RPMSpecsEditor Users Guide

Copyright © 2025 by Select Calibration Inc. All rights reserved.

Table of Contents

Introduction

The *RPMSpecsEditor* utility allows the user to create a SPEC file that is read by the *RPMBuild* utility in order to create an RPM file. An RPM file is a package format used by the RPM Package Manager, primarily for installing software on GNU/Linux distributions such as Red Hat, Fedora, SUSE Linux, and CentOS. The *RPMSpecsEditor* is geared toward Qt applications containing various libraries and plugins but should be adaptable to other frameworks if needed.

Overview

The *RPMSpecsEditor* utility consists of a main window with tabs for variables, expressions, libraries, dependency requirements, change log, and options.

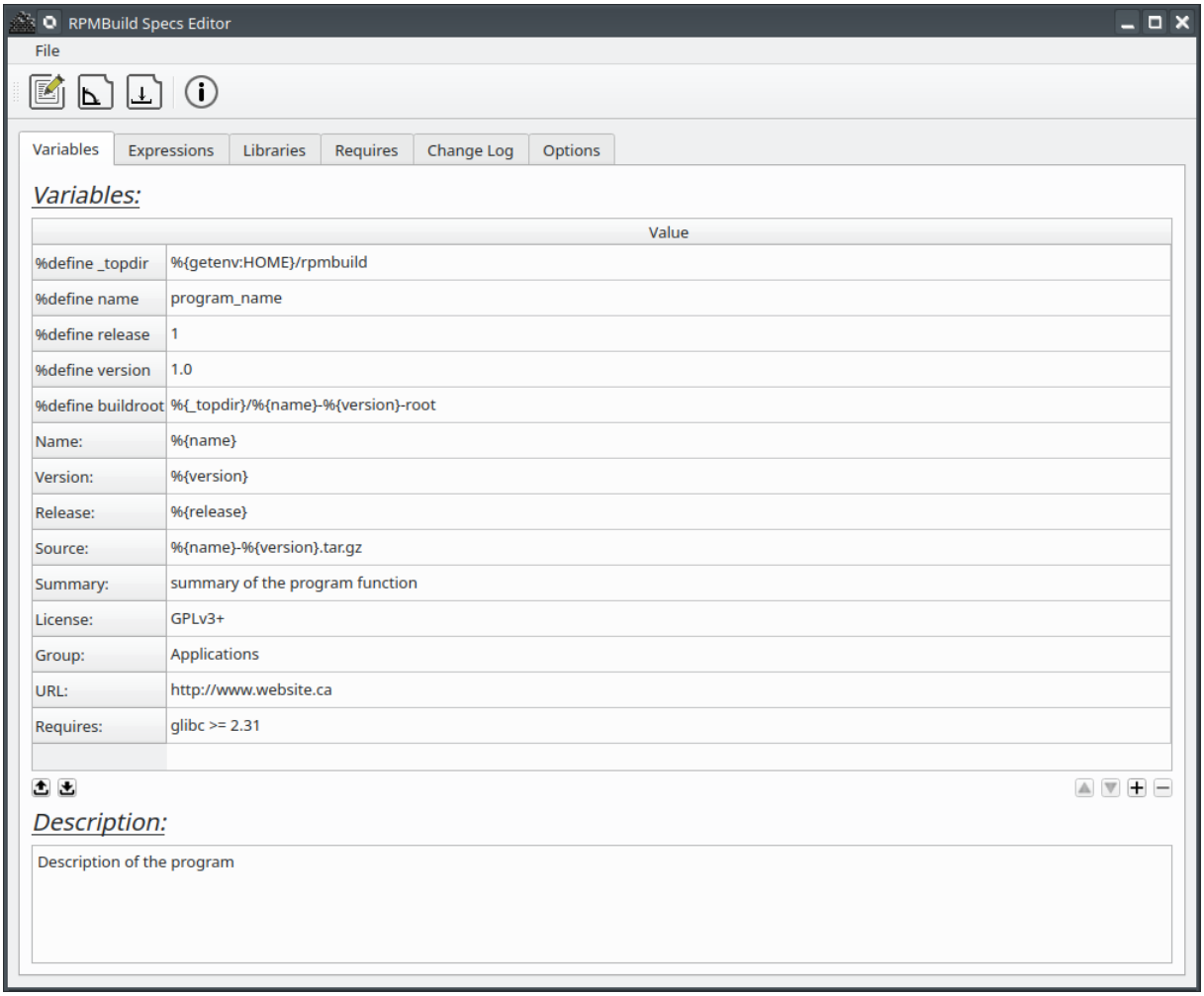


Illustration 1: RPMSpecs utility main window.

SPEC File

The SPEC file is used by the *RPMBuild* utility in order to create an RPM file. The SPEC file has uniquely identified sections that are used in the build process.

Table 1: RPM SPEC file partial list of sections:

Variable	*Description*
%description	Description of the program.
%prep	Steps necessary to prepare the software for building.
%build	Commands to build the software.
%install	Commands to install the software.
%post	Commands to be executed following the installation of the software.
%postun	Commands to be executed following the removal of the software.
%files	List of files that will be included in the package.
%changelog	Record of changes made for each release.

Variables

The SPEC file can contain variables that can be used for various purposes. For example, the variable *%{buildroot}* is defined as *%{_topdir}/%{name}-%{version}-root* and is easier and cleaner to write as opposed to the expanded version.

Examples of variables written at the beginning of a SPECs file:

```
%define _topdir                 %{getenv:HOME}/rpmbuild
%define name                    program_name
%define release                 1
%define version                 1.0
%define buildroot               %{_topdir}/%{name}-%{version}-root
Name:                           %{name}
Version:                        %{version}
Release:                        %{release}
Source:                         %{name}-%{version}.tar.gz
Summary:                        summary of the program function
License:                        GPLv3+
Group:                          Applications
URL:                            http://www.website.ca
Requires:                       glibc >= 2.31
Requires:                       libz1
Requires:                       fontconfig
Requires:                       libfreetype6
Requires:                       libglvnd
Requires:                       libXext6
Requires:                       libX11-6
Requires:                       libexpat1
Requires:                       libbz2-1
```

```
Requires:                       libpng16-16
Requires:                       libxcb1
Requires:                       libXau6
Requires:                       libdrm2
Requires:                       libX11-xcb1
Requires:                       libxcb-icccm4
Requires:                       libxcb-image0
Requires:                       libxcb-shm0
Requires:                       libxcb-util1
Requires:                       libxcb-keysyms1
Requires:                       libxcb-randr0
Requires:                       libxcb-render-util0
Requires:                       libxcb-render0
Requires:                       libxcb-shape0
Requires:                       libxcb-sync1
Requires:                       libxcb-xfixes0
Requires:                       libxcb-xinerama0
Requires:                       libxcb-xkb1
Requires:                       libxcb-xinput0
Requires:                       libSM6
Requires:                       libICE6
Requires:                       libxkbcommon-x11-0
Requires:                       libxkbcommon0
Requires:                       libuuid1
Requires:                       libxcb-glx0
```

Variables are added to the SPEC file in the same order listed in the *RPMSpecsEditor* variables section. The order is important as variables must be defined before they can be used.

Some variables, such as *%{_topdir}*, have special meaning for the *RPMBuild* utility. Commonly used RPM macro variables are shown in the following tables.

Table 2: RPMBuild utility macro variables:

macro	*Definition*	*Comment*
%{buildroot}	%{_buildrootdir}/%{name}-%{version}-%{release}.%{_arch}	same as $BUILDROOT
%{_topdir}	%{getenv:HOME}/rpmbuild	
%{_builddir}	%{_topdir}/BUILD	
%{_rpmdir}	%{_topdir}/RPMS	
%{_sourcedir}	%{_topdir}/SOURCES	
%{_specdir}	%{_topdir}/SPECS	
%{_srcrpmdir}	%{_topdir}/SRPMS	
%{_buildrootdir}	%{_topdir}/BUILDROOT	

Table 3: General macro variables:

macro	*Definition*	*Comment*
%{_sysconfdir}	/etc	

macro	*Definition*	*Comment*
%{_prefix}	/usr	can be defined to /app for flatpak builds
%{_exec_prefix}	%{_prefix}	default: /usr
%{_includedir}	%{_prefix}/include	default: /usr/include
%{_bindir}	%{_exec_prefix}/bin	default: /usr/bin
%{_libdir}	%{_exec_prefix}/%{_lib}	default: /usr/%{_lib}
%{_libexecdir}	%{_exec_prefix}/libexec	default: /usr/libexec
%{_datadir}	%{_datarootdir}	default: /usr/share
%{_infodir}	%{_datarootdir}/info	default: /usr/share/info
%{_mandir}	%{_datarootdir}/man	default: /usr/share/man
%{_docdir}	%{_datadir}/doc	default: /usr/share/doc
%{_rundir}	/run	
%{_localstatedir}	/var	
%{_sharedstatedir}	/var/lib	
%{_lib}	lib64	lib on 32bit platforms

Requires Variable Entry

The *Requires* entries describe what is needed on the target system to run the application. The default list of variables include an entry for the *glibc* version which must be equal to or greater than some value so an RPM is created on a system that has a current *glibc* version greater than that of the target system it will not run. The *Require* entries ensures that the minimum requirements for the application are met and will deal with this during installation instead of simply not working.

With respect to *glibc* the following command can be run to determine what is currently installed:

```
ron@linux-4o1p:~> ldd --version
ldd (GNU libc) 2.31
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Written by Roland McGrath and Ulrich Drepper.
```

Additional *Requires* variable entries can be automatically added to the specs file based on a scan of the selected plugin and libraries folder. This is done by default but can be disabled if necessary.

Variable Editor

Variables can be manipulated a variety of way. In addition to adding or removing entries the order can be changed using the UP/DOWN buttons. Variable entries can also be archived and restored as base settings.

Icon	*Description*
	Store current variable entries and values.

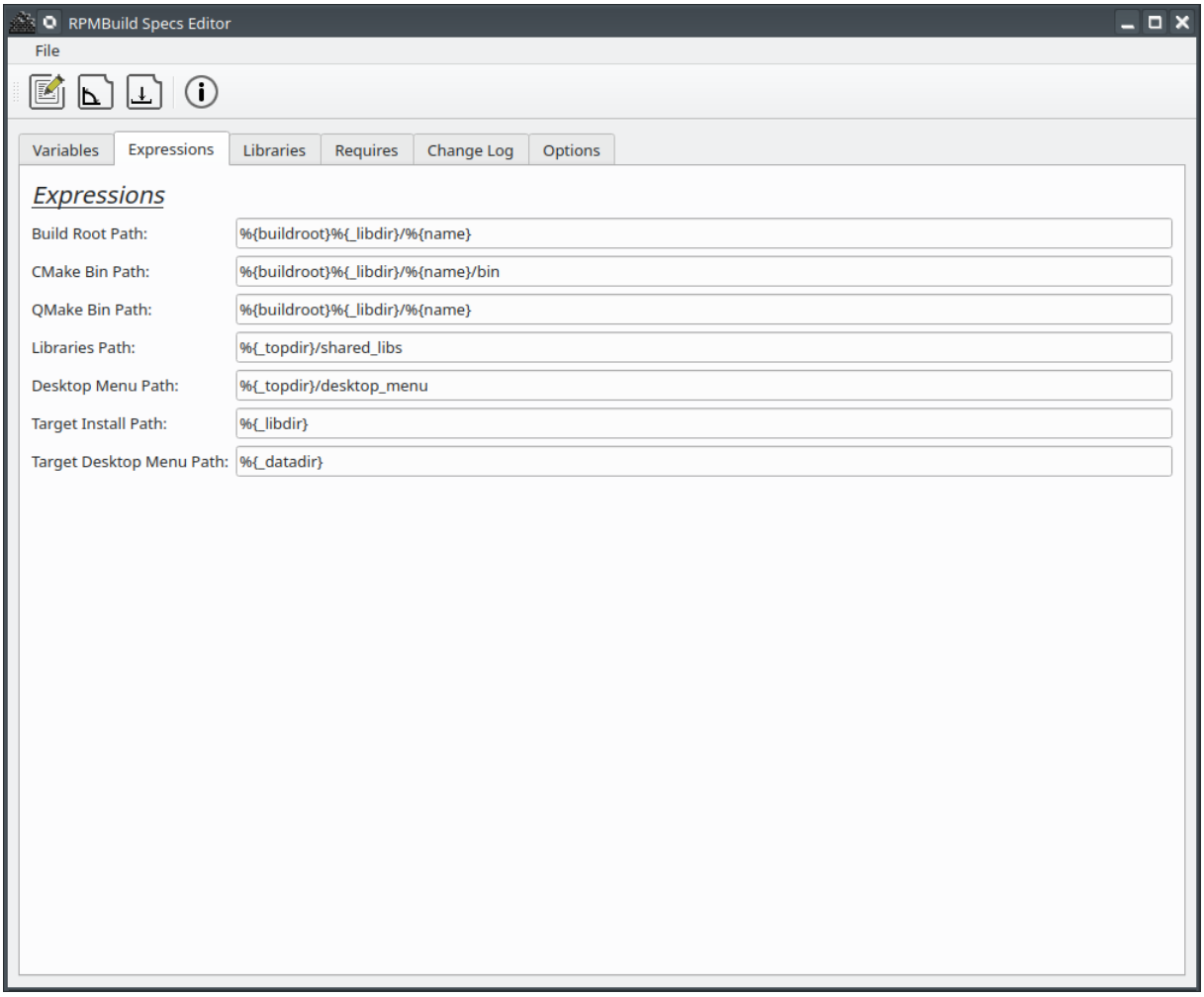
	Restore the current variables and entries from previously saved values.
	Move the selected entry up.
	Move the selected entry down.
	Add a new variable entry.
	Remove a selected variable entry.

Description

The description section describes the purpose of the program and any other relevant information. The description can be quite lengthy if needed.

Expressions

The Expression section contains entries that are used in various sections of the SPEC file. For example, the entry *Build Root Path* has a default value of *%{buildroot}%{_libdir}/%{name}* and is used to define the location of the application image prior to installation.



The Expression entries have the following roles:

Table 4: Expression Entries:

Name	*Description*
Build Root Path	Location where the software image is created.
CMake Bin Path	Location of the compiled executable when processed using CMake.
QMake Bin Path	Location of the compiled executable when processed using QMake.
Libraries Path	Location of libraries and plugins used by the software.
Desktop Menu Path	Location of the files necessary for the desktop menu (*.desktop, *.png).
Target Install Path	Target location of the software when installed.
Target Desktop Menu Path	Target location of the menu entries when installed.

The following is an example of where the *Build Root Path* entry is used in the generated SPEC file:

```
...
%build

qmake PREFIX=%{buildroot}%{_libdir}/%{name}
make

%install

mkdir -p %{buildroot}%{_libdir}/%{name}
mkdir -p %{buildroot}%{_libdir}/%{name}/platforms
mkdir -p %{buildroot}%{_libdir}/%{name}/lib
mkdir -p %{buildroot}%{_libdir}/%{name}/xcbglintegrations
mkdir -p %{buildroot}%{_datadir}/applications
mkdir -p %{buildroot}%{_datadir}/pixmaps

make install

patchelf --set-rpath '$ORIGIN/lib' %{buildroot}%{_libdir}/%{name}/%{name}
...
```

Libraries

The Libraries section allows selection of folders and specific library files to be included with the software. The folders can be used for Qt plugins or for dynamically linked library files.

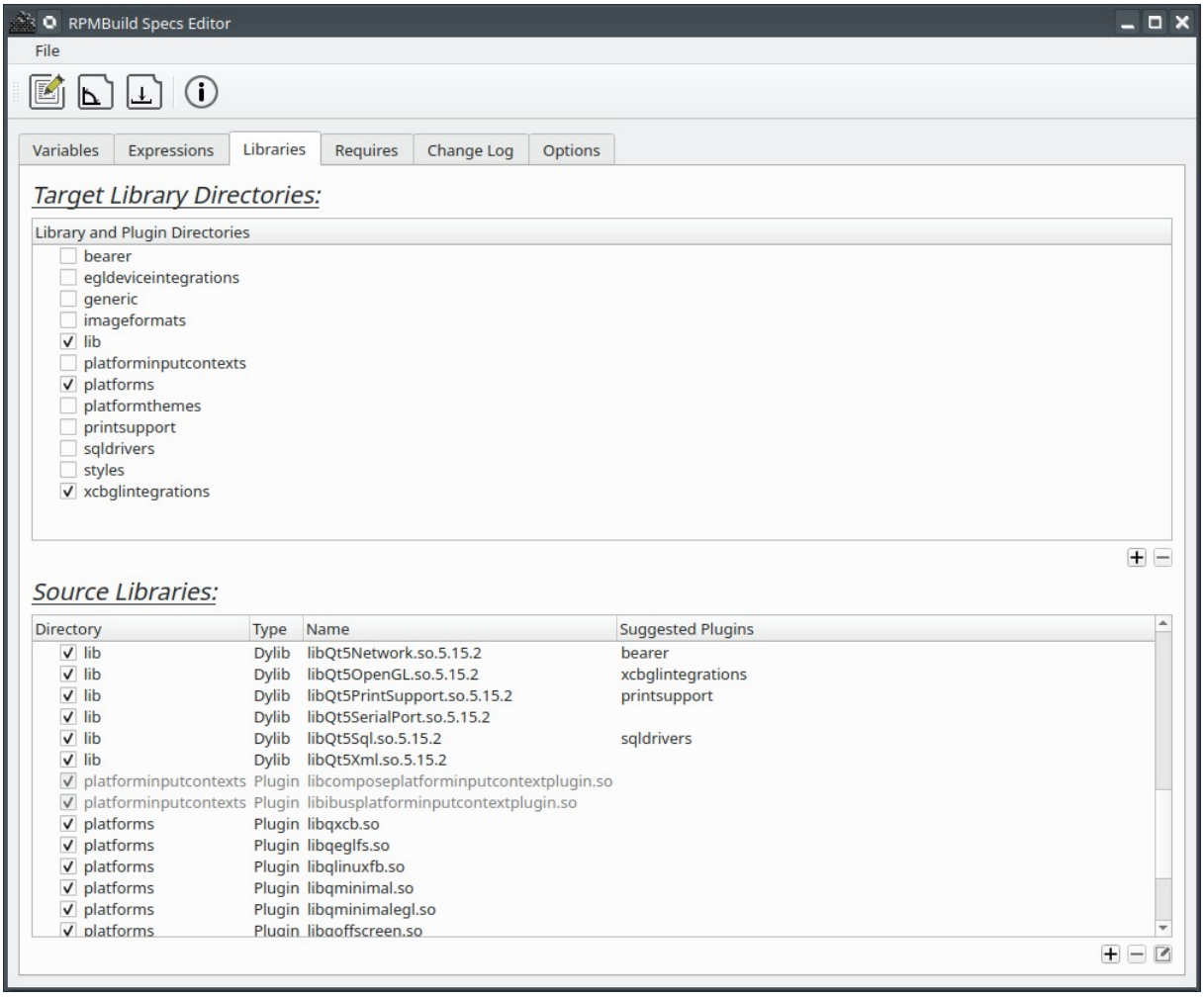


Illustration 2: Target and source libraries.

Target Library Directories

The target library directories represents the name of the directory that will be included in the RPM installation. In the example shown in illustration 2 the directories *platforms*, *lib*, and *xcbglintegrations* will be installed as sub directories of the installed software.

Source Libraries

The source libraries list the individual files that will be copied to the associated target directory of the generated RPM file. For plugin files a version number is not part of the file name where dynamic libraries is expected to have the full version numbers as part of the name.

The Library entries can be manipulated with the following functions:

Table 5: Library Functions:

Icon	*Description*

	Add target directory or library.
	Remove target directory or library.
	Modify name of library. For multiple library selections the version numbers can be changed without having to modify each entry individually. See section Source Library Versions for details.

Source Library Versions

The source dynamic libraries are expected to have three version values in the format of lib<some_name>.so.1.2.3. In the event that the Qt version has changed all library files can be updated to a specific version by selecting all libraries and change the version numbers to match what is needed.

Requires

The *Requires* section contains details of the packages that would need to be installed on the target system to provide the required libraries used by the application. This option scans the shared libraries folder, including plugin libraries, as opposed to just the executable which is done automatically when the rpm file is created.

Installing the RPM file directly should identify and automatically install all required libraries defined in the RPM file. The debian version created by SCI uses the Alien utility to covert the RPM to a DEB package which does not transfer information related to required libraries so manual installation of additional packages may be necessary.

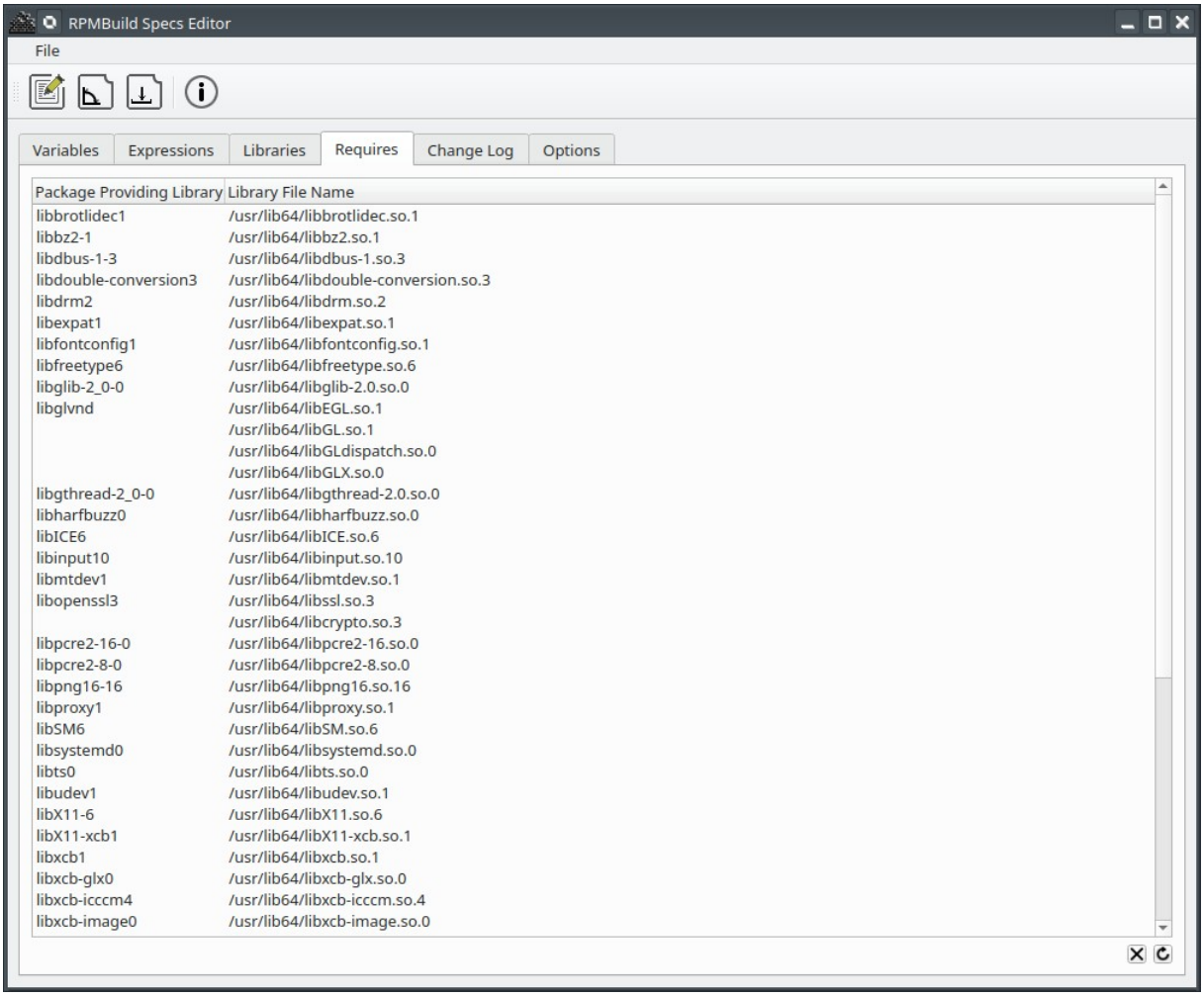


Illustration 3: Requires section.

Table 6: Requires Functions:

Icon	*Description*
	Update libraries and packages from the *shared_libs* path defined in *Options* section.
	Clear all listed libraries and packages.

Change Log

The *Change Log* section contains details of the revision history for the program. Each entry consist of two parts where the first is the date and author name followed by one or more changes. The line entries must start with either '*' or '-' to identify a new entry or entry change. The order the entries must be newest revisions at the top and oldest revisions at the bottom. The following

shows an example of a change log with the newest entry on top:

```
* Wed Sep 3 2025 <author name>
- Fixed various bugs
- Added feature to do something better.
- Cleaned up something.
* Sun Nov 6 2016 <author name>
- Initial RPM release
```

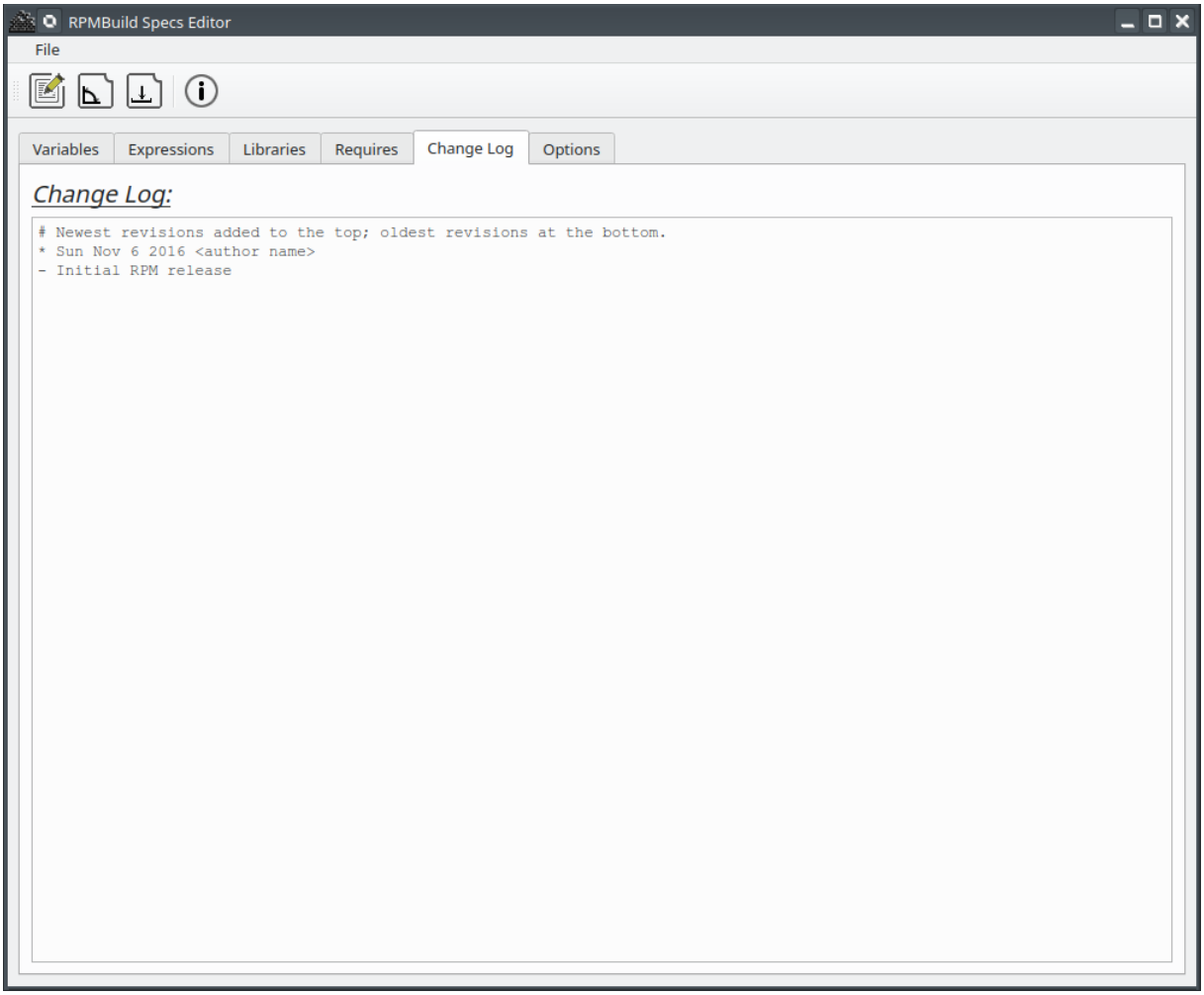


Illustration 4: Change Log section.

Options

The *Options* section contains entries that will impact the generated SPEC file.

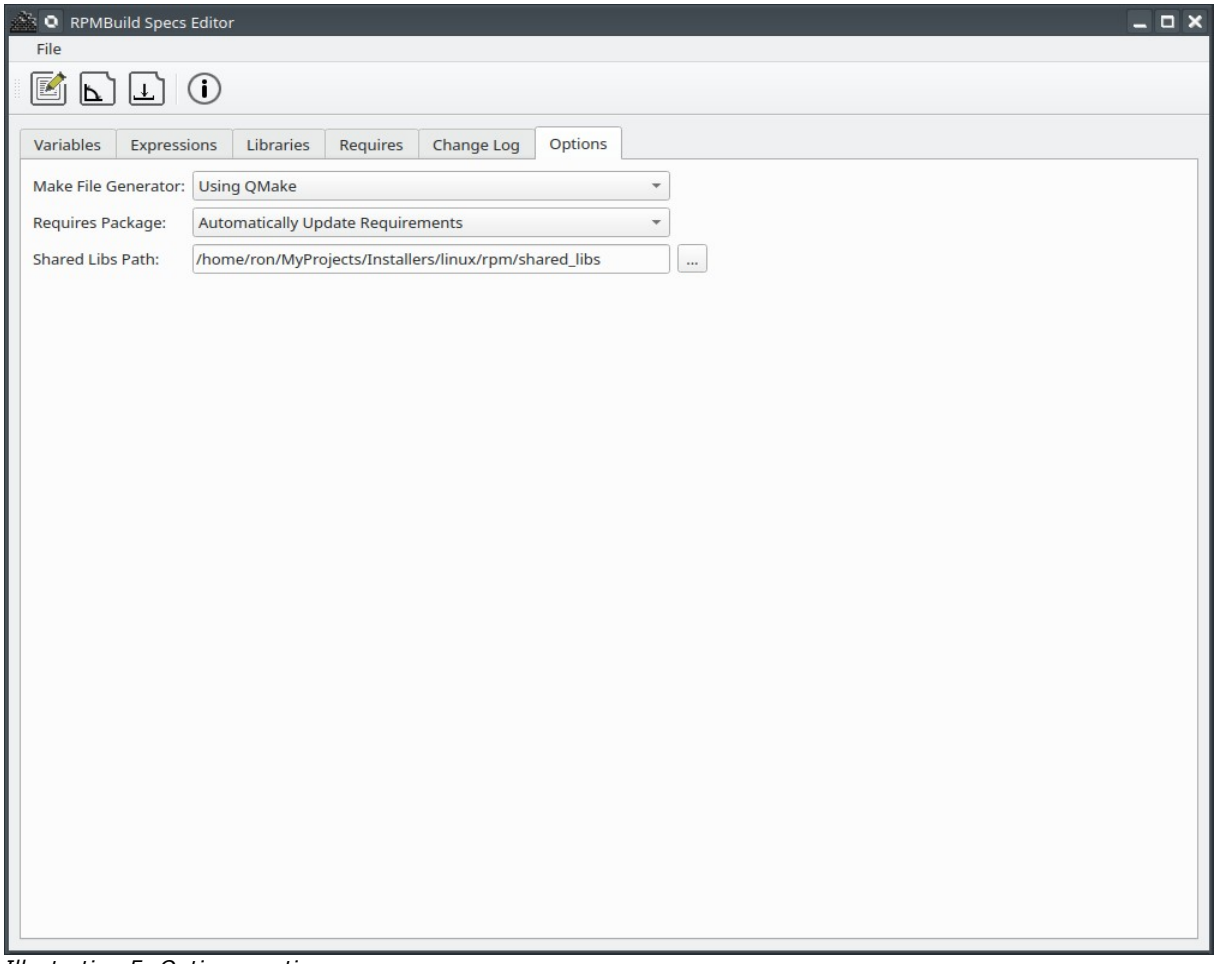


Illustration 5: Options section.

Table 7: Options:

Name	*Description*
Make File Generator	Selection for the type of make-file generator used in the spec file.
Requires Package	Method for dealing with required packages used by the libraries and plugins.
Shared Lib Path	Location where the shared libraries and plugins are located.

RPM Creation Example

This example describes the entire process to create an RPM file from a Qt application using Qmake to create the make file. At a minimum it is assumed that a version of Qt is installed and utilities such as *GCC*, *make*, *patchelf*, and *RPMBuild* are also installed.

RPMBuild Directory Setup

A suitable directory structure must be created in order to use the *RPMBuild* utility. In this example the build folder is called rpmbuild and is below the users home folder.

```
├── BUILD
│   └── SOURCES
├── RPMS
│   └── x86_64
├── SOURCES
├── SPECS
├── SRPMS
├── desktop_menu
├── shared_libs
│   └── bearer
│   └── egldeviceintegrations
│   └── generic
│   └── imageformats
│   └── lib
│   └── platforminputcontexts
│   └── platforms
│   └── platformthemes
```

The folders *desktop_menu* and *shared_libs* contain the menu data and copies of libraries and plugins that can be used to create the RPM file.

Source Files

The source file is a tar archive of the application and associated files. For Qt this will include the *PRO* file or *CmakeLists.txt* file, source files, resource files, and anything else related to the application. The naming of the archive (internal and external) is important. For example, an application with the name MyApp version 1.0 should be in a directory called MyApp-1.0 and then compressed with the tar command and saved in the SOURCES folder with the same naming convention.

```
tar -czf MyApp-1.0.tar.gz MyApp-1.0
```

SPEC File

The SPEC file is created by the *PRMSpecsEditor* utility and saved in the SPECS folder. The variable names for program and version must be updated to match the name and version of the application.

Desktop Menu Items

The desktop menu entries describe the application in some detail and will be added to the operating system launcher when installed. There are two files where the first is a text file with the extension of .desktop and the second is a PNG image file for the application icon. Both will have the name of the application (i.e. MyApp.desktop and MyApp.png).

Typical Desktop Entry:

```
[Desktop Entry]
Categories=Utility;
Encoding=UTF-8
```

```
Name=Description Of MyApp
GenericName=My Application
Comment=My Application
Exec=MyApp
Icon=MyApp.png
Terminal=false
StartupNotify=true
Type=Application
```

Shared Files

Depending on the nature of the utility it may be necessary to create a link to other locations or files in order to compile the target application. For example, the directory ~/MyProjects/core is needed in order to compile many of the applications used by SCI. Copying the necessary files and folders to the build folder used by *RPMBuild* is one option but a better option is to create links to anything external that might be needed.

```
ln -s ~/MyProjects/core ~/rpmbuild/BUILD/core
```

In the above example a link is created to a required directory containing files necessary for compilation of the application.

Build RPM

The build process is done from the root of the *RPMBuild* directory. If using the terminal the command would be something like this:

```
rpmbuild -v -bb --clean SPECS/MyApp.spec
```

where:

```
-v        = verbose output
-bb       = build only binary packages
--clean   = remove build tree after packages are made
SPECS/xxx = name of specs file
```

If source file RPM'S are needed then use the appropriate command line option. In this example only an executable is required.

If creation is successful there will be an RPM file created in the RPMS folder with the name MyApp-1.0-1.x86_64.rpm (assuming the target is x86_64).

RPM Signing

The RPM files can be signed to show that the application was created by a credible source and that modifications have not been made following the creation of the RPM. The command to do this is:

```
rpm --resign MyApp-1.0-1.x86_64.rpm
```

This will only work if a suitable signing certificate exists. If the goal is for large scale distribution than a certificate should be acquired allowing this software to be installed anywhere. If the RPM is unsigned or signed with a locally generated certificate then the user installing the RPM will have the option to abort installation due to not being signed or signed with an unknown certificate.

Troubleshooting

Following the installation of an application on a target system the application may not launch properly if dependencies are missing and this happens particularly with the plugins.

Dependency problems should only be an issue when installing on debian based systems as the debian package is created from the rpm file using the Alien utility and appears to ignore the 'Requires' entries from the RPM package. The solution is to create a similar utility for debian systems that creates a debian package directly (a future project idea).

Example Application Launch Problem

The *ballbarviewer* application was installed on a debian based GNU/Linux system and did not launch properly from the application menu following the installation. When opened from a terminal window the following was returned:

```
ron@ron-VirtualBox:$ ballbarviewer  ← launch application from terminal window.

qt.qpa.plugin: Could not load the Qt platform plugin "xcb" in "" even though it
was found.
This application failed to start because no Qt platform plugin could be
initialized. Reinstalling the application may fix this problem.

Available platform plugins are: eglfs, linuxfb, minimal, minimalegl, offscreen,
vnc, xcb.

Aborted (core dumped)
```

To see detailed information related to plugins it may be necessary to set the environment variable to show debug information for Qt plugins. This is done from the terminal window prior to trying to launch the application.

```
export QT_DEBUG_PLUGINS=1
```

Finding Missing Libraries

The *ballbarviewer* application is typically installed in the folder */usr/lib64/ballbarviewer* and all libraries and plugins will be below this directory.

```
ron@ron-VirtualBox:$ ls /usr/lib64/ballbarviewer/platforms
libqeglfs.so  libqlinuxfb.so  libqminimalegl.so  libqminimal.so  libqoffscreen.so
libqvnc.so  libqxcb.so
```

The missing libraries needed by libqxcb.so can be found using the GNU/Linux LDD command. The output was piped through *grep* looking only for lines containing the text '*not found*'.

```
ron@ron-VirtualBox:$ ldd /usr/lib64/ballbarviewer/platforms/libqxcb.so | grep
"not found"
   libxcb-xinerama.so.0 => not found
   libxcb-xinerama.so.0 => not found
```

Installing Missing Libraries

Once the missing libraries have been identified they can be installed. The name of the library is usually not the name of the package containing it but this can be determined by a variety of methods. In this example the package name is '*libxcb-xinerama0*'.

```
ron@ron-VirtualBox:$ sudo apt-get install libxcb-xinerama0
[sudo] password for ron:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  libxcb-xinerama0
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 0 B/5,260 B of archives.
After this operation, 37.9 kB of additional disk space will be used.
Selecting previously unselected package libxcb-xinerama0:amd64.
(Reading database ... 184296 files and directories currently installed.)
Preparing to unpack .../libxcb-xinerama0_1.14-2_amd64.deb ...
Unpacking libxcb-xinerama0:amd64 (1.14-2) ...
Setting up libxcb-xinerama0:amd64 (1.14-2) ...
Processing triggers for libc-bin (2.31-0ubuntu9.18) ...


ron@ron-VirtualBox:$ ballbarviewer  ← worked this time
```

Revision History

Date	*Version*	*Changes*
Oct 2, 2025	1.0	New Program