

The directory is not changed after archive expansion.

4.2.2.4. Using the %setup -D and %setup -T macros

The **-D** option disables deleting of source code directory, and is particularly useful if the **%setup** macro is used several times. With the **-D** option, the following lines are not used:

```
rm -rf 'cello-1.0'
```

The **-T** option disables expansion of the source code tarball by removing the following line from the script:

```
/usr/bin/gzip -dc '/builddir/build/SOURCES/cello-1.0.tar.gz' | /usr/bin/tar -xvvof -
```

4.2.2.5. Using the %setup -a and %setup -b macros

The **-a** and **-b** options expand specific sources:

The **-b** option stands for **before**, and it expands specific sources before entering the working directory. The **-a** option stands for **after**, and it expands those sources after entering. Their arguments are source numbers from the SPEC file preamble.

In the following example, the **cello-1.0.tar.gz** archive contains an empty **examples** directory. The examples are shipped in a separate **examples.tar.gz** tarball and they expand into the directory of the same name. In this case, use **-a 1**, if you want to expand **Source1** after entering the working directory:

```
Source0: https://example.com/%{name}/release/%{name}-%{version}.tar.gz
Source1: examples.tar.gz
...
%prep
%setup -a 1
```

In the following example, examples are provided in a separate **cello-1.0-examples.tar.gz** tarball, which expands into **cello-1.0/examples**. In this case, use **-b 1**, to expand **Source1** before entering the working directory:

```
Source0: https://example.com/%{name}/release/%{name}-%{version}.tar.gz
Source1: %{name}-%{version}-examples.tar.gz
...
%prep
%setup -b 1
```

4.2.3. Common RPM macros in the %files section

This section lists advanced RPM Macros that are needed in the **%files** section of a SPEC file.

Table 4.1. Advanced RPM Macros in the %files section

Macro	Definition
%license	The macro identifies the file listed as a LICENSE file and it will be installed and labeled as such by RPM. Example: **%license LICENSE**

Macro	Definition
%doc	The macro identifies a file listed as documentation and it will be installed and labeled as such by RPM. The macro is used for documentation about the packaged software and also for code examples and various accompanying items. In the event code examples are included, care should be taken to remove executable mode from the file. Example: **%doc README**
%dir	The macro ensures that the path is a directory owned by this RPM. This is important so that the RPM file manifest accurately knows what directories to clean up on uninstall. Example: **%dir %{_libdir}/%{name}**
%config(noreplace)	The macro ensures that the following file is a configuration file and therefore should not be overwritten (or replaced) on a package install or update if the file has been modified from the original installation checksum. If there is a change, the file will be created with **.rpmnew** appended to the end of the filename upon upgrade or install so that the pre-existing or modified file on the target system is not modified. Example: **%config(noreplace) %{_sysconfdir}/%{name}/%{name}.conf**

4.2.4. Displaying the built-in macros

Red Hat Enterprise Linux provides multiple built-in RPM macros.

Procedure

1. To display all built-in RPM macros, run:

```
rpm --showrc
```

> **NOTE**
>
> The output is quite sizeable. To narrow the result, use the command above with the **grep** command.

2. To find information about the RPMs macros for your system's version of RPM, run:

```
rpm -ql rpm
```

> **NOTE**
>
> RPM macros are the files titled **macros** in the output directory structure.

4.2.5. RPM distribution macros

Different distributions provide different sets of recommended RPM macros based on the language implementation of the software being packaged or the specific guidelines of the distribution.

The sets of recommended RPM macros are often provided as RPM packages, ready to be installed with the **yum** package manager.

Once installed, the macro files can be found in the **/usr/lib/rpm/macros.d/** directory.

To display the raw RPM macro definitions, run:

```
rpm --showrc
```

The above output displays the raw RPM macro definitions.

To determine what a macro does and how it can be helpful when packaging RPMs, run the **rpm --eval** command with the name of the macro used as its argument:

```
rpm --eval %{_MACRO}
```

For more information, see the **rpm** man page.

4.2.5.1. Creating custom macros

You can override the distribution macros in the **~/.rpmmacros** file with your custom macros. Any changes that you make affect every build on your machine.

> **WARNING**
>
> Defining any new macros in the **~/.rpmmacros** file is not recommended. Such macros would not be present on other machines, where users may want to try to rebuild your package.

To override a macro, run :

```
%_topdir /opt/some/working/directory/rpmbuild
```

You can create the directory from the example above, including all subdirectories through the **rpmdev-setuptree** utility. The value of this macro is by default **~/rpmbuild**.

```
%_smp_mflags -l3
```

The macro above is often used to pass to Makefile, for example **make %{?_smp_mflags}**, and to set a number of concurrent processes during the build phase. By default, it is set to **-jX**, where **X** is a number of cores. If you alter the number of cores, you can speed up or slow down a build of packages.

4.3. EPOCH, SCRIPTLETS AND TRIGGERS

This section covers **Epoch**, **Scriptlets**, and **Triggers**, which represent advanced directives for RMP SPEC files.

All these directives influence not only the SPEC file, but also the end machine on which the resulting RPM is installed.

4.3.1. The Epoch directive

The **Epoch** directive enables to define weighted dependencies based on version numbers.

If this directive is not listed in the RPM SPEC file, the **Epoch** directive is not set at all. This is contrary to common belief that not setting **Epoch** results in an **Epoch** of 0. However, the **yum** utility treats an unset **Epoch** as the same as an **Epoch** of 0 for the purposes of depsolving.

However, listing **Epoch** in a SPEC file is usually omitted because in majority of cases introducing an **Epoch** value skews the expected RPM behavior when comparing versions of packages.

> **Example 4.2. Using Epoch**
>
> If you install the **foobar** package with **Epoch: 1** and **Version: 1.0**, and someone else packages **foobar** with **Version: 2.0** but without the **Epoch** directive, the new version will never be considered an update. The reason being that the **Epoch** version is preferred over the traditional **Name-Version-Release** marker that signifies versioning for RPM Packages.

Using of **Epoch** is thus quite rare. However, **Epoch** is typically used to resolve an upgrade ordering issue. The issue can appear as a side effect of upstream change in software version number schemes or versions incorporating alphabetical characters that cannot always be compared reliably based on encoding.

4.3.2. Scriptlets

Scriptlets are a series of RPM directives that are executed before or after packages are installed or deleted.

Use **Scriptlets** only for tasks that cannot be done at build time or in an start up script.

4.3.2.1. Scriptlets directives

A set of common **Scriptlet** directives exists. They are similar to the SPEC file section headers, such as **%build** or **%install**. They are defined by multi-line segments of code, which are often written as a standard POSIX shell script. However, they can also be written in other programming languages that RPM for the target machine's distribution accepts. RPM Documentation includes an exhaustive list of available languages.

The following table includes **Scriptlet** directives listed in their execution order. Note that a package containing the scripts is installed between the **%pre** and **%post** directive, and it is uninstalled between the **%preun** and **%postun** directive.

Table 4.2. Scriptlet directives

Directive	Definition
%pretrans	Scriptlet that is executed just before installing or removing any package.
%pre	Scriptlet that is executed just before installing the package on the target system.
%post	Scriptlet that is executed just after the package was installed on the target system.

Directive	Definition
%preun	Scriptlet that is executed just before uninstalling the package from the target system.
%postun	Scriptlet that is executed just after the package was uninstalled from the target system.
%posttrans	Scriptlet that is executed at the end of the transaction.

4.3.2.2. Turning off a scriptlet execution

To turn off the execution of any scriptlet, use the **rpm** command together with the **--no_scriptlet_name_** option.

Procedure

- For example, to turn off the execution of the **%pretrans** scriptlets, run:

```
# rpm --nopretrans
```

  You can also use the **-- noscripts** option, which is equivalent to all of the following:

  - **--nopre**
  - **--nopost**
  - **--nopreun**
  - **--nopostun**
  - **--nopretrans**
  - **--noposttrans**

Additional resources

- For more details, see the **rpm(8)** man page.

4.3.2.3. Scriptlets macros

The **Scriptlets** directives also work with RPM macros.

The following example shows the use of systemd scriptlet macro, which ensures that systemd is notified about a new unit file.

```
$ rpm --showrc | grep systemd
-14: transaction_systemd_inhibit %{plugindir}/systemd_inhibit.so
-14: _journalcatalogdir /usr/lib/systemd/catalog
-14: _presetdir /usr/lib/systemd/system-preset
-14: _unitdir   /usr/lib/systemd/system
-14: _userunitdir       /usr/lib/systemd/user
/usr/lib/systemd/systemd-binfmt %{?} >/dev/null 2>&1 || : /usr/lib/systemd/systemd-sysctl %{?}
>/dev/null 2>&1 || :
```

```
-14: systemd_post
-14: systemd_postun
-14: systemd_postun_with_restart
-14: systemd_preun
-14: systemd_requires
Requires(post): systemd
Requires(preun): systemd
Requires(postun): systemd
-14: systemd_user_post  %systemd_post --user --global %{?} -14: systemd_user_postun %{nil} -
14: systemd_user_postun_with_restart %{nil} -14: systemd_user_preun systemd-sysusers %
{?} >/dev/null 2>&1 || :
echo %{?} | systemd-sysusers - >/dev/null 2>&1 || : systemd-tmpfiles --create %{?} >/dev/null
2>&1 || :

$ rpm --eval %{systemd_post}

if [ $1 -eq 1 ] ; then
        # Initial installation
        systemctl preset  >/dev/null 2>&1 || :
fi

$ rpm --eval %{systemd_postun}

systemctl daemon-reload >/dev/null 2>&1 || :

$ rpm --eval %{systemd_preun}

if [ $1 -eq 0 ] ; then
        # Package removal, not upgrade
        systemctl --no-reload disable  > /dev/null 2>&1 || :
        systemctl stop  > /dev/null 2>&1 || :
fi
```

4.3.3. The Triggers directives

Triggers are RPM directives which provide a method for interaction during package installation and uninstallation.

> **WARNING**
>
> **Triggers** may be executed at an unexpected time, for example on update of the containing package. **Triggers** are difficult to debug, therefore they need to be implemented in a robust way so that they do not break anything when executed unexpectedly. For these reasons, Red Hat recommends to minimize the use of **Triggers**.

The order of execution and the details for each existing **Triggers** are listed below:

```
all-%pretrans
…
```

```
any-%triggerprein (%triggerprein from other packages set off by new install)
new-%triggerprein
new-%pre      for new version of package being installed
…          (all new files are installed)
new-%post     for new version of package being installed

any-%triggerin (%triggerin from other packages set off by new install)
new-%triggerin
old-%triggerun
any-%triggerun (%triggerun from other packages set off by old uninstall)

old-%preun    for old version of package being removed
…          (all old files are removed)
old-%postun   for old version of package being removed

old-%triggerpostun
any-%triggerpostun (%triggerpostun from other packages set off by old un
          install)
…
all-%posttrans
```

The above items are found in the **/usr/share/doc/rpm-4.*/triggers** file.

4.3.4. Using non-shell scripts in a SPEC file

The **-p** scriptlet option in a SPEC file enables the user to invoke a specific interpreter instead of the default shell scripts interpreter (**-p /bin/sh**).

The following procedure describes how to create a script, which prints out a message after installation of the **pello.py** program:

Procedure

1. Open the **pello.spec** file.
2. Find the following line:

```
install -m 0644 %{name}.py* %{buildroot}/usr/lib/%{name}/
```

3. Under the above line, insert:

```
%post -p /usr/bin/python3
print("This is {} code".format("python"))
```

4. Build your package as described in Building RPMs.
5. Install your package:

```
# yum install /home/<username>/rpmbuild/RPMS/noarch/pello-0.1.2-1.el8.noarch.rpm
```

6. Check the output message after the installation:

```
Installing       : pello-0.1.2-1.el8.noarch                    1/1
Running scriptlet: pello-0.1.2-1.el8.noarch                    1/1
This is python code
```

NOTE

To use a Python 3 script, include the following line under **install -m** in a SPEC file:

```
%post -p /usr/bin/python3
```

To use a Lua script, include the following line under **install -m** in a SPEC file:

```
%post -p <lua>
```

This way, you can specify any interpreter in a SPEC file.

4.4. RPM CONDITIONALS

RPM Conditionals enable conditional inclusion of various sections of the SPEC file.

Conditional inclusions usually deal with:

- Architecture-specific sections
- Operating system-specific sections
- Compatibility issues between various versions of operating systems
- Existence and definition of macros

4.4.1. RPM conditionals syntax

RPM conditionals use the following syntax:

If *expression* is true, then do some action:

```
%if expression
…
%endif
```

If *expression* is true, then do some action, in other case, do another action:

```
%if expression
…
%else
…
%endif
```

4.4.2. RPM conditionals examples

This section provides multiple examples of RPM conditionals.

4.4.2.1. The %if conditionals

Example 4.3. Using the %if conditional to handle compatibility between Red Hat Enterprise Linux 8 and other operating systems

```
%if 0%{?rhel} == 8
sed -i '/AS_FUNCTION_DESCRIBE/ s/^/#/' configure.in
sed -i '/AS_FUNCTION_DESCRIBE/ s/^/#/' acinclude.m4
%endif
```

This conditional handles compatibility between RHEL 8 and other operating systems in terms of support of the AS_FUNCTION_DESCRIBE macro. If the package is built for RHEL, the **%rhel** macro is defined, and it is expanded to RHEL version. If its value is 8, meaning the package is build for RHEL 8, then the references to AS_FUNCTION_DESCRIBE, which is not supported by RHEL 8, are deleted from autoconfig scripts.

Example 4.4. Using the %if conditional to handle definition of macros

```
%define ruby_archive %{name}-%{ruby_version}
%if 0%{?milestone:1}%{?revision:1} != 0
%define ruby_archive %{ruby_archive}-%{?milestone}%{?!milestone:%{?revision:r%{revision}}}
%endif
```

This conditional handles definition of macros. If the **%milestone** or the **%revision** macros are set, the **%ruby_archive** macro, which defines the name of the upstream tarball, is redefined.

4.4.2.2. Specialized variants of %if conditionals

The **%ifarch** conditional, **%ifnarch** conditional and **%ifos** conditional are specialized variants of the **%if** conditionals. These variants are commonly used, hence they have their own macros.

4.4.2.2.1. The %ifarch conditional

The **%ifarch** conditional is used to begin a block of the SPEC file that is architecture-specific. It is followed by one or more architecture specifiers, each separated by commas or whitespace.

Example 4.5. An example use of the %ifarch conditional

```
%ifarch i386 sparc
…
%endif
```

All the contents of the SPEC file between **%ifarch** and **%endif** are processed only on the 32-bit AMD and Intel architectures or Sun SPARC-based systems.

4.4.2.2.2. The %ifnarch conditional

The **%ifnarch** conditional has a reverse logic than **%ifarch** conditional.

Example 4.6. An example use of the %ifnarch conditional

```
%ifnarch alpha
…
%endif
```

All the contents of the SPEC file between **%ifnarch** and **%endif** are processed only if not done on a Digital Alpha/AXP-based system.

4.4.2.2.3. The %ifos conditional

The **%ifos** conditional is used to control processing based on the operating system of the build. It can be followed by one or more operating system names.

Example 4.7. An example use of the %ifos conditional

```
%ifos linux
…
%endif
```

All the contents of the SPEC file between **%ifos** and **%endif** are processed only if the build was done on a Linux system.

4.5. PACKAGING PYTHON 3 RPMS

Most Python projects use Setuptools for packaging, and define package information in the **setup.py** file. For more information about Setuptools packaging, see the Setuptools documentation.

You can also package your Python project into an RPM package, which provides the following advantages compared to Setuptools packaging:

- Specification of dependencies of a package on other RPMs (even non-Python)
- Cryptographic signing
  With cryptographic signing, content of RPM packages can be verified, integrated, and tested with the rest of the operating system.

4.5.1. SPEC file description for a Python package

A SPEC file contains instructions that the **rpmbuild** utility uses to build an RPM. The instructions are included in a series of sections. A SPEC file has two main parts in which the sections are defined:

- Preamble (contains a series of metadata items that are used in the Body)
- Body (contains the main part of the instructions)

An RPM SPEC file for Python projects has some specifics compared to non-Python RPM SPEC files. Most notably, a name of any RPM package of a Python library must always include the prefix determining the version, for example, **python3** for Python 3.6, **python38** for Python 3.8, or **python39** for Python 3.9.

Other specifics are shown in the following SPEC file **example for the python3-detox package**. For description of such specifics, see the notes below the example.

```
%global modname detox                                                    1

Name:          python3-detox                                             2
Version:       0.12
Release:       4%{?dist}
Summary:       Distributing activities of the tox tool
License:       MIT
URL:           https://pypi.io/project/detox
Source0:       https://pypi.io/packages/source/d/%{modname}/%{modname}-%{version}.tar.gz

BuildArch:     noarch

BuildRequires:  python36-devel                                           3
BuildRequires:  python3-setuptools
BuildRequires:  python36-rpm-macros
BuildRequires:  python3-six
BuildRequires:  python3-tox
BuildRequires:  python3-py
BuildRequires:  python3-eventlet

%?python_enable_dependency_generator                                     4

%description

Detox is the distributed version of the tox python testing tool. It makes efficient use of multiple CPUs
by running all possible activities in parallel.
Detox has the same options and configuration that tox has, so after installation you can run it in the
same way and with the same options that you use for tox.

   $ detox

%prep
%autosetup -n %{modname}-%{version}

%build
%py3_build                                                               5

%install
%py3_install

%check
%{__python3} setup.py test                                               6

%files -n python3-%{modname}
%doc CHANGELOG
%license LICENSE
%{_bindir}/detox
%{python3_sitelib}/%{modname}/
%{python3_sitelib}/%{modname}-%{version}*

%changelog
...
```

1. The **modname** macro contains the name of the Python project. In this example it is **detox**.
2. When packaging a Python project into RPM, the **python3** prefix always needs to be added to the original name of the project. The original name here is **detox** and the **name of the RPM** is **python3-detox**.
3. **BuildRequires** specifies what packages are required to build and test this package. In BuildRequires, always include items providing tools necessary for building Python packages: **python36-devel** and **python3-setuptools**. The **python36-rpm-macros** package is required so that files with **/usr/bin/python3** interpreter directives are automatically changed to **/usr/bin/python3.6**.
4. Every Python package requires some other packages to work correctly. Such packages need to be specified in the SPEC file as well. To specify the **dependencies**, you can use the **%python_enable_dependency_generator** macro to automatically use dependencies defined in the **setup.py** file. If a package has dependencies that are not specified using Setuptools, specify them within additional **Requires** directives.
5. The **%py3_build** and **%py3_install** macros run the **setup.py build** and **setup.py install** commands, respectively, with additional arguments to specify installation locations, the interpreter to use, and other details.
6. The **check** section provides a macro that runs the correct version of Python. The **%{__python3}** macro contains a path for the Python 3 interpreter, for example **/usr/bin/python3**. We recommend to always use the macro rather than a literal path.

4.5.2. Common macros for Python 3 RPMs

In a SPEC file, always use the macros that are described in the following Macros for Python 3 RPMs table rather than hardcoding their values.

In macro names, always use **python3** or **python2** instead of unversioned **python**. Configure the particular Python 3 version in the **BuildRequires** of the SPEC file to **python36-rpm-macros**, **python38-rpm-macros**, or **python39-rpm-macros**.

Table 4.3. Macros for Python 3 RPMs

Macro	Normal Definition	Description
%{__python3}	/usr/bin/python3	Python 3 interpreter
%{python3_version}	3.6	The full version of the Python 3 interpreter.
%{python3_sitelib}	/usr/lib/python3.6/site-packages	Where pure-Python modules are installed.
%{python3_sitearch}	/usr/lib64/python3.6/site-packages	Where modules containing architecture-specific extensions are installed.

Macro	Normal Definition	Description
%py3_build		Runs the **setup.py build** command with arguments suitable for a system package.
%py3_install		Runs the **setup.py install** command with arguments suitable for a system package.

4.5.3. Automatic provides for Python RPMs

When packaging a Python project, make sure that the following directories are included in the resulting RPM if these directories are present:

- **.dist-info**
- **.egg-info**
- **.egg-link**

From these directories, the RPM build process automatically generates virtual **pythonX.Ydist** provides, for example, **python3.6dist(detox)**. These virtual provides are used by packages that are specified by the **%python_enable_dependency_generator** macro.

4.6. HANDLING INTERPRETER DIRECTIVES IN PYTHON SCRIPTS

In Red Hat Enterprise Linux 8, executable Python scripts are expected to use interpreter directives (also known as hashbangs or shebangs) that explicitly specify at a minimum the major Python version. For example:

```
#!/usr/bin/python3
#!/usr/bin/python3.6
#!/usr/bin/python2
```

The **/usr/lib/rpm/redhat/brp-mangle-shebangs** buildroot policy (BRP) script is run automatically when building any RPM package, and attempts to correct interpreter directives in all executable files.

The BRP script generates errors when encountering a Python script with an ambiguous interpreter directive, such as:

```
#!/usr/bin/python
```

or

```
#!/usr/bin/env python
```

4.6.1. Modifying interpreter directives in Python scripts

Modify interpreter directives in the Python scripts that cause the build errors at RPM build time.

Prerequisites

- Some of the interpretive directives in your Python scripts cause a build error.

Procedure

To modify interpreter directives, complete one of the following tasks:

- Apply the **pathfix.py** script from the **platform-python-devel** package:

  ```
  # pathfix.py -pn -i %{__python3} PATH ...
  ```

  Note that multiple ***PATH*s** can be specified. If a ***PATH*** is a directory, **pathfix.py** recursively scans for any Python scripts matching the pattern **^[a-zA-Z0-9_]+\.py$**, not only those with an ambiguous interpreter directive. Add this command to the **%prep** section or at the end of the **%install** section.

- Modify the packaged Python scripts so that they conform to the expected format. For this purpose, **pathfix.py** can be used outside the RPM build process, too. When running **pathfix.py** outside an RPM build, replace ***%{__python3}*** from the example above with a path for the interpreter directive, such as **/usr/bin/python3**.

If the packaged Python scripts require a version other than Python 3.6, adjust the preceding commands to include the required version.

4.6.2. Changing /usr/bin/python3 interpreter directives in your custom packages

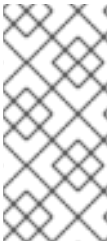
By default, interpreter directives in the form of **/usr/bin/python3** are replaced with interpreter directives pointing to Python from the **platform-python** package, which is used for system tools with Red Hat Enterprise Linux. You can change the **/usr/bin/python3** interpreter directives in your custom packages to point to a specific version of Python that you have installed from the AppStream repository.

Procedure

- To build your package for a specific version of Python, add the ***python*-rpm-macros*** subpackage of the respective **python** package to the **BuildRequires** section of the SPEC file. For example, for Python 3.6, include the following line:

  ```
  BuildRequires: python36-rpm-macros
  ```

  As a result, the **/usr/bin/python3** interpreter directives in your custom package are automatically converted to **/usr/bin/python3.6**.



NOTE

To prevent the BRP script from checking and modifying interpreter directives, use the following RPM directive:

```
%undefine %brp_mangle_shebangs
```

4.7. RUBYGEMS PACKAGES

This section explains what RubyGems packages are, and how to re-package them into RPM.

4.7.1. What RubyGems are

Ruby is a dynamic, interpreted, reflective, object-oriented, general-purpose programming language.

Programs written in Ruby are typically packaged using the RubyGems project, which provides a specific Ruby packaging format.

Packages created by RubyGems are called gems, and they can be re-packaged into RPM as well.

NOTE

This documentation refers to terms related to the RubyGems concept with the **gem** prefix, for example .gemspec is used for the **gem specification**, and terms related to RPM are unqualified.

4.7.2. How RubyGems relate to RPM

RubyGems represent Ruby's own packaging format. However, RubyGems contain metadata similar to those needed by RPM, which enables the conversion from RubyGems to RPM.

According to Ruby Packaging Guidelines, it is possible to re-package RubyGems packages into RPM in this way:

- Such RPMs fit with the rest of the distribution.
- End users are able to satisfy dependencies of a gem by installing the appropriate RPM-packaged gem.

RubyGems use similar terminology as RPM, such as SPEC files, package names, dependencies and other items.

To fit into the rest of RHEL RPM distribution, packages created by RubyGems must follow the conventions listed below:

- Names of gems must follow this pattern:

```
rubygem-%{gem_name}
```

- To implement a shebang line, the following string must be used:

```
#!/usr/bin/ruby
```

4.7.3. Creating RPM packages from RubyGems packages

This section describes how to create RPM packages from packages created by RubyGems.

To create a source RPM for a RubyGems package, two files are needed:

- A gem file
- An RPM SPEC file

4.7.3.1. RubyGems SPEC file conventions

A RubyGems SPEC file must meet the following conventions:

- Contain a definition of **%{gem_name}**, which is the name from the gem's specification.
- The source of the package must be the full URL to the released gem archive; the version of the package must be the gem's version.
- Contain the **BuildRequires:** a directive defined as follows to be able to pull in the macros needed to build.

  ```
  BuildRequires:rubygems-devel
  ```

- Not contain any RubyGems **Requires** or **Provides**, because those are autogenerated.
- Not contain the **BuildRequires:** directive defined as follows, unless you want to explicitly specify Ruby version compatibility:

  ```
  Requires: ruby(release)
  ```

  The automatically generated dependency on RubyGems (**Requires: ruby(rubygems)**) is sufficient.

Macros
Macros useful for packages created by RubyGems are provided by the **rubygems-devel** packages.

Table 4.4. RubyGems' macros

Macro name	Extended path	Usage
%{gem_dir}	/usr/share/gems	Top directory for the gem structure.
%{gem_instdir}	%{gem_dir}/gems/%{gem_name}-%{version}	Directory with the actual content of the gem.
%{gem_libdir}	%{gem_instdir}/lib	The library directory of the gem.
%{gem_cache}	%{gem_dir}/cache/%{gem_name}-%{version}.gem	The cached gem.
%{gem_spec}	%{gem_dir}/specifications/%{gem_name}-%{version}.gemspec	The gem specification file.
%{gem_docdir}	%{gem_dir}/doc/%{gem_name}-%{version}	The RDoc documentation of the gem.

Macro name	Extended path	Usage
%{gem_extdir_mri}	%{_libdir}/gems/ruby/%{gem_name}-%{version}	The directory for gem extension.

4.7.3.2. RubyGems SPEC file example

This section provides an example SPEC file for building gems together with an explanation of its particular sections.

An example RubyGems SPEC file

```
%prep
%setup -q -n  %{gem_name}-%{version}

# Modify the gemspec if necessary
# Also apply patches to code if necessary
%patch0 -p1

%build
# Create the gem as gem install only works on a gem file
gem build ../%{gem_name}-%{version}.gemspec

# %%gem_install compiles any C extensions and installs the gem into ./%%gem_dir
# by default, so that we can move it into the buildroot in %%install
%gem_install

%install
mkdir -p %{buildroot}%{gem_dir}
cp -a ./%{gem_dir}/* %{buildroot}%{gem_dir}/

# If there were programs installed:
mkdir -p %{buildroot}%{_bindir}
cp -a ./%{_bindir}/* %{buildroot}%{_bindir}

# If there are C extensions, copy them to the extdir.
mkdir -p %{buildroot}%{gem_extdir_mri}
cp -a .%{gem_extdir_mri}/{gem.build_complete,*.so} %{buildroot}%{gem_extdir_mri}/
```

The following table explains the specifics of particular items in a RubyGems SPEC file:

Table 4.5. RubyGems' SPEC directives specifics

SPEC directive	RubyGems specifics

SPEC directive	RubyGems specifics
%prep	RPM can directly unpack gem archives, so you can run the **gem unpack** comamnd to extract the source from the gem. The **%setup -n %{gem_name}-%{version}** macro provides the directory into which the gem has been unpacked. At the same directory level, the **%{gem_name}-%{version}.gemspec** file is automatically created, which can be used to rebuild the gem later, to modify the **.gemspec**, or to apply patches to the code.
%build	This directive includes commands or series of commands for building the software into machine code. The **%gem_install** macro operates only on gem archives, and the gem is recreated with the next gem build. The gem file that is created is then used by **%gem_install** to build and install the code into the temporary directory, which is **./%{gem_dir}** by default. The **%gem_install** macro both builds and installs the code in one step. Before being installed, the built sources are placed into a temporary directory that is created automatically. The **%gem_install** macro accepts two additional options: **-n <gem_file>**, which allows to override gem used for installation, and **-d <install_dir>**, which might override the gem installation destination; using this option is not recommended. The **%gem_install** macro must not be used to install into the **%{buildroot}**.
%install	The installation is performed into the **%{buildroot}** hierarchy. You can create the directories that you need and then copy what was installed in the temporary directories into the **%{buildroot}** hierarchy. If this gem creates shared objects, they are moved into the architecture-specific **%{gem_extdir_mri}** path.

For more information on RubyGems SPEC files, see Ruby Packaging Guidelines.

4.7.3.3. Converting RubyGems packages to RPM SPEC files with gem2rpm

The **gem2rpm** utility converts RubyGems packages to RPM SPEC files.

4.7.3.3.1. Installing gem2rpm

Procedure

- To install **gem2rpm** from RubyGems.org, run:

```
$ gem install gem2rpm
```

4.7.3.3.2. Displaying all options of gem2rpm

Procedure

- To see all options of **gem2rpm**, run:

```
gem2rpm --help
```

4.7.3.3.3. Using gem2rpm to covert RubyGems packages to RPM SPEC files

Procedure

- Download a gem in its latest version, and generate the RPM SPEC file for this gem:

```
$ gem2rpm --fetch <gem_name> > <gem_name>.spec
```

The described procedure creates an RPM SPEC file based on the information provided in the gem's metadata. However, the gem misses some important information that is usually provided in RPMs, such as the license and the changelog. The generated SPEC file thus needs to be edited.

4.7.3.3.4. Editing gem2rpm templates

It is recommended to edit the template from which the RPM SPEC file is generated rather than the generated SPEC file itself.

The template is a standard Embedded Ruby (ERB) file, which includes variables listed in the following table.

Table 4.6. Variables in the gem2rpm template

Variable	Explanation
package	The **Gem::Package** variable for the gem.
spec	The **Gem::Specification** variable for the gem (the same as format.spec).
config	The **Gem2Rpm::Configuration** variable that can redefine default macros or rules used in spec template helpers.
runtime_dependencies	The **Gem2Rpm::RpmDependencyList** variable providing a list of package runtime dependencies.
development_dependencies	The **Gem2Rpm::RpmDependencyList** variable providing a list of package development dependencies.
tests	The **Gem2Rpm::TestSuite** variable providing a list of test frameworks allowing their execution.
files	The **Gem2Rpm::RpmFileList** variable providing an unfiltered list of files in a package.
main_files	The **Gem2Rpm::RpmFileList** variable providing a list of files suitable for the main package.
doc_files	The **Gem2Rpm::RpmFileList** variable providing a list of files suitable for the **-doc** subpackage.
format	The **Gem::Format** variable for the gem. Note that this variable is now deprecated.

Procedure

- To see all available templates, run:

```
$ gem2rpm --templates
```

To edit the **gem2rpm** templates, follow this procedure:

Procedure

1. Save the default template:

```
$ gem2rpm -T > rubygem-<gem_name>.spec.template
```

2. Edit the template as needed.

3. Generate the SPEC file using the edited template:

```
$ gem2rpm -t rubygem-<gem_name>.spec.template <gem_name>-<latest_version.gem >
<gem_name>-GEM.spec
```

You can now build an RPM package using the edited template as described in Building RPMs.

4.8. HOW TO HANDLE RPM PACKAGES WITH PERLS SCRIPTS

Since RHEL 8, the Perl programming language is not included in the default buildroot. Therefore, the RPM packages that include Perl scripts must explicitly indicate the dependency on Perl using the **BuildRequires:** directive in RPM SPEC file.

4.8.1. Common Perl-related dependencies

The most frequently occurring Perl-related build dependencies used in **BuildRequires:** are :

- **perl-generators**
  Automatically generates run-time **Requires** and **Provides** for installed Perl files. When you install a Perl script or a Perl module, you must include a build dependency on this package.

- **perl-interpreter**
  The Perl interpreter must be listed as a build dependency if it is called in any way, either explicitly via the **perl** package or the **%__perl** macro, or as a part of your package's build system.

- **perl-devel**
  Provides Perl header files. If building architecture-specific code which links to the **libperl.so** library, such as an XS Perl module, you must include **BuildRequires: perl-devel**.

4.8.2. Using a specific Perl module

If a specific Perl module is required at build time, use the following procedure:

Procedure

- Apply the following syntax in your RPM SPEC file:

```
BuildRequires: perl(MODULE)
```

NOTE

Apply this syntax to Perl core modules as well, because they can move in and out of the **perl** package over time.

4.8.3. Limiting a package to a specific Perl version

To limit your package to a specific Perl version, follow this procedure:

Procedure

- Use the **perl(:VERSION)** dependency with the desired version constraint in your RPM SPEC file: For example, to limit a package to Perl version 5.22 and higher, use:

```
BuildRequires: perl(:VERSION) >= 5.22
```

WARNING

Do not use a comparison against the version of the **perl** package because it includes an epoch number.

4.8.4. Ensuring that a package uses the correct Perl interpreter

Red Hat provides multiple Perl interpreters, which are not fully compatible. Therefore, any package that delivers a Perl module must use at run time the same Perl interpreter that was used at build time.

To ensure this, follow the procedure below:

Procedure

- Include versioned **MODULE_COMPAT Requires** in RPM SPEC file for any package that delivers a Perl module:

```
Requires:  perl(:MODULE_COMPAT_%(eval `perl -V:version`; echo $version))
```

CHAPTER 5. NEW FEATURES IN RHEL 8

This section documents the most notable changes in RPM packaging between Red Hat Enterprise Linux 7 and 8.

5.1. SUPPORT FOR WEAK DEPENDENCIES

5.1.1. Introduction to Weak dependencies policy

Weak dependencies are variants of the **Requires** directive. These variants are matched against virtual **Provides:** and package names using **Epoch-Version-Release** range comparisons.

Weak dependencies have two strengths (**weak** and **hint**) and two directions (**forward** and **backward**), as summarized in the following table.

NOTE

The **forward** direction is analogous to **Requires:**. The **backward** has no analog in the previous dependency system.

Table 5.1. Possible combinations of Weak dependencies' strengths and directions

Strength/Direction	Forward	Backward
Weak	Recommends:	Supplements:
Hint	Suggests:	Enhances:

The main advantages of the **Weak dependencies** policy are:

- It allows smaller minimal installations while keeping the default installation feature rich.
- Packages can specify preferences for specific providers while maintaining the flexibility of virtual provides.

5.1.1.1. Weak dependencies

By default, **Weak dependencies** are treated similarly to regular **Requires:**. Matching packages are included in the **YUM** transaction. If adding the package leads to an error, **YUM** by default ignores the dependency. Hence, users can exclude packages that would be added by **Weak dependencies** or remove them later.

Conditions of use
You can use **Weak dependencies** only if the package still functions without the dependency.

NOTE

It is acceptable to create packages with very limited functionality without adding any of its weak requirements.

Use cases

Use **Weak dependencies** especially where it is possible to minimize the installation for reasonable use cases, such as building virtual machines or containers that have a single purpose and do not require the full feature set of the package.

Typical use cases for **Weak dependencies** are:

- Documentation
  - Documentation viewers if missing them is handled gracefully
- Examples
- Plug-ins or add-ons
  - Support for file formats
  - Support for protocols

5.1.1.2. Hints

Hints are by default ignored by **YUM**. They can be used by GUI tools to offer add-on packages that are not installed by default but can be useful in combination with the installed packages.

Do not use **Hints** for the requirements of the main use cases of a package. Include such requirements in the strong or **Weak dependencies** instead.

Package Preference
YUM uses **Weak dependencies** and **Hints** to decide which package to use if there is a choice between multiple equally valid packages. Packages that are pointed at by dependencies from installed or to be installed packages are preferred.

Note, the normal rules of dependency resolution are not influenced by this feature. For example, **Weak dependencies** cannot enforce an older version of a package to be chosen.

If there are multiple providers for a dependency, the requiring package can add a **Suggests:** to provide a hint to the dependency resolver about which option is preferred.

Enhances: is only used when the main package and other providers agree that adding the hint to the required package is for some reason the cleaner solution.

Example 5.1. Using Hints to prefer one package over another

```
Package A: Requires: mysql

Package mariadb: Provides: mysql

Package community-mysql: Provides: mysql
```

If you want to prefer the **mariadb** package over the **community-mysql** package → use:

```
Suggests: mariadb to Package A.
```

5.1.1.3. Forward and Backward dependencies

Forward dependencies are, similarly to **Requires**, evaluated for packages that are being installed. The best of the matching packages are also installed.

In general, prefer **Forward dependencies**. Add the dependency to the package when getting the other package added to the system.

For **Backward dependencies**, the packages containing the dependency are installed if a matching package is installed as well.

Backward dependencies are mainly designed for third party vendors who can attach their plug-ins, add-ons, or extensions to distribution or other third party packages.

5.2. SUPPORT FOR BOOLEAN DEPENDENCIES

Starting with version 4.13, RPM is able to process boolean expressions in the following dependencies:

- **Requires**
- **Recommends**
- **Suggests**
- **Supplements**
- **Enhances**
- **Conflicts**

This section describes boolean dependencies syntax, provides a list of boolean operators, and explains boolean dependencies nesting as well as boolean dependencies semantics.

5.2.1. Boolean dependencies syntax

Boolean expressions are always enclosed with parenthesis.

They are build out of normal dependencies:

- Name only or name
- Comparison
- Version description

5.2.2. Boolean operators

RPM 4.13 introduced the following boolean operators:

Table 5.2. Boolean operators introduced with RPM 4.13

Boolean operator	Description	Example use
and	Requires all operands to be fulfilled for the term to be true.	Conflicts: (pkgA and pkgB)

Boolean operator	Description	Example use
or	Requires one of the operands to be fulfilled for the term to be true.	Requires: (pkgA >= 3.2 or pkgB)
if	Requires the first operand to be fulfilled if the second is. (reverse implication)	Recommends: (myPkg-langCZ if langsupportCZ)
if else	Same as the **if** operator, plus requires the third operand to be fulfilled if the second is not.	Requires: myPkg-backend-mariaDB if mariaDB else sqlite

RPM 4.14 introduced the following additional boolean operators:

Table 5.3. Boolean operators introduced with RPM 4.14

Boolean operator	Description	Example use
with	Requires all operands to be fulfilled by the same package for the term to be true.	Requires: (pkgA-foo with pkgA-bar)
without	Requires a single package that satisfies the first operand but not the second. (set subtraction)	Requires: (pkgA-foo without pkgA-bar)
unless	Requires the first operand to be fulfilled if the second is not. (reverse negative implication)	Conflicts: (myPkg-driverA unless driverB)
unless else	Same as the **unless** operator, plus requires the third operand to be fulfilled if the second is.	Conflicts: (myPkg-backend-SDL1 unless myPkg-backend-SDL2 else SDL2)

IMPORTANT

The **if** operator cannot be used in the same context with the **or** operator, and the **unless** operator cannot be used in the same context with **and**.

5.2.3. Nesting

Operands themselves can be used as boolean expressions, as shown in the below examples.

Note that in such case, operands also need to be surrounded by parenthesis. You can chain the **and** and **or** operators together repeating the same operator with only one set of surrounding parenthesis.

Example 5.2. Example use of operands applied as boolean expressions

```
Requires: (pkgA or pkgB or pkgC)
```

```
Requires: (pkgA or (pkgB and pkgC))

Supplements: (foo and (lang-support-cz or lang-support-all))

Requires: (pkgA with capB) or (pkgB without capA)

Supplements: ((driverA and driverA-tools) unless driverB)

Recommends: myPkg-langCZ and (font1-langCZ or font2-langCZ) if langsupportCZ
```

5.2.4. Semantics

Using **Boolean dependencies** does not change the semantic of regular dependencies.

If **Boolean dependencies** are used, checking for one match all names are checked and the boolean value of there being a match is then aggregated over the Boolean operators.

IMPORTANT

For all dependencies with the exception of **Conflicts:**, the result has to be **True** to not prevent an install. For **Conflicts:**, the result has to be **False** to not prevent an install.

WARNING

Provides are not dependencies and cannot contain boolean expressions.

5.2.4.1. Understanding the output of the if operator

The **if** operator is also returning a boolean value, which is usually close to what the intuitive understanding is. However, the below examples show that in some cases intuitive understanding of **if** can be misleading.

Example 5.3. Misleading outputs of the if operator

This statement is true if pkgB is not installed. However, if this statement is used where the default result is false, things become complicated:

```
Requires: (pkgA if pkgB)
```

This statement is a conflict unless pkgB is installed and pkgA is not:

```
Conflicts: (pkgA if pkgB)
```

So you might rather want to use:

```
Conflicts: (pkgA and pkgB)
```

The same is true if the **if** operator is nested in **or** terms:

```
Requires: ((pkgA if pkgB) or pkgC or pkg)
```

This also makes the whole term true, because the **if** term is true if pkgB is not installed. If pkgA only helps if pkgB is installed, use **and** instead:

```
Requires: ((pkgA and pkgB) or pkgC or pkg)
```

5.3. SUPPORT FOR FILE TRIGGERS

File triggers are a kind of RPM scriptlets, which are defined in a SPEC file of a package.

Similar to **Triggers**, they are declared in one package but executed when another package that contains the matching files is installed or removed.

A common use of **File triggers** is to update registries or caches. In such use case, the package containing or managing the registry or cache should contain also one or more **File triggers**. Including **File triggers** saves time compared to the situation when the package controls updating itself.

5.3.1. File triggers syntax

File triggers have the following syntax:

```
%file_trigger_tag [FILE_TRIGGER_OPTIONS] — PATHPREFIX…
body_of_script
```

Where:

file_trigger_tag defines a type of file trigger. Allowed types are:

- **filetriggerin**
- **filetriggerun**
- **filetriggerpostun**
- **transfiletriggerin**
- **transfiletriggerun**
- **transfiletriggerpostun**

FILE_TRIGGER_OPTIONS have the same purpose as RPM scriptlets options, except for the **-P** option.

The priority of a trigger is defined by a number. The bigger number, the sooner the file trigger script is executed. Triggers with priority greater than 100000 are executed before standard scriptlets, and the other triggers are executed after standard scriptlets. The default priority is set to 1000000.

Every file trigger of each type must contain one or more path prefixes and scripts.

5.3.2. Examples of File triggers syntax

This section shows concrete examples of **File triggers** syntax:

```
%filetriggerin — /lib, /lib64, /usr/lib, /usr/lib64
/usr/sbin/ldconfig
```

This file trigger executes **/usr/bin/ldconfig** directly after the installation of a package that contains a file having a path starting with **/usr/lib** or **/lib**. The file trigger is executed just once even if the package includes multiple files with the path starting with **/usr/lib** or **/lib**. However, all file names starting with **/usr/lib** or **/lib** are passed to standard input of trigger script so that you can filter inside of your script as shown below:

```
%filetriggerin — /lib, /lib64, /usr/lib, /usr/lib64
grep "foo" && /usr/sbin/ldconfig
```

This file trigger executes **/usr/bin/ldconfig** for each package containing files starting with **/usr/lib** and containing **foo** at the same time. Note that the prefix-matched files include all types of files including regular files, directories, symlinks and others.

5.3.3. File triggers types

File triggers have two main types:

- File triggers executed once per package
- File triggers executed once per transaction

File triggers are further divided based on the time of execution as follows:

- Before or after installation or erasure of a package
- Before or after a transaction

5.3.3.1. Executed once per package File triggers

File triggers executed once per package are:

- %filetriggerin
- %filetriggerun
- %filetriggerpostun

%filetriggerin
This file trigger is executed after installation of a package if this package contains one or more files that match the prefix of this trigger. It is also executed after installation of a package that contains this file trigger and there is one or more files matching the prefix of this file trigger in the **rpmdb** database.

%filetriggerun
This file trigger is executed before uninstallation of a package if this package contains one or more files that match the prefix of this trigger. It is also executed before uninstallation of a package that contains this file trigger and there is one or more files matching the prefix of this file trigger in **rpmdb**.

%filetriggerpostun

This file trigger is executed after uninstallation of a package if this package contains one or more files that match the prefix of this trigger.

5.3.3.2. Executed once per transaction File triggers

File triggers executed once per transaction are:

- %transfiletriggerin
- %transfiletriggerun
- %transfiletriggerpostun

%transfiletriggerin
This file trigger is executed once after a transaction for all installed packages that contain one or more files that match the prefix of this trigger. It is also executed after a transaction if there was a package containing this file trigger in that transaction and there is one or more files matching the prefix of this trigger in **rpmdb**.

%transfiletriggerun
This file trigger is executed once before a transaction for all packages that meet the following conditions:

- The package will be uninstalled in this transaction
- The package contains one or more files that match the prefix of this trigger

It is also executed before a transaction if there is a package containing this file trigger in that transaction and there is one or more files matching the prefix of this trigger in **rpmdb**.

%transfiletriggerpostun
This file trigger is executed once after a transaction for all uninstalled packages that contain one or more file that matches the prefix of this trigger.

NOTE

The list of triggering files is not available in this trigger type.

Therefore, if you install or uninstall multiple packages that contain libraries, the ldconfig cache is updated at the end of the whole transaction. This significantly improves the performance compared to RHEL 7 where the cache was updated for each package separately. Also the scriptlets which called ldconfig in %post and %postun in SPEC file of every package are no longer needed.

5.3.4. Example use of File triggers in glibc

This section shows a real-world example of use of **File triggers** within the **glibc** package.

In RHEL 8, **File triggers** are implemented in **glibc** to call the **ldconfig** command at the end of an installation or uninstallation transaction.

This is ensured by including the following scriptlets in the **glibc's** SPEC file:

```
%transfiletriggerin common -P 2000000 – /lib /usr/lib /lib64 /usr/lib64
/sbin/ldconfig
%end
```

```
%transfiletriggerpostun common -P 2000000 – /lib /usr/lib /lib64 /usr/lib64
/sbin/ldconfig
%end
```

Therefore, if you install or uninstall multiple packages, the **ldconfig** cache is updated for all installed libraries after the whole transaction is finished. Consequently, it is no longer necessary to include the scriptlets calling **ldconfig** in RPM SPEC files of individual packages. This improves the performance compared to RHEL 7, where the cache was updated for each package separately.

5.4. STRICTER SPEC PARSER

The SPEC parser has now some changes incorporated. Hence, it can identify new issues that were previously ignored.

5.5. SUPPORT FOR FILES ABOVE 4 GB

On Red Hat Enterprise Linux 8, **RPM** can use 64-bit variables and tags, which enables operating on files and packages bigger than 4 GB.

5.5.1. 64-bit RPM tags

Several RPM tags exist in both 64-bit versions and previous 32-bit versions. Note that the 64-bit versions have the **LONG** string in front of their name.

Table 5.4. RPM tags available in both 32-bit and 64-bit versions

32-bit variant tag name	62-bit variant tag name	Tag description
RPMTAG_SIGSIZE	RPMTAG_LONGSIGSIZE	Header and compressed payload size.
RPMTAG_ARCHIVESIZE	RPMTAG_LONGARCHIVESIZE	Uncompressed payload size.
RPMTAG_FILESIZES	RPMTAG_LONGFILESIZES	Array of file sizes.
RPMTAG_SIZE	RPMTAG_LONGSIZE	Sum of all file sizes.

5.5.1.1. Using 64-bit tags on command line

The **LONG** extensions are always enabled on the command line. If you previously used scripts containing the **rpm -q --qf** command, you can add **long** to the name of such tags:

```
rpm -qp --qf="[%{filenames} %{longfilesizes}\n]"
```

5.6. OTHER FEATURES

Other new features related to RPM packaging in Red Hat Enterprise Linux 8 are:

- Simplified signature checking output in non-verbose mode
- Support for the enforced payload verification

- Support for the enforcing signature checking mode
- Additions and deprecations in macros

CHAPTER 6. ADDITIONAL RESOURCES ABOUT RPM PACKAGING

This section provides references to various topics related to RPMs, RPM packaging, and RPM building. Some of these are advanced and extend the introductory material included in this documentation.

Red Hat Software Collections Overview - The Red Hat Software Collections offering provides continuously updated development tools in latest stable versions.

Red Hat Software Collections - The Packaging Guide provides an explanation of Software Collections and details how to build and package them. Developers and system administrators with basic understanding of software packaging with RPM can use this Guide to get started with Software Collections.

Mock - Mock provides a community-supported package building solution for various architectures and different Fedora or RHEL versions than has the build host.

RPM Documentation - The official RPM documentation.

Fedora Packaging Guidelines - The official packaging guidelines for Fedora, useful for all RPM-based distributions.