RPM Spec file 🚧

RPM Spec File Format See also: **RPM Files**	This is a very early version of information about the RPM spec file format. 🚧 For now it holds some information, mostly copied from the identified references. It's not complete and not properly organized yet. The gaol is to integrate information from various sites, organize it in an easy-to-find way, with links to web pages where more information is available.		
Information about RPM	Maximum RPM book (RedHat © 2000)		
RPM Spec file format References	RPM SPEC file directives @ Red Hat RPM Spec File format @ rpm.org and it's markdown source file @ GitHub Macro syntax @ rpm.org		
Comments	Comments in spec file have a # at the **beginning** of a line.	`# this is a comment`	
Directives (preamble tags)	Since RPM 4.20 (released October 2024), preamble tags can be indented with white space. Older versions require the tag at the beginning of the line. • In Emacs rpm-spec-mode, the **rpm-spec-tag-face** controls the face used to show directives/tags.		
Name (must match SPEC file name)	The **Name** tag contains the proper name of the package. • Names must not include whitespace and may include a hyphen '-' (unlike version and release tags). • Names should not include any numeric operators ('<', '>','=') as future versions of rpm may need to reserve characters other than '-'.		
Summary	A brief (< 70 characters), 1-line summary of the package.	**DocDir**	Declare a non-default documentation directory for the package. Usually not needed.
Version	Version of the packaged content, typically software. • The version string consists of alphanumeric characters, which can optionally be segmented with the separators **.**, **_** and **+**, plus **~** and **^** . • Tilde (**~**) can be used to force sorting lower than base (**1.1~201601** < **1.1**). • Caret (**^**) can be used to force sorting higher than base (**1.1^201601** > **1.1**). • These are useful for handling pre- and post-release versions, such as **1.0~rc1** and **2.0^a**.		
Release	Package release, used for distinguishing between different builds of the same software version. 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.		
Licence	Short (< 70 characters) summary of the package license. Example: `License: GPLv3`	**SourceLicense**	If license of the sources differ from the main package the license tag of the source package can be set with this. • If not given the license tag of the source and the main package are the same.
Group	Optional, short (< 70 characters) group of the package. Example: **Group**: Development/Libraries	**Vendor**	Optional. Name of the vendor organization.
URL	The full URL for more information about the program. • Most often this is the upstream project website for the software being packaged.	**BugURL**	Bug reporting URL for the package.

Source. (Source0, Source1, …)	Used to declare source(s) used to build the package. All sources will will be packaged into **source RPMs**.
	• If needed, more **Source**X directives can be added, incrementing the number each time, for example: **Source1**, **Source2**, **Source3**, and so on.
	• **Source** numbers do not need to be consecutive and may include leading zeroes.
	• Unnumbered source tag Source: is also supported and is automatically assigned the next available integer.
	
	All sources will be packaged into source rpms. Arbitrary number of sources may be declared, for example:
	`Source0: thesoft-1.0.tar.gz`
	`Source1: thesoft-data-1.0.zip`
	
	It normally identifies the 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.		

Patch	Used to declare patches applied on top of **Source**s. • All patches declared will be packaged into source rpms. • The directive can be applied in two ways: with or without numbers at the end of Patch. Just like sources, patches can be numbered or unnumbered and are indexed the same way. • If no number is given, one is assigned to the entry internally. It's also possible to give the numbers explicitly using **Patch0**, **Patch1**, **Patch2**,… • 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. • Unless there is a specific reason to use numbered patches, the recommended approach is to use unnumbered patches and apply them using `%autosetup` or `%autopatch` macro which automatically applies all patches in the order they are given in the SPEC file.		
NoSource **NoPatch**	Files ending in .nosrc.rpm are generally source RPM packages whose spec files have one or more **NoSource**: or **NoPatch**: directives in them. • Both directives use the named source or patch file to build the resulting binary RPM package as usual, but they are not included in the source RPM package. • The original intent of this ability of RPM was to allow proprietary or non-distributable software to be built using RPM, but to keep the proprietary or non-distributable parts out of the resulting source RPM package, so that they would not get distributed. • They also have utility if you are building RPM packages for software which is archived at a well-known location and does not require that you distribute the source with the binary, for example, for an organization's internal use, where storing large quantities of source is not as meaningful. • The end result of all this, though, is that you can't rebuild ``no-source'' RPM packages using `rpm --rebuild' unless you also have the sources or patches which are not included in the .nosrc.rpm.		
Prefixes (or **Prefix**)	Specify prefixes this package may be installed into, used to make packages relocatable. Very few packages are. • See Relocatable Packages for details.	**RemovePathPostfixes**	Colon separated lists of path postfixes that are removed from the end of file names when adding those files to the package. Used on sub-package level. • Used for creating sub-packages with conflicting files, such as different variants of the same content (eg minimal and full versions of the same software).
BuildArch or **BuildArchitectures**	Specifies the architecture which the resulting binary package will run on. • Typically this is a CPU architecture like sparc, i386. • The string 'noarch' is reserved for specifying that the resulting binary package is architecture independent for example, if written entirely in an interpreted programming language. • Typical platform independent packages are html, perl, python, java, and ps packages. • As a special case, BuildArch: noarch can be used on sub-package level to allow eg. documentation of otherwise arch-specific package to be shared across multiple architectures. • If not set, the package automatically inherits the Architecture of the machine on which it is built, for example x86_64. ⚠️ Note that BuildArch causes the spec parsing to recurse from the start, causing any macros before that line to be expanded twice. This can yield unexpected results, in particular with `%global`.		
ExcludeArch	Package is not buildable on architectures listed here. • Used when software is portable across most architectures except some, for example due to endianess issues.	**ExclusiveArch**	Package is only buildable on architectures listed here. • For example, it's probably not possible to build an i386-specific BIOS utility on ARM, and even if it was it probably would not make any sense.
ExcludeOS	Package is not buildable on specific OSes listed here.	**ExclusiveOS**	Package is only buildable on OSes listed here.
BuildRequires	Capabilities required to build the package. A comma or whitespace-separated list of packages required for building the program written in a compiled language. • Build dependencies are identical to install dependencies except: 1) they are prefixed with build (e.g. BuildRequires: rather than Requires:) 2) they are resolved before building rather than before installing. So, if you were to write a specfile for a package that requires gcc to build, you would add BuildRequires: gcc If your package could not be built w/o a specific version of the libraries to access an ext2 file system, you could express this as BuildRequires: e2fsprofs-devel = 1.17-1 • There can be multiple entries of **BuildRequires,** each on its own line in the SPEC file.		

Provides	If **Provides** is added to a package, the package can be referred to by dependencies other than its name.			
Requires • **Requires(pre)** • **Requires(preun)** • **Requires(post) …** Multiple qualifiers can be supplied separated by comma, as long as they're not semantically contradictory: **meta** qualifier contradicts any ordered qualifier, eg **meta** and **verify** can be combined, and **pre** and **verify** can be combined, but **pre** and **meta** can not. As noted above, dependencies qualified as install-time only (pretrans, pre, post, posttrans or combination of them) can be removed after the installation transaction completes if there are no other dependencies to prevent that. This is a common source of confusion.	Capabilities this package requires to function at all. Besides ensuring required packages get installed, this is also used to order installs and erasures. 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.			
	Additional context can be supplied using Requires(qualifier) syntax, accepted qualifiers are:			
	pre relates to %pre scriptlet	Denotes the dependency must be present in before the package is is installed, and is used a strong ordering hint to break possible dependency loops. A pre-dependency is free to be removed once the install-transaction completes.	**post** relates to %post scriptlet	Denotes the dependency must be present right after the package is is installed, and is used a strong ordering hint to break possible dependency loops. A post-dependency is free to be removed once the install-transaction completes.
	preun relates to %preun scriplet	Denotes the dependency must be present in before the package is is removed, and is used a strong ordering hint to break possible dependency loops.	**postun** relates to %postun scriptlet	Denotes the dependency must be present right after the package is is removed, and is used a strong ordering hint to break possible dependency loops.
	pretrans relates to %pretrans and %preuntrans scriptlet	Denotes the dependency must be present right after the package is is removed, and is used a strong ordering hint to break possible dependency loops.	**posttrans** relates to %posttrans and %postuntrans scriptlet	Denotes the dependency must be present at the end of transaction, ie cannot be removed during the transaction. As such, it does not affect transaction ordering. A posttrans-dependency is free to be removed after the the install-transaction completes.
	verify	Relates to %verify scriptlet execution. As %verify scriptlet is not executed during install/erase, this does not affect transaction ordering.	**interp**	Denotes a scriptlet interpreter dependency, usually added automatically by rpm. Used as a strong ordering hint for breaking dependency loops.
(since rpm >= 4.16) ➨	**meta**	Denotes a "meta" dependency, which must not affect transaction ordering. Typical use-cases would be meta-packages and sub-package cross-dependencies whose purpose is just to ensure the sub-packages stay on common version.		

Autoprov **Autoreq**	Control per-package automatic dependency generation for **Provides** and **Requires**. • Accepted values are: `1`/`0` or `yes`/`no`, • default is always `yes`.	**Autoreqprov**	Autoreqprov is equal to specifying Autoreq and Autoprov separately.
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.	**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.
DistTag		**VCS**	
Distribution		**ModularityLabel**	
Buildsystem	Automatically populate the spec build scripts for the given build system, such as `Buildsystem: autotools". • See declarative build documentation for more details.	**Packager**	Optional package distribution/vendor/maintainer name / contact information. • Rarely used in specs, typically filled in by buildsystem macros.
BuildRoot	Obsolete and unused in rpm >= 4.6.0, but permitted for compatibility with old packages that might still depend on it. • Do not use in new packages.	**Prereq**	Obsolete, do not use.
Epoch	Optional numerical value which can be used to override normal version-release sorting order. • It's use should be avoided if at all possible. Non-existent epoch is exactly equal to zero epoch in all version comparisons.	**Icon** (obsolete)	Used to attach an icon to an rpm package file. Obsolete.
BuildConflicts	Capabilities which conflict, ie cannot be installed during the package package build. For example if somelib-devel presence causes the package to fail build, you would add: BuildConflicts: somelib-devel		

Body section items	In Emacs rpm-spec-mode, the **rpm-spec-section-face** controls the face used to show section items. ☝ RPM holds an embedded Lua interpreter allowing all scripts to be written in Lua. See **Lua in RPM @ rpm.org**.		
%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. • See the `%setup` macro below.	**%build**	Command or series of commands for building the software into machine code (for compiled languages) or byte code (for some interpreted languages).
%install ☝ This is only run when **creating a package**, not when the end-user installs the package.	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.		
%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 installed in the end user's system. See Common RPM macros in the %file section.	**%changelog**	A record of changes that have happened to the package between different **Version** or **Release** builds.

Scriptlets @ Fedora **Scriptlets @ Red Hat** • How to turn off script execution • See triggers directives	The RPM sections which allow packages to run code on installation and removal. These chunks of code are called ***scriptlets***. The **scriptlets syntax** is similar to the **%build** and **%install** sections. • The **%pretrans**, **%pre, %post**, **%preun**, **%postun**, **%posttrans** are called by RPM install, upgrade and uninstall. • They are passed an argument (accessible as **$1** in the scriptlet) that can be used to identify: **==0** := uninstall, **==1** := install, **>=1** := upgrade • The **order of execution of scriptlets** depends on whether it's an installation, and upgrade or an un-install.			
☝ Use other script interpreter ➡ **order of execution of scriptlets** ⬇	The **-p** script option enables writing scripts that are executed by specific interpreter instead of the default /bin/sh. Example: **%post -p /usr/bin/python3**	**RPM install**	**RPM upgrade**	**RPM uninstall**
%pretrans	Executed just before installing or removing any package. • It can NOT have any dependency. • It's best avoided. If absolutely required, it must be written in **lua**. See **Lua in RPM @ rpm.org**	$1==1	$1>=1	(N/A)
%pre	Run before a package is installed/upgraded.	$1==1	$1>=1	(N/A)
%post	Run after a package is installed/upgraded.	$1==1	$1>=1	(N/A)
%triggerin of other packages				
%triggerin of new packages				
%triggerin of old package				
%triggerun of other packages				
%preun	Run just before uninstalling the package from the target system.	(N/A)	$1==1	$1==0

%postun	Run just after the package was uninstalled from the target system.	(N/A)	$1==1	$1==0
%triggerpostun				(N/A)
%posttrans	Executed at the end of the transaction.	$1==1	$1>=1	(N/A)

conditionals in RPM spec files:	allow conditional blocks of code to be used depending on various properties such as conditional expressions, architecture and operating system.									
• **Operators:**	**Logical:**	&&, ||, !	**Relational:**	`!=, ==, <, >, <=, >=`	**Arithmetic:**	`+, -, /, *`	**Ternary:**	`? :`	**Parentheses**	

• **expression:**	**%if**	**%else**	**%endif**	Test for the existence of a macro, like in:	`%if %{defined with_foo} && %{undefined with_bar}`
				string comparison, like in:	`%if "%{optimize_flags}" != "none"`
				mathematical statement, like in:	`%if 0%{?fedora} > 10 || 0%{?rhel} > 7`

• **architecture:**	**%ifarch** **%ifnarch**	**%elifarch**		To select logic for multiple platforms:	`%ifarch s390 s390x`
					`BuildRequires: s390utils-devel`
					`%endif`

• **Operating System:**	**%ifos** **%ifnos**	**%elifos**			

Macros		
%global	A macro can be declared into the global scope as follows:	`%global <name>[(opts)] <body>`
	An important and useful feature of **%global** is that <body> is expanded at the time of definition, as opposed to time of use with regular macros. This is important inside parametric macros because otherwise the body could be referring to macros that are out of scope at the time of use, but also useful to avoid re-expansion of expensive macros.	
%setup		
%license	The macro identifies the file listed as a LICENSE file and it will be installed and labeled as such by RPM.	`%license LICENSE`
%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. • When code examples are included, care should be taken to remove executable mode from the file.	`%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.	`%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.	
	`%config(noreplace) %{_sysconfdir}/%{name}/%{name}.conf`	
%attr		
%defattr		