OpenNA RPM HOWTO

Version: 1.2
Issue Date: 2004-03-07

Written by Gerhard Mourani <gmourani@openna.com>
Edited by Joe Rodriguez Jr. <jrodriguez@mypcbox.com>

Open Network Architecture, Inc.
11090 Drouart
Montreal, Quebec
H3M 2S3
Canada

Table of Contents

Abstract

In the effort to produce a quality product and corresponding documentation, this document was constructed by Open Network Architecture, Inc, its affiliates, partners, and general user community to serve as step-by-step instruction on how to compile a RPM package that will integrate well with the OpenNA Linux distribution of GNU/Linux. The document assumes that code being compiled is either from a previous RPM source or from the original tar source.

It is important to note that RPM itself is a software package build and deployment tool which is used to encapsulate all aspects of program package into a single resultant RPM package for deployment. This resultant RPM File contains all the necessary information need to install the compiled program(s) on a given system along with the necessary pre and post operational instruction needed by the program on the system.

For sake in understanding this document, it is important to distinguish the difference between source (.src.rpm) and binary (.i686.rpm) packages. The first contains the complete original source tree from the programmer, plus all the information the packager has added in order to configure, compile, and install the program. This information generally consists of a SPEC file (the file used to tell rpm what operations to perform in order to create the package) along with patches, if needed.

The second contains the compiled binary and all the files that will be installed on the target system. It also contains the procedures used to put the files in their correct location on the system and the necessary actions/procedures to perform in order to have the program operational.

In addition, this documents takes you through a detailed procedure for building a binary package for source using RPM however does not address the extensibility of RPM build and deployment tool itself. The task of learning the inner workings of the RPM tool itself is left to the reader and is not openly addressed in this document.

Install the software

Although RPM was originally designed to work with RedHat Linux, it also works on other rpm-based distributions: OpenNA Linux, Mandrake, Suse, Caldera, etc; RPM is already installed on these systems. In any instance please be aware that the binary RPM you will build for OpenNA Linux may not work across the distributions, although OpenNA makes every effort possible to stay compatible with Red Hat.

In order to compile RPM packages from source code, you will first need to install the OpenNA Linux Workstation type install. This particular installation type takes care of providing all the required compiler packages and other important development libraries and headers that are needed to compile source code on your system. Please note that if you also want to build RPM for **G**raphical **U**ser **I**nterface (GUI), then just say "Yes" to GUI during the setup of your OpenNA Linux Workstation to have the additional GUI development packages installed on to your system.

As previously mentioned, the OpenNA Linux Workstation installation type will have all the required development packages and compilers installed for you. At this point everything related to RPM building will be located under the "`/usr/src/openna/`" directory path on your Workstation. The following provides a brief description about what each subdirectory under the "`/usr/src/openna/`" directory path is used for.

In order for you to successfully build complete software packages using RPM you must know the RPM build structure. This structure is defined as follows:

```
/usr/src/openna/
            BUILD/
            RPMS/i686/
            SOURCE/
            SRPMS/
            SPECS/
```

- The "**BUILD**" directory is used to unpack the source code and build it.

- The "**RPMS**" directory is used to keep successfully built RPM packages. This is where the result of your built RPMS will be stored once successfully compiled.

- The "**SOURCE**" directory is where the source code and any possible patches, files, etc you will need to build the package will be located.

- The "**SRPMS**" directory is used to keep a successful build of SRPMS packages. This is where the result of your SRPMS build will be stored once successfully compiled.

- The "**SPECS**" directory as its name implies is used to store all SPEC files used to build RPM packages. This is also the directory under which you should move in before starting to configure the SPEC file and build it with the appropriated rpm command as shown further down in this document.

Building RPM from an existing source RPM

This is generally the case for building software packages which are already included in the distribution. When you find the source RPM you wish to modify for OpenNA Linux distribution , locate the RPM source file and install it by issuing the following command: "**`rpm -Uvh packagename-version.src.rpm`**". The resultant effect for issuing this command will be the installation of all source files into the RPM directory "`/usr/src/openna/`".

At this point, all that is needed is to modify its SPEC file and then rebuild the package.

Inside the SPEC file

The SPEC file contains all the information needed by RPM to:

1. Compile a program and build source and binary RPMs,
2. Install and uninstall the program on the user's machine.

The fact that these two types of information are merged into a single file may be quite confusing for most beginners. Actually, this is due to the source tar tree, which already contains this information. As the installation procedure is extracted from the installation process generally run by "**`make install`**" into the source tree, both parts are tightly linked.

As you build more and more RPMs, you will find that there are some options that have not addressed in this document. As mentioned in the Abstract section of this document, RPM is an extremely versatile tool, so it is the reader's responsibility in discovering how to use these additional options. In addition, when first building an RPM from source it is always good practice to open up the SPEC files to look at them and see how they work.

As you will discover about most OpenNA source RPM packages, the OpenNA Linux SPEC files are divided into 7 distinct sections. Each section is responsible for specifically doing or providing something during the build process of the RPM package. The following provides a listing of all sections of an OpenNA Linux SPEC file:

- The Header section
- The Prep section
- The Build section
- The Install section
- The Clean section
- The Files section
- The Changelog section

Every OpenNA Linux SPEC files that you expect to build should have the above sections defined and used to be compatible with the way OpenNA writes and builds RPM packages. Many other Linux distributions use the same method in building packages for their particular distribution, however the order by which these sections are structured may be different. Therefore if you wish to build or update new RPM packages for compatibility with the OpenNA Linux distribution, you should adhere to these guidelines as closely as possible and respect any tabs and/or space characters OpenNA puts and uses in their respective SPEC files.

To understand the OpenNA structure of a SPEC file, the following provides a detailed look at each of the sections briefly mentioned above along with a detailed explanation of the data elements contained within each section.

The Header section:

The **Header** section is used to define all information required by RPM to build, store, manage, and list the future package. Most information contained in this section will be used to display the RPM package information when it is queried by the RPM command "**`rpm -qi softwarename`**" on your system. Some of this information is used by the RPM database to store and manage updates, removal, etc. Other information is necessary to get the name of the software source being built and also where and how other information is used to integrate external patches and files.

```
Summary:        Small description of the software
Name:           softwarename
Version:        1.0
Release:        1
License:        GPL
```

Summary: This line is used to put a small description of the software. For example, if Exim is the software you want to build as RPM package, then you can use a small description for Exim as follows: A Mail Transfer Agent (MTA).

Name: This line is used to define the name of the package we are going to compile. For example if you are going to download the Exim source code (.tar.gz) and compile it into a RPM package for use, then exim which is the name of the software will be the name to use here. This will be used later in package's name and package database on the user's machine too.

Version: This line is used to define the version number of the software we want to compile as RPM package. For example, if we have downloaded Eject and the Eject version is 4.24, then this will be the version number to define and use here.

Release: This line is used to define the release number for the package we want to build. It's always a good practice to start from 1 and increment the number every time we provide a new RPM version of the same software. For example if this is the first time that we compile Exim to provide RPM package, then we can put 1 here for Exim version 4.24 and increment it to 2 if we provide a new RPM package for Exim version 4.24. Now what happens if the Exim version changes from version 4.24 to 4.30. In this case we can start again from 1 since it is a new version number different from the previous version.

License: This line is used to define the type of license associated with the software you want to build. Please take special attention to copyright and license, check inside the source code to get the correct license associated with the software and put it here.

```
Group:          SERVER/System Environment/Base
Source0:        %{name}-%{version}.tar.gz
URL:            http://www.domain.org/
```

Group: This line is used to define in which part of the general package tree to place this package. With OpenNA Linux we have two main distinctive group which are GUI and SERVER, GUI relates to every software running under GUI environment and SERVER to every software running under SERVER environment. After that, we use subgroup like "System Environment" and "Base". It is mandatory to follow it, otherwise your package will mess with the other ones, in the package tree selection of OpenNA Linux installer, or in package manager front-ends.

Source0: This line is used to define what source file to use for building the package. To resume, it is the name of the source code to use and compile but as you can see it is defined and listed as macro "`%{name}-%{version}.tar.gz`" which mean according to what we have defined previously under the "Name:" and "Version:" tags as softwarename and 1.0. Therefore translating "`%{name}-%{version}.tar.gz`" will become "`softwarename-1.0.tar.gz`" which is the source code we want to compile here.

URL: This line is used to define the home page of the program as people will know where to find new sources should they take a liking to upgrading the source and do a recompilation.

```
Distribution:           OpenNA Linux
Vendor:                 OpenNA, Inc.
Packager:               Open Network Architecture Inc.
```

Distribution: This line is used to define the name of the distribution under which the RPM package applies. It is an optional tag that you can use into your SPEC file if you want.

Vendor: This line is used to define the vendor name. In our example, it is set to OpenNA, Inc. but you are free to put what you want here or to not use it at all. It is an optional tag that you can use into your SPEC file if you want.

Packager: This line is used to define the name of the organization that make the package or the name of the person that make it. It is an optional tag that you can use into your SPEC file if you want.

```
ExclusiveArch:          i686
ExclusiveOS:            Linux
BuildRoot:              %{_tmppath}/%{name}-%{version}-root
```

ExclusiveArch: This line is used to define the CPU architecture under which the package will been compiled to run on. In other term, it can only build on the specific arch specified. It is also used for the extension of the RPM package (`.i686.rpm`). Please note that if you put i686 here, then the build RPM package will only install on a i686 and above CPU architecture. This is what with use with OpenNA Linux.

ExclusiveOS: This line is used to define the OS under which the RPM package has been made to run on. This excludes a piece of software from a particular operating system. In our example, we build RPM for Linux only.

BuildRoot: This line is used to define the temporally path (directory) under which all compiled binaries, libraries, files, etc will be keep before being packaged into RPM once everything was properly build. You should not have to change the default path which is to put everything under "`/var/tmp/softwarename-version-root/`".

```
BuildRequires:          db4-devel, openssl-devel
Requires:               openssl, db4
```

BuildRequires: This line is used to define the list of required RPM packages that should already be installed on your system for the package to properly build. In other words, this is where we list all development packages (x-devel packages) our software may need to properly compile and build. This also helps a developer to know what packages are required to already be installed on the system for the package to compile.

Requires: This line is used to define the list of required RPM packages that should be installed on your system for the software to run. In other words, this is where you list here all RPM packages from which your compiled software depend on to run once installed into the OS. Be advised that RPM will compute the dependencies on dynamic libs automatically. In our example, OpenSSL and db4 RPM packages should already be installed into the OS before installing this new RPM or you will get dependencies problem.

```
Conflicts:              sendmail, postfix, qmail
Provides:               smtpdaemon
```

Conflicts: This line is used to define names of similar software packages doing and providing the same service which conflict with the software we want to build as RPM package here. In other words, what other packages does this conflict with. In our example, Exim provides mail service like Sendmail, Postfix, and Qmail and to avoid possible problem of software doing the same thing to be installed on the same server, we define here the name of all software making the same job as Exim does.

In this way, if during install of the new RPM package, any of the name services listed in the "Conflics" tag is already available on the system, then RPM will remove it before installing the new RPM for Exim. It is rare that you have to use this tag into your SPEC file.

Provides: This line is used to define what service is provided by the RPM you are going to build. For example, Apache provides a webserver server, by the same token vsFTPd provides ftpserver capabilies. In our example, we are going to provide a RPM package for Exim, therefore here we define "smtpdaemon" as the service to inform the system that this RPM will provides an smtp service. It is rare that you have to use this tag in your SPEC file and the only time that you could potentially use it would be when the software you are going to build as RPM package is made to provide service on Unix.

```
Source1:                software-pamd
Patch0:                 OpenNA-patch0
```

Source1: This line is used to define external files you want to include in the build. The external files defined here should exist and be located under the "`/usr/src/openna/SOURCE/`" directory for the SPEC file to find and use it. For example, you may have a PAM (**P**lugable **A**uthentication **M**odule) file to include with your software. This PAM file is called "software-pamd" in our example and handle PAM parameters for the software we want to build here.

Therefore, we include the file into the SPEC file with the "Source1:" tag in order for the build to add it into the package. If you have more than one file to include, then just define them as "Source2:", "Source3:", etc. Again, it could be rare that you have to use this tag into your SPEC file and the only reason you would use it would be when you have multiple external files to add to your build.

Patch0: This line is used to define patches you want to include in the build. This tag is used often when we patch the source code of the original software, however only when we have to patch the source code. This tag is particularly important when you find bugs or whenever you have changed something into the original source code, you have to produce a patch of your changes for other people to know what you have made/changed in the code. This is where we list and define them.

If you have more than one patch to include into the build, then just define them as "Patch1:", "Patch2:", etc. Remember that you have to use this tag only if you have made changes to the original source code of the software and have provided your change through a patch file. With OpenNA Linux, we identify patches as "OpenNA-patch0", "OpenNA-patch1", etc.

```
%description
This is a description....
```

%description: This is a special tag used inside the header section of the SPEC file to give a more complete description of the software being installed in order to help the user to decide whether he wants to install the package or not.

Take a note that sometime, you have to include more than one "%description" tag into your SPEC file and this could happen when you have to separate the build package into two or more packages like "`name-devel.i686.rpm`" and/or "`name-tools.i686.rpm`", etc.

The Prep section:

The **Prep** section is used by RPM to create the top-level build directory (`/usr/src/openna/BUILD`), unpack the original sources into the build directory, and apply optional patches (if available) to the source. It may be then followed by any command wanted by the packager to get the sources into a ready-to-build state.

```
%setup -q
%patch0 -p1
```

%setup -q: This is a built-in script macro used in the "%prep" section to cd (change directory) into the build tree, extract the source(s), change ownership and permissions information of source files. This macro should always be present into the SPEC file for RPM to build.

%patch0 -p1: This is a built-in script macro responsible for applying the patch to the source; its parameter "`-p`" is passed to the patch program. Imagine if you had another patch declared Patch1: .. in the header section, you would add another line: %patch1 -px here to use it as another patch for the software. This macro should be present only if you have some patches to apply to the original source code of the software you want to build with RPM.

This macro is directly related to the one available in the Header section of the SPEC file "Patch0:". Therefore, if nothing is defined in the Header section for the patch, you cannot define and use it here. In addition, each defined patch in the Header section should have its own corresponding macro patch defined here (in the Prep section).

The Build section:

The Build section contains the built-in script macro responsible for the actual build of the software. It consists of the commands being issued when building a package from an un-tarred source tree. What we often see here is the "`%configure`" and "`%make`" macros.

```
%configure
%make
```

%configure: This built-in script macro is used for configuring the source. It works in the same way as when you issue a simple "`./configure`" command, however some differences exist. "`%configure`" also issues a "`./configure`" with many add-ons such as export `CFLAGS="$RPM_OPT_FLAGS"` before the configure, and options such as i686-openna-linux --`prefix=/usr --datadir=/usr/share` etc. Therefore when using this macro in your SPEC file you don't have to add any `--prefix=/usr`, `--bindir=/usr/bin`, etc lines because it will automatically do it for you.

What you still can add are option like `--enable-something`, etc if required or available depending on the source code your are going to configure. Sometimes these arguments are not supported by the configure script. In such case, you would have to discover the reason, and issue the "`./configure`" with the appropriate parameters. This macro should be present only if the source code you are trying to configure supports the "`./configure`" command. Some software only need "`make`" to build and don't provide any "`configure`" scripts in their source code.

%make: This built-in script macro basically performs a `make` with the appropriate multiprocessor parameter `-j num` (if available). It works in the same way as when you issue a simple '`make`" command to build a source code. In many (but not all) cases a simple make will do here. This macro should be present only if the source code you are trying to build support the '`make`" command (again most support it).

The Install section:

The **Install** section contain the scripts responsible for the actual installation of the package into the simulation installation directory.: `$RPM_BUILD_ROOT`. This section will contain all commands necessary to have the software ready to run on the user's system. Usually a '`%makeinstall`" built-in script macro command will be enough for the compiled software to properly be packaged and installed but many other commands can be added to this section. Use of this script macro will depend of how the original source code is made and which part of the installation should be made manually (if necessary). Again, this will highly depend on the way the original source code install.

```
%makeinstall
mkdir -p $RPM_BUILD_ROOT%{_sysconfdir}/pam.d
install %{SOURCE1} $RPM_BUILD_ROOT%{_sysconfdir}/pam.d/mysoftware
```

%makeinstall: This built-in script macro installs the software into the simulation installation directory. It works in the same way as when you issue a simple '`make install`" command to install a source code. In some cases the configure script in the original source code of the software is partially broken or incomplete and you may need to lurk in the Makefiles to guess the additional parameters to have it install correctly.

This is where you can place into some SPEC file, additional command in this section. One of the most common ones is when you have to use "`make DESTDIR=$RPM_BUILD_ROOT install`" instead of just "`%makeinstall`". This macro should always be present in the SPEC file for RPM to build.

mkdir -p $RPM_BUILD_ROOT%{_sysconfdir}/pam.d: This line is used to create a directory called "`/etc/pam.d/`" under the simulation installation directory "`$RPM_BUILD_ROOT`" to handle the "`mysoftware`" file which is going to be installed (as outlined in the next paragraph). We manually create it here because the software source code doesn't do it automatically for us.

install %{SOURCE1} $RPM_BUILD_ROOT%{_sysconfdir}/pam.d/mysoftware: This line is used to manually install the external file called "`software-pamd`" as previously defined into the Header section of the SPEC file under the "Source1:" heading. Since "`software-pamd`" is an external file (added by us into the build) which doesn't come from the original source code of the software, the rpm build doesn't know what to do with it and we have to manually inform it about what we want it to do with this file.

Here we instruct rpm to install the file "`%{SOURCE1}`" into the simulation installation directory "`$RPM_BUILD_ROOT`" under the "`%{_sysconfdir}/pam.d`" subdirectory and call it "`mysoftware`".

The Clean section:

The **Clean** section of the SPEC file contains the script responsible for cleaning the build directory tree, `$RPM_BUILD_ROOT`, which is used to handle the simulation installation. Additionally, this section also provides optional `%pre`, `%post`, `%preun`, and `%postun` macros that are used and required only when you have library files and/or user accounts to automatically add and configure onto the system. These predefined macros allow the package builder to write a piece of code which will execute on the client machine during install or un-install of a package.

```
%clean
%post
%pre
%preun
%postun
```

%clean: This built-in script macro is meant to clean the build directory tree, `$RPM_BUILD_ROOT` before packaging the RPM. The command "`rm -rf $RPM_BUILD_ROOT`" always appears after the "`%clean`" macro and it is the one responsible for removal of the `$RPM_BUILD_ROOT` directory.

If you want to keep the simulation installation directory "`$RPM_BUILD_ROOT`", then you can add a comment "#" before the line and the `$RPM_BUILD_ROOT` directory will not be removed before packaging the RPM. In this way, you can check inside this directory for installed files, binaries, and library locations which are required under the Files section of the SPEC file that list all the files that you want to include and package in the RPM.

%pre: This built-in script macro is executed just before the package is installed on the system. It is often used when you build software providing services on the system that need user account to be created. This is true for software like Apache, vsFTPd, BIND, Samba, Exim, SQL, etc. In this case, we will use the "`%prep`" macro to create the user account.

Here is an example for Apache, you can see that we first create a group account and then the user account with the appropriated number and name. Remember that this is just an example, and depending on the service you want to build as RPM package, the group number ID and user account name will change.

```
/usr/sbin/groupadd -g 48 www > /dev/null 2>&1 || :
/usr/sbin/useradd -c "Web Server" -d /home/httpd -g 48 \
      -s /sbin/nologin -u 48 www > /dev/null 2>&1 || :
```

%post: This built-in script macro is executed just after the package is installed on the system. It can appears into different sentence depending of if the package you are building contains libraries or initialization script files.

Sometime we use it to automatically update libraries path like "`/usr/lib/`", "`/usr/X11R6/lib/`", etc inside the "`/etc/ld.so.conf`" file. The command used to do it is "`%post -p /sbin/ldconfig`". This is just a example for the use of this macro and you have to note that many other command can be used and this will depend of what the developer want to do with the software to package.

%preun: This built-in script macro is executed just before the package is uninstalled from the system. It is the opposite of the "`%prep`" macro and we (often) use it to stop running service and to remove initialization files.

Here is an example for Apache, firstly we stop the running process, then uninstall the initialization script file. Again, this is a example and depending of what you want to build with RPM, this will change.

```
if [ $1 = 0 ]; then
  /sbin/service httpd stop > /dev/null 2>&1 || :
  /sbin/chkconfig --del httpd > /dev/null 2>&1 || :
fi
```

%postun: This built-in script macro is executed just after the package is uninstalled on the system. It is the opposite of the "`%post`" macro and we (often) use it to remove group and user account from the system or libraries path.

Here is an example for Apache, firstly we update the "`/etc/ld.so.conf`" file, then remove the user account and the group ID. Again, this is a example and depending of what you want to build with RPM, this will change.

```
/sbin/ldconfig
if [ $1 = 0 ]; then
  /usr/sbin/userdel www > /dev/null 2>&1 || :
  /usr/sbin/groupdel www > /dev/null 2>&1 || :
fi
```

The scope of such scripts may be very large. One have to remember that these scripts will be run as root... They correspond to the tasks a system administrator would have to accomplish when installing a new program on a system.

The Files section:

The **Files** section consists of a list of all files created by RPM in the build directory tree "`/var/tmp/softwarename/`". The file list must be written by hand in the SPEC file to be packed into the package. Usually, files section start as follow:

```
%files
%defattr(-,root,root)
```

Here the "`%defattr(-,root,root)`" buit-in script macro (which should always appears in the SPEC file) defines the attributes to be applied to each file being copied to the user's system. The three arguments given means:

`-` : all the attributes for regular files are remained unchanged,
`root` : the owner of the file is root,
`root` : the group of the file is root.

Other buit-in script macros referring to pre defined path are also used like `%{_mandir}`, `%{_bindir}`, `%{_sbindir}`, etc. You have buit-in script macros for every kind of path you need and all pre defined buit-in script macros for path are listed inside the "`/usr/lib/rpm/macros`" file. Look inside this file to get a list of all available buit-in script macros for pre defined path.

Some examples:
If you want to list all binaries that will install under "`/usr/bin/`" directory, then use the following macro: `%attr(0511,root,root) %{_sbindir}/*`

Here the macro "`%attr(0511,root,root)`" mean that all files inside "`/usr/sbin/`" will have permission set to 0511 owned by the root user and group set to the root user. The "`%{_sbindir}/*`" macro is the path for "`/usr/sbin/`" and "`*`" means to list and pack all files available under this directory.

If you want to add to the list a configuration file available under the "`/etc/`" directory, then use something like the following macro: `%attr(0644,root,root) %config(noreplace) %{_sysconfdir}/myconfig.conf`

Here the macro "`%attr(0644,root,root)`" means that file called "`myconfig.conf`" will have permission set to 644 owned by the root user and group set to the root user.

The "`%config(noreplace)`" macro means that if previous installed file into the same directory exist, then no dot replace it with the new one coming from the new RPM package we are going to build. The "`%{_sysconfdir}/myconfig.conf`" macro means that we want to add in the list the file called "`myconfig.conf`" and located under the "`/etc/`" directory (`%{_sysconfdir}`).

The Changelog section:

The **Changelog** section is used to keep track of different changes made to the package. Every new release build of the package must correspond to a paragraph in this section. The structure of this section have to be respected as following:

```
* Mon Dec 01 2003 Gerhard Mourani gmourani@openna.com
- Initial build for mysoftware.
```

The first line of the paragraph begins with * and separated by a space; three letters for the day of the week; three letters for the month; two figures for the day of the month; four figures for the year; First name of the packager; Last name of the packager; e-mail of the packager between bracket, then follow one line per modification applied to the package beginning by a `-`.

The build

Our SPEC file is finally complete. Take a long breathe, sit down and type: **`rpm -ba mypackage.spec`** to build the RPM package. There are then two possibilities for the last line of your process:

exit 1: There is a problem in your build.
exit 0: Everything is ok and the rpm successfully build.

Testing your RPM

When the build process of your rpm has been successfully completed, you next step will be to perform some test to verify if everything is correct. The proper way to do it will be to install the rpm in question into a machine different from the compilation one if possible and check for the following:

1. Is the rpm installs in the corresponding directories with the correct name? If yes, then is the rpm completely uninstalls from the system without letting some empty directory or file name?

2. Are all the expected files created at their expected place with the expected rights and owners?

3. Are all the installation modifications (if any) effective?

4. Is the rpm software is functional and run fine?

5. Try various different installs and uninstalls to check whether all expected features are well implemented, for example without required packages.

Send your work

Just upload your rpm and source rpm (package.i686.rpm and package.src.rpm) to `ftp://ftp.openna.org/incoming/` Then send an e-mail to Mathieu Masseboeuf and/or Dougal Ballantyne (Testers) in order to warn them (www.openna.org).

Building RPM from source code

In this part of the documentation, I'm going to explain you how to build a RPM package from source code. I will guide you from the beginning where we will download the source code to the end where the corresponding code will be build and available as RPM package to install on different computers.

This is just an example and everything as explained for the software code we are going to download and build also apply to any other source code.

In our example, we will download, compile, and build source code for Eject as RPM package. Eject is a small Linux program that ejects removable media.

Downloading the source code

Your first step, will be to point your browser to the Eject website and download a copy of the latest version of Eject (2.0.13). The site URL is : `http://members.rogers.com/jefftranter/`. Download the latest copy of the software (eject-2.0.13.tar.gz) and transfer it into your OpenNA Linux Workstation. The above is true for any software source code your want to build as RPM package. You have to download and transfer it into your OpenNA Linux Workstation.

Putting the source code in the right directory

Once Eject (or any source code you want to build as RPM package) has been transfered on your OpenNA Linux Workstation, you have to move it into the RPM build directory before starting to work on it. Remember that any actions for building RPM packages should be made under the "`/usr/src/openna/`" directory and any source code (software-version.tar.gz or .bz2) should be placed into the "`/usr/src/openna/SOURCE/`" directory for RPM to find and build it.

- To put the code source into the SOURCE directory, use the command:
  `[root@deep /tmp]# cp eject-2.0.13.tar.gz /usr/src/openna/SOURCES/`

Checking the source code

Now, we have to move into the SOURCE directory and decompress (unzip) the source code, then move inside the code and read the README file accompanying the software to get important information (if available) about how to properly compile this software. We also have to issue a "`./configure --help`" command to see if personalized or specific options are available with the software before creating the SPEC file to build it as RPM package.

- To move into the SOURCE directory, use the command:
  `[root@deep /tmp]# cd /usr/src/openna/SOURCES/`

- To decompress the source code, use the command:
  `[root@deep SOURCES]# tar xzpf eject-2.0.13.tar.gz`

- To move into the source code, use the command:
  `[root@deep SOURCES]# cd eject-2.0.13/`

Most source code today provide a README file that should be the first file to read every time you want to build source code software.

- To read the REAME file, use the command:
  [root@deep eject-2.0.13]# less README

From the REAME file accompanying this software, we can see that we satisfy any requirement for the software to properly build. Nothing else (like an additional software to be installed) is require. We have everything and can continue our work.

Before starting to write a SPEC file for the software, you have to run a "`./configure --help`" command into its source code directory to get additional information about specific options that could be available with this program. This will allow us to how if we will have to add some special option into its SPEC file or not.

- Run the configure command to see what options are available:
  [root@deep eject-2.0.13]# ./configure --help

From the above running command, you can see that everything is standard (there is no specific options) with this software and for this reason we don't have to include any additional options into its SPEC file.

Building the SPEC file

Ok, the Eject source code is available under the SOURCE directory and nothing special is required to build it (it should build without any problem with commands like `./configure`, `make` and `make install` as many other software compile. Therefore, we can start to create the appropriated SPEC file for this software.

To do so, we have to move into the "`/usr/src/openna/SPECS/`" directory before creating the file (SPEC) because this directory is where RPM look for available SPEC file and where we will issue the RPM build command to build the RPM package.

- To move into the SPECS directory, use the command:
  [root@deep eject-2.0.13]# cd /usr/src/openna/SPECS/

Now we have to create the SPEC file for the software. Any SPEC files are created based on the initial name of the software you want to build as RPM package followed by the extension ".spec". This means that for Eject, we have to create a file called "`eject.spec`" (name + .spec).

- To create the Eject SPEC file, use the command:
  [root@vslc SPECS]# touch eject.spec

Once the above file is created, we can edit it and start to fill it with all the required information and tags for RPM to build. If you need more information about what should be filled inside this empty file and which tags to use, then refer to the previous section of this article.

Here we start from an empty file and we have to cup and past any required tags from the example SPEC file (as shown as the beginning of this article) and change information as requested. We start from the header section through the latest section (Changelog).

The header section:

```
Summary:
Name:
Version:
Release:
License:
```

Related to the software we want to build here, we will logically fill the above information tags as follow:

```
Summary:        A program that ejects removable media
Name:           eject
Version:        2.0.13
Release:        1
License:        GPL
```

The "**Summary**" tag can be filled with anything that you want, in my example I give a brief and clear summary for the software. The "**Name**" tag should absolutely be the name of the source code software. The "**Version**" tag should also be the version of the source code. The '**Release**" tag can be what you want and it is a good idea to start with 1 for the first RPM build of the software. The "**License**" tag relates to the original license of the software, usually GPL but verify inside the source code to get the correct license information.

```
Group:
Source0:
URL:
```

Related to the software we want to build here, we will logically fill the above information tags as follow:

```
Group:          SERVER/System Environment/Base
Source0:        %{name}-%{version}.tar.gz
URL:            http://members.rogers.com/jefftranter/eject.html
```

The "**Group**" tag here is defined as "SERVER/System Environment/Base", "SERVER" because this software is made to run under SERVER, "System Environment" because it is a system software, and "Base" because it relate to a base install. The "Source0" tag is defined as macro meaning that the real name will be "`eject-2.0.13.tar.gz`" (the source code archive name located under the SOURCE directory). The "URL" tag defines the location (website) of the software.

```
Distribution:
Vendor:
Packager:

ExclusiveArch:
ExclusiveOS:
BuildRoot:
```

Related to the software we want to build here, we will logically fill the above information tags as follow:

```
Distribution:     OpenNA Linux
Vendor:           OpenNA, Inc.
Packager:         Open Network Architecture Inc. <noc@openna.com>

ExclusiveArch:    i686
ExclusiveOS:      Linux
BuildRoot:        %{_tmppath}/%{name}-%{version}-root
```

The "**Distribution**" tag is used to define the distribution from which this RPM package is going to be compiled, in our case "OpenNA Linux" and you should not have to change it as long as you are building RPM packages for OpenNA Linux. The "Vendor" tag can be replaced by what you want, here I use OpenNA, Inc. because I'm working for OpenNA, Inc. The "Packager" tag can be your name and email address.

The "**ExclusiveArch**", "**ExclusiveOS**", and "**BuildRoot**" tags should never be changed, see at the beginning of this article for more information about each one.

```
%description
```

Related to the software we want to build here, we will logically fill the above information tag as follow:

```
%description
Eject allows removable media (typically a CD-ROM, floppy disk, tape, or
JAZ or ZIP disk) to be ejected under software control. The command can
also control some multi-disc CD-ROM changers, the auto-eject feature
supported by some devices, and more.
```

From here you can see that most of all informations as shown above come from README, INSTALL files of the software source code. From these files (README, INSTALL), I know the description of the software, the name of the software, the version number, summary, License, and URL.

The Prep section:

```
%prep
%setup -q
```

Here there is nothing special, I use the `%prep` and `%setup -q` macros for RPM to automatically move into the source code of the software before starting to compile it as defined into the next section of the SPEC file.

The Build section:

```
%build
CFLAGS="$RPM_OPT_FLAGS -fomit-frame-pointer"
%configure
%{__make}
```

Remember, the Build section is where RPM start to configure and compile the software. From the above tags, I've defined a flag into the SPEC file for RPM to add it into the default flag to be used during the compile process. Here the "`CFLAGS="$RPM_OPT_FLAGS -fomit-frame-pointer`" line means that RPM have to get optimization flags information as defined into the "`/usr/lib/rpm/i686-linux/macros`" file and add to this information, the "`-fomit-frame-pointer`" option.

Someone may say, Why you do it in this way? Because it is not all software that need or compile with this option (`-fomit-frame-pointer`), some need it where other will fail to compile if it is defined. Therefore adding this option into the SPEC file give me the freedom to define general optimization flags into the "`/usr/lib/rpm/i686-linux/macros`" file and keep special one for specific software. In this example I've used the "`-fomit-frame-pointer`" flag but you can replace it with any other specific flags that you may have to add to your SPEC file.

The "**%configure**" and "**%{__make}**" macros as defined above will be used by RPM to configure and build the software. Here I don't define any additional options with "`configure`" because we have see during the "`./configure --help`" command used previously that nothing special is available (with this software) to be added here. This program configures and installs in the standard way as many other do.

The Install section:

```
%install
rm -rf $RPM_BUILD_ROOT
%makeinstall
```

As usually, I'm firstly use the "`rm -rf $RPM_BUILD_ROOT`" macro command to remove any previous "`RPM_BUILD_ROOT`" directory that could exist (it is always a good idea to add this command into the Install section of your SPEC file), then issue the "`%makeinstall`" macro command to install the compiled files into the simulation directory before RPM packs them to produce a RPM package.

The Clean section:

```
%clean
rm -rf $RPM_BUILD_ROOT
```

Here we just use the "`rm -rf $RPM_BUILD_ROOT`" command to inform the builder to remove the simulation directory (where all compiled files reside) once RPM successfully build. At this time of our reading, we will add a "#" at the beginning of the command line to avoid RPM to run it (`#rm -rf $RPM_BUILD_ROOT`) and remove the simulation directory.

We do this because it is the first time that we compile this software and we don't know what files, binaries, libraries, headers, etc will be installed and where. Therefore if you add a "#" sign at the beginning of the "`rm -rf $RPM_BUILD_ROOT`" command, RPM will not remove the simulation directory even if the software compiles and builds successfully.

In this way, we can move into the simulation directory (`/var/tmp/software-version-root`) and see what will be installed with this software and where. The information will be very important to fill the next section (The Files section) of our SPEC file.

The Files section:

```
%files
%defattr(-,root,root)
```

As you are supposed to know now, the Files section of the SPEC file is where we manually define and list all files that will be installed (related to the software we are going to build). RPM doesn't have a way to know this information and this is why we have to manually list them here. Since we have previously commented out the command line to remove the simulation directory (`rm -rf $RPM_BUILD_ROOT`) under the Clean section, we can get the list of files that should be installed.

To do so, we have to move into the simulation directory of the software and fill this section (The Files section) according to what is available into this directory (the simulation directory). But at this time, we will use the above macros for the Files section.

The Changelog section:

```
%changelog
* Tue Dec 09 2003 Gerhard Mourani <gmourani@openna.com>
- Initial build for Eject.
```

This section is the last one of our SPEC file. We use it to list any changes that are made into the software. Since this is the first time that we build this program as RPM package, we can use the above information.

Reviewing your SPEC file:

Ok, before going to build the RPM package, we will review the SPEC file to be sure that everything is correct. Here is how the file should look after filling all the information as described above. Remember, the SPEC file is not complete but this is enough for starting the build to get additional information in order to complete it.

```
Summary:                A program that ejects removable media
Name:                   eject
Version:                2.0.13
Release:                1
License:                GPL

Group:                  SERVER/System Environment/Base
Source0:                %{name}-%{version}.tar.gz
URL:                    http://members.rogers.com/jefftranter/

Distribution:           OpenNA Linux
Vendor:                 OpenNA, Inc.
Packager:               Open Network Architecture Inc. <noc@openna.com>

ExclusiveArch:          i686
ExclusiveOS:            Linux
BuildRoot:              %{_tmppath}/%{name}-%{version}-root

%description
```

```
Eject allows removable media (typically a CD-ROM, floppy disk, tape, or
JAZ or ZIP disk) to be ejected under software control. The command can
also control some multi-disc CD-ROM changers, the auto-eject feature
supported by some devices, and more.


%prep
%setup -q


%build
CFLAGS="$RPM_OPT_FLAGS -fomit-frame-pointer"
%configure
%{__make}


%install
rm -rf $RPM_BUILD_ROOT
%makeinstall


%clean
#rm -rf $RPM_BUILD_ROOT


%files
%defattr(-,root,root)


%changelog
* Tue Dec 09 2003 Gerhard Mourani <gmourani@openna.com>
- Initial build for Eject.
```

What is missing into our SPEC file as shown above? Of course, the list of installed files (under the Files section), any external patches that we may need (under the Header and Prep sections) and any possible additional commands like "`strip`" to strip binaries, or "`rm`" to remove unneeded files, etc (under the Install and clean sections).

Build it:

Now it's time to build and get some result, errors, etc... The command to build a RPM package is "`rpmbuild -ba software.spec`". Based on our example software, the command will be as follow.

- To build RPM for Eject, use the command:
  `[root@deep SPECS]# rpmbuild -ba eject.spec`

The above command will starts the build process for the software and depending of the complexity and size of the program, as well as the speed of your processor, this will take few seconds, minutes or hours.

```
RPM build errors:
    Installed (but unpackaged) file(s) found:
   /usr/bin/eject
   /usr/bin/volname
   /usr/share/man/man1/eject.1.gz
   /usr/share/man/man1/volname.1.gz
```

Once your RPM finished to build, you will get the above error. This is normal because we have not defined the list of installed files under the File section of our SPEC file. Now we know what to add into the Files section and if you want to get a better perceptive on what should be added, then just move into the simulation directory of the software (`cd /var/tmp/eject-2.0.13-root/`) and see how the tree structure is made.

According to the new information, we can rewrite our SPEC file and add the missing parts as follow. By the way, we can strip the resulting installed binaries by adding a strip command into the Install section.

```
Summary:                A program that ejects removable media
Name:                   eject
Version:                2.0.13
Release:                1
License:                GPL

Group:                  SERVER/System Environment/Base
Source0:                %{name}-%{version}.tar.gz
URL:                    http://members.rogers.com/jefftranter/

Distribution:           OpenNA Linux
Vendor:                 OpenNA, Inc.
Packager:               Open Network Architecture Inc. <noc@openna.com>

ExclusiveArch:          i686
ExclusiveOS:            Linux
BuildRoot:              %{_tmppath}/%{name}-%{version}-root

%description
Eject allows removable media (typically a CD-ROM, floppy disk, tape, or
JAZ or ZIP disk) to be ejected under software control. The command can
also control some multi-disc CD-ROM changers, the auto-eject feature
supported by some devices, and more.


%prep
%setup -q


%build
CFLAGS="$RPM_OPT_FLAGS -fomit-frame-pointer"
%configure
%{__make}


%install
rm -rf $RPM_BUILD_ROOT
%makeinstall

strip $RPM_BUILD_ROOT%{_bindir}/* || :


%clean
rm -rf $RPM_BUILD_ROOT


%files
%defattr(-,root,root)
%attr(0511,root,root) %{_bindir}/*
%attr(0440,root,man)  %{_mandir}/*/*
```

```
%changelog
* Tue Dec 09 2003 Gerhard Mourani <gmourani@openna.com>
- Initial build for Eject.
```

So, what have been changed into this SPEC file related to our previous one? We can see that we have added the missing list files into the Files section, removed the "#" comment from the "`rm -rf $RPM_BUILD_ROOT`" command line into the Clean section (because we no longer need it now), and added a new line into the Install section to strip all installed binaries.

Lets build the package again to see what will happen now.

- To build RPM for Eject, use the command:
  `[root@deep SPECS]# rpmbuild -ba eject.spec`

The above command will starts the build process again and will (this time) success to build the package. We can see this with the following lines at the end of the build process.

```
+ umask 022
+ cd /usr/src/openna/BUILD
+ cd eject-2.0.13
+ rm -rf /var/tmp/eject-2.0.13-root
+ exit 0
```

Here we know that the build successfully complete with the exit code 0. Remember that code "`exit 0`" means everything was successfully made where code "`exit 1`" means that something failed during the build time and you have to reedit your SPEC file and fix the problem.

Congratulation! Your first RPM package is built and you can distribute and install it. RPM packages are stored under "`/usr/src/openna/RPMS/i686/`" directory where SRPMS packages are stored under "`/usr/src/openna/SRPMS/`" directory. You can remove everything under the "`/usr/src/openna/SOURSES/`" and "`/usr/src/openna/BUILD/`" directories.

Adding patches to the SPEC file:

From version to version, sometime you have to add patches to the original code source of the software to fix bug, security, etc. The way to add patch into the SPEC file is as follow.

Name your patch as OpenNA-patch0 for the first patch, OpenNA-patch1 for the second, etc and put them into your SOURCES directory, then edit the corresponding SPEC file and add the following lines for each patch that you have.

Under the Header section:
```
Patch0:          OpenNA-patch0
```

Under the Prep section:
```
%patch0 -p1
```

That all you have to do, RPM will automatically find the patch and will apply it to the source code during the build process. The name of the patch can be what you want but with OpenNA Linux, we like to have them defined as OpenNA-patchx.

If you don't know how to create patches, then here is an example to help you quickly understand the way to do it. To create a patch, we use the Linux “`diff`” command under the source code of the software we want to patch. This means that we have to make a copy of the original source code before using the “`diff`” command.

In the following example, I suppose that we want to create a patch for the Eject software available under the SOURCES directory. Therefore we move under the SOURCES directory, decompress the source code, make a full copy of the original source code, modify codes that we want to patch, then use the “`diff`” command to produce the patch.

- To create a patch, use the commands:

```
[root@deep /]# cd /usr/src/openna/SOURCES/
[root@deep SOURCES]# tar xjpf eject-2.0.13.tar.gz
[root@deep SOURCES]# cp -a eject-2.0.13 eject-2.0.13.orig
[root@deep SOURCES]# cd eject-2.0.13
[root@deep eject-2.0.13]# vi eject.c
[root@deep eject-2.0.13]# cd ..
[root@deep SOURCES]# diff -ur eject-2.0.13.orig/ eject-2.0.13 > OpenNA-patch0
```

In the above example, I made a full copy of the source code (`eject-2.0.13`) and name it `eject-2.0.13.orig` then move inside the source code and modify what I have to modify then come back into the SOURCES directory (`cd ..`) to finally use the “`diff`” command to produce the patch.

Other useful resources related to RPM

There are a number of external resources available to help you with RPM, over and above the RPM man page, and this article.

RPM HOWTO by Donnie Barnes (`http://www.rpm.org/RPM-HOWTO/`)
Maximum RPM by Edward C. Bailey (`http://www.rpm.org/max-rpm/`)