

# Руководство по сборке RPM-пакетов для дистрибутивов АЛЬТ

Валентин Соколов, Мария Фоканова и другие.

# Оглавление

Вступление	1
PDF Версия	1
Структура документации	1
Вклад в руководство	2
Установка необходимых пакетов для процесса сборки	2
Введение в пакетные менеджеры	4
Основные команды RPM	4
Что такое RPM-пакет?	9
Подготовка к сборке RPM-пакетов	9
Рабочее пространство для сборки RPM-пакетов	10
Что такое SPEC-файл?	12
Пример .спес-файла	12
Пункты преамбулы	13
Составляющие основной части	14
RPM Макросы	17
Макросы путей системных каталогов	18
Макросы меню	18
Другие системные макросы	18
Прочие макросы	18
Пользовательские макросы	20
Инструмент Gear	21
Вступление	21
Структура репозитория	21
Правила экспорта	22
Основные типы устройства gear-репозитория	23
Быстрый старт Gear	23
Создание gear-репозитория путём импорта созданного ранее srpm-пакета	24
Создание gear-репозитория на основе готового git-репозитория	25
Сборка пакета из gear-репозитория	25
Фиксация изменений в репозитории	25
Hasher start	27
Что такое hasher?	27
Принцип действия	27
Настройка Hasher	27
Сборка в hasher	28
Сборочные зависимости	29
man hasher	29
Монтирование файловых систем внутри hasher	29

Примеры сборки пакетов с использованием инструментов Альт.....	31
Подготовка пространства .....	31
Написание sres файла и правил Gear .....	32
Описание пакета с исходными текстами на C++.....	35

# Вступление

Уважаемый читатель!

Мы хотим обратить Ваше внимание на то, что для успешного прохождения руководства необходимо внимательно читать текст и уделять внимание деталям. Каждый раздел содержит информацию, которая поможет Вам понять процесс сборки пакетов и улучшить свои навыки в этой области.

В тексте документации присутствуют ссылки на дополнительную информацию. Мы настоятельно рекомендуем Вам переходить по этим ссылкам и знакомиться с материалами более подробно. Это поможет Вам лучше понимать темы, раскрытые в руководстве, и получить полезную дополнительную информацию.

## PDF Версия

Вы также можете скачать [PDF версию данного документа](#).

## Структура документации

Перед тем, как приступить к сборке, нужно создать структуру каталогов, необходимую RPM, находящуюся в Вашем «домашнем» каталоге:

- Отображение файловой структуры будет представлено следующим образом:

```
$ tree ~/RPM/  
/home/user/RPM/  
|-- BUILD  
|-- BUILDROOT  
|-- RPMS  
|   |-- i586  
|   |-- x86_64  
|   `-- noarch  
|-- SOURCES  
|-- SPECS  
`-- SRPMS
```

- В дальнейшем вывод команд будет продемонстрирован следующим образом:

```
Name:      bello  
Version:  
Release:   alt1  
Summary:
```

- Темы, представляющие интерес, или словарные термины упоминаются либо как ссылки на соответствующую документацию или веб-сайт выделены **жирным** шрифтом, либо

*курсивом*. Первые упоминания некоторых терминов ссылаются на соответствующую документацию.

- Названия утилит, команд и других элементов, обычно встречающихся в коде, написаны **моноширинным** шрифтом.

#### ПРИМЕЧАНИЕ

Для сокращения команд, встречающихся в тексте, будет использоваться нотация:

- - команды без административных привилегий будут начинаться с символа “\$”
- - команды с административными привилегиями будут начинаться с символа “#”

#### ПРИМЕЧАНИЕ

По умолчанию `sudo` может быть отключено. Для получения административных привилегий используется команда `su`. Для включения `sudo` в стандартном режиме можно использовать команду:

```
# control sudowheel enabled
```

## Вклад в руководство

Вы можете внести свой вклад в это руководство, отправив запрос на принятие изменений (Pull Request) в [репозиторий](#).

## Установка необходимых пакетов для процесса сборки

Чтобы следовать данному руководству, Вам потребуется установить следующие пакеты:

#### ПРИМЕЧАНИЕ

Некоторые из этих пакетов устанавливаются по умолчанию в [Альт](#). Установка проводится с правами суперпользователя.

```
# apt-get update
```

```
Получено: 1 http://ftp.altlinux.org p10/branch/x86_64 release [4223B]
Получено: 2 http://ftp.altlinux.org p10/branch/x86_64-i586 release [1665B]
Получено: 3 http://ftp.altlinux.org p10/branch/noarch release [2844B]
Получено 8732B за 0s (81,8kB/s).
Найдено http://ftp.altlinux.org p10/branch/x86_64/classic pkglist
Найдено http://ftp.altlinux.org p10/branch/x86_64/classic release
Найдено http://ftp.altlinux.org p10/branch/x86_64/gostcrypto pkglist
Найдено http://ftp.altlinux.org p10/branch/x86_64/gostcrypto release
Найдено http://ftp.altlinux.org p10/branch/x86_64-i586/classic pkglist
Найдено http://ftp.altlinux.org p10/branch/x86_64-i586/classic release
Найдено http://ftp.altlinux.org p10/branch/noarch/classic pkglist
Найдено http://ftp.altlinux.org p10/branch/noarch/classic release
```

Чтение списков пакетов... Завершено

Построение дерева зависимостей... Завершено

```
# apt-get install gcc rpm-build rpmlint make python gear hasher patch rpmdevtools
```

# Введение в пакетные менеджеры

RPM — это семейство пакетных менеджеров, применяемых в различных дистрибутивах GNU/Linux, в том числе и в проекте [Сизиф](#) и в дистрибутивах [Альт](#). Практически каждый крупный проект, использующий RPM, имеет свою версию пакетного менеджера, отличающуюся от остальных.

**Различия между представителями семейства RPM выражаются в:**

- наборе макросов, используемых в .спес-файлах,
- различном поведении RPM при сборке «по умолчанию» — при отсутствии каких-либо указаний в .спес-файлах,
- формате строк зависимостей,
- мелких отличиях в семантике операций (например, в операциях сравнения версий пакетов),
- мелких отличиях в формате файлов.

Для пользователя различия чаще всего заключаются в невозможности поставить «неродной» пакет из-за проблем с зависимостями или из-за формата пакета.

**RPM в проекте Сизиф также не является исключением. Основные отличия RPM в Альт и Сизиф от RPM других крупных проектов заключаются в следующем:**

- обширный набор макросов для сборки различных типов пакетов,
- отличающееся поведение «по умолчанию» для уменьшения количества шаблонного кода в .спес-файлах,
- наличие механизмов для автоматического поиска межпакетных зависимостей,
- наличие так называемых set-version зависимостей (начиная с [4.0.4-alt98.46](#) ), обеспечивающих дополнительный контроль за изменением ABI библиотек,
- до [p8](#) и выпусков [8.x](#) включительно — очень древняя версия «базового» RPM (4.0.4), от которого началось развитие ветки RPM в Sisyphus (в Sisyphus и [p9](#) осуществлён частичный переход на [rpm 4.13](#)).

## Основные команды RPM

Для ознакомления с данным разделом потребуется пакет. В качестве примера мы будем использовать пакет [Yodl-docs](#).

**Как узнать информацию о RPM-пакете без установки?**

После скачивания пакета можно посмотреть данные о нём перед установкой. Для этого используется **-qip**, (Query | Install | Package) чтобы вывести информацию о пакете.

### ПРИМЕЧАНИЕ

ключ **-p** (-package) работает не с базой RPM-пакетов, а с конкретным пакетом. Например: чтобы получить информацию о файлах, находящихся в пакете, который не установлен в систему, используют ключи **-qpl**(Query | Package | List).

```
$ rpm -qip yodl-docs-4.03.00-alt2.noarch.rpm
```

Вывод:

```
Name       : yodl-docs
Epoch     : 1
Version    : 4.03.00
Release    : alt2
DistTag    : sisyphus+271589.100.1.2
Architecture: noarch
Install Date: (not installed)
Group      : Documentation
Size       : 3701571
License    : GPL
Signature  : DSA/SHA1, Чт 13 мая 2021 05:44:49, Key ID 95c584d5ae4ae412
Source RPM : yodl-4.03.00-alt2.src.rpm
Build Date : Чт 13 мая 2021 05:44:44
Build Host : darktemplar-sisyphus.hasher.altlinux.org
Relocations : (not relocatable)
Packager   : Aleksei Nikiforov <darktemplar@altlinux.org>
Vendor     : ALT Linux Team
URL        : https://gitlab.com/fbb-git/yodl
Summary    : Documentation for Yodl
Description :
Yodl is a package that implements a pre-document language and tools to
process it. The idea of Yodl is that you write up a document in a
pre-language, then use the tools (eg. yodl2html) to convert it to some
final document language. Current converters are for HTML, ms, man, LaTeX
SGML and texinfo, plus a poor-man's text converter. Main document types
are "article", "report", "book" and "manpage". The Yodl document
language is designed to be easy to use and extensible.

This package contains documentation for Yodl.
```

### Как установить RPM-пакет?

Для установки используется параметр **-ivh** (Install | Verbose | Hash).

#### ПРИМЕЧАНИЕ

Ключи **-v** и **-h** не влияют на установку, а служат для вывода наглядного процесса сборки в консоль. Ключ **-v** (verbose) выводит детальные значения. Ключ **-h** (hash) выводит **"#"** по мере установки пакета.

```
$ rpm -ivh yodl-docs-4.03.00-alt2.noarch.rpm
```

Вывод:



```
Подготовка...
##### [100%]
Обновление / установка...
1: yodl-docs-1:4.03.00-alt2
##### [100%]
Running /usr/lib/rpm/posttrans-filetriggers
```

### Проверка установки пакета в системе.

```
$ rpm -q () yodl-docs
```

Вывод:

```
yodl-docs-4.03.00-alt2.noarch
```

### Просмотр файлов пакета, установленного в системе.

```
$ rpm -ql yodl-docs
```

Вывод:

```
/usr/share/doc/yodl
/usr/share/doc/yodl-doc
/usr/share/doc/yodl-doc/AUTHORS.txt
/usr/share/doc/yodl-doc/CHANGES
/usr/share/doc/yodl-doc/changeLog
/usr/share/doc/yodl-doc/yodl.dvi
/usr/share/doc/yodl-doc/yodl.html
/usr/share/doc/yodl-doc/yodl.latex
/usr/share/doc/yodl-doc/yodl.pdf
/usr/share/doc/yodl-doc/yodl.ps
/usr/share/doc/yodl-doc/yodl.txt
/usr/share/doc/yodl-doc/yodl01.html
/usr/share/doc/yodl-doc/yodl02.html
/usr/share/doc/yodl-doc/yodl03.html
/usr/share/doc/yodl-doc/yodl04.html
/usr/share/doc/yodl-doc/yodl05.html
/usr/share/doc/yodl-doc/yodl06.html
/usr/share/doc/yodl/AUTHORS.txt
/usr/share/doc/yodl/CHANGES
/usr/share/doc/yodl/changeLog
```

### Просмотр недавно установленных пакетов.

```
rpm -qa --last|head
```

Вывод:

```
yodl-docs-4.03.00-alt2.noarch          Чт 22 дек 2022 18:09:10
source-highlight-3.1.9-alt1.git.904949c.x86_64 Вт 20 дек 2022 18:38:29
libsource-highlight-3.1.9-alt1.git.904949c.x86_64 Вт 20 дек 2022 18:38:29
gem-asciidoc-2.0.10-alt1.noarch        Вт 20 дек 2022 18:34:04
w3m-0.5.3-alt4.git20200502.x86_64     Вт 20 дек 2022 18:23:05
sgml-common-0.6.3-alt15.noarch        Вт 20 дек 2022 18:23:05
libmaa-1.4.7-alt4.x86_64              Вт 20 дек 2022 18:23:05
docbook-style-xsl-1.79.1-alt4.noarch   Вт 20 дек 2022 18:23:05
docbook-dtds-4.5-alt1.noarch          Вт 20 дек 2022 18:23:05
dict-1.12.1-alt4.1.x86_64            Вт 20 дек 2022 18:23:05
```

### Поиск пакета в системе.

Команда **grep** поможет определить, установлен пакет в системе или нет:

```
$ rpm -qa | grep yodl-docs
```

Вывод:

```
yodl-docs-4.03.00-alt2.noarch
```

### Проверка файла, относящегося к пакету.

Предположим, что нужно узнать, к какому конкретному пакету относится файл. Для этого используют команду:

```
$ rpm -qf /usr/share/doc/yodl-doc
```

Вывод:

```
yodl-docs-4.03.00-alt2.noarch
```

### Вывод информации о пакете.

Чтобы получить информацию о пакете, установленном в систему, используем команду:

```
$ rpm -qi yodl-docs
```

Вывод:

```
Name       : yodl-docs
Epoch     : 1
Version   : 4.03.00
Release   : alt2
DistTag   : sisyphus+271589.100.1.2
Architecture: noarch
Install Date: Чт 22 дек 2022 18:09:10
Group     : Documentation
Size      : 3701571
License   : GPL
Signature : DSA/SHA1, Чт 13 мая 2021 05:44:49, Key ID 95c584d5ae4ae412
Source RPM : yodl-4.03.00-alt2.src.rpm
Build Date : Чт 13 мая 2021 05:44:44
Build Host : darktemplar-sisyphus.hasher.altlinux.org
Relocations : (not relocatable)
Packager  : Aleksei Nikiforov <darktemplar@altlinux.org>
Vendor    : ALT Linux Team
URL       : https://gitlab.com/fbb-git/yodl
Summary   : Documentation for Yodl
Description :
Yodl is a package that implements a pre-document language and tools to
process it. The idea of Yodl is that you write up a document in a
pre-language, then use the tools (eg. yodl2html) to convert it to some
final document language. Current converters are for HTML, ms, man, LaTeX
SGML and texinfo, plus a poor-man's text converter. Main document types
are "article", "report", "book" and "manpage". The Yodl document
language is designed to be easy to use and extensible.
```

### Обновление пакета.

Для обновления пакета используется параметр **-Uvh**.

```
$ rpm -Uvh yodl-docs-4.03.00-alt2.noarch.rpm
```

Вывод:

```
Подготовка...
##### [100%]
пакет yodl-docs-1:4.03.00-alt2.noarch уже установлен
```

### ПРИМЕЧАНИЕ

Справку по ключам можно получить, набрав в консоли команду `rpm --help`

# Что такое RPM-пакет?

RPM-пакет - это архив, содержащий в себе архив `.cpio` с файлами, а также метаданные - имя пакета, его описание, зависимости и т.д. Менеджер пакетов RPM использует эти метаданные для проверки наличия необходимых пакетов из списка зависимостей, исполнения инструкций по установке файлов и сохранения общей информации о пакете у себя в базе.

Существует два типа RPM-пакетов:

- SRPM-пакеты (исходники) - архив с расширением `.src.rpm`. SRPM содержит исходный код, при необходимости патчи к нему и спес-файл, в котором описывается, как собрать исходный код в RPM-пакет.
- RPM-пакеты - архив с расширением `.rpm`. RPM содержит исполняемые файлы и библиотеки.

## Подготовка к сборке RPM-пакетов

Пакет `rpmdevtools`, установленный на этапе [Необходимые пакеты](#), предоставляет несколько утилит, упрощающие подготовку к сборке RPM-пакетов. Чтобы перечислить эти утилиты, выполните в консоли следующую команду:

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

```
/usr/bin/rpmdev-bumpspec  
/usr/bin/rpmdev-checksig  
/usr/bin/rpmdev-cksum  
/usr/bin/rpmdev-diff  
/usr/bin/rpmdev-extract  
/usr/bin/rpmdev-md5  
/usr/bin/rpmdev-newinit  
/usr/bin/rpmdev-newspec  
/usr/bin/rpmdev-packager  
/usr/bin/rpmdev-rmdevelrpms  
/usr/bin/rpmdev-setuptree  
/usr/bin/rpmdev-sha1  
/usr/bin/rpmdev-sha224  
/usr/bin/rpmdev-sha256  
/usr/bin/rpmdev-sha384  
/usr/bin/rpmdev-sha512  
/usr/bin/rpmdev-sort  
/usr/bin/rpmdev-sum  
/usr/bin/rpmdev-vercmp  
/usr/bin/rpmdev-wipetree  
/usr/bin/rpminfo  
/usr/bin/rpmls
```

Для получения дополнительной информации о вышеуказанных утилитах см. их страницы

руководства или диалоговые окна справки.

## Рабочее пространство для сборки RPM-пакетов

Чтобы создать дерево каталогов, которое является рабочей областью сборки RPM-пакетов, используйте утилиту `rpmdev-setuptree`, или же создайте каталоги вручную, используя утилиту `mkdir`:

```
$ rpmdev-setuptree

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

Описание созданных каталогов:

Каталог	Назначение
BUILD	Содержит все файлы, которые появляются при сборке пакета.
RPMS	Здесь формируются собранные RPM-пакеты ( <code>.rpm</code> ) в подкаталогах для разных архитектур, например, в подкаталогах <code>x86_64</code> и <code>noarch</code> .
SOURCES	Здесь находятся архивы исходного кода и патчи. Утилита <code>rpmbuild</code> ищет их здесь.
SPECS	Здесь хранятся спес-файлы.
SRPMS	Здесь находятся пакеты с исходниками ( <code>.src.rpm</code> ).

После создания дерева каталогов перейдём в файл `~/home/.rpmmacros`. В нём содержится следующая информация:

- Месторасположение структуры каталогов для сборки;
- Ключ для подписи пакетов;
- Значение поля *Packager*

```
%_topdir<----->%homedir/RPM
#%_tmppath<---->%homedir/tmp

# %packager<--->Joe Hacker <joe@email.address>
# %_gpq_name<-->joe@email.address
```

Раскомментируйте поле *Packager*: Впишите своё имя, фамилию и почту. Поле с ключём для подписи Вы заполните позже. О том, как создавать ключи, Вы узнаете в разделе [Создание](#)

**ПРИМЕЧАНИЕ**

Действия, описанные ниже, необходимы для корректного прохождения документации, они обязательны к выполнению!

Итак, если Вы воспользовались утилитой `rpmdev-setuptree`, а не создали дерево каталогов вручную, обратите внимание на файл `~/home/.rpmmacros`:

```
%_topdir<----->%homedir/RPM
#%_tmppath<---->%homedir/tmp

%packager<--->Joe Hacker <joe@email.address>
# %_gpg_name<-->joe@email.address

%_arch_install_post \
  [ "%{buildarch}" = "noarch" ] || QA_CHECK_RPATHS=1 ; \
  case "${QA_CHECK_RPATHS:-}" in [1yY]*) /usr/lib/rpm/check-rpaths ;; esac \
  /usr/lib/rpm/check-buildroot
```

В файле появилась секция `_%_arch_install_post`. Данную секцию необходимо удалить, вернув файл к исходному состоянию, иначе процесс сборки будет завершаться с ошибкой.

```
%_topdir<----->%homedir/RPM
#%_tmppath<---->%homedir/tmp

%packager<--->Valentin Sokolov <sova@altlinux.org>
# %_gpg_name<-->joe@email.address
```

# Что такое SPEC-файл?

Спес-файл можно рассматривать как "инструкцию", которую утилита `rpmbuild` использует для фактической сборки RPM-пакет. Он сообщает системе сборки, что делать, определяя инструкции в серии разделов. Разделы определены в *Преамбуле* и в *Основной части*. *Преамбула* содержит ряд элементов метаданных, которые используются в *Основной части*. Тело содержит основную часть инструкций.

## Пример .спес-файла

Данный пример взят из [ALT Linux Wiki](#).

```
Name: sampleprog
Version: 1.0
Release: alt1

Summary: Sample program specfile
Summary(ru_RU.UTF-8): Пример спек-файла для программы

License: GPLv2+
Group: Development/Other
Url: http://www.altlinux.org/SampleSpecs/program

Packager: Sample Packager <sample@altlinux.org>

Source: %name-%version.tar
Patch0: %name-1.0-alt-makefile-fixes.patch

%description
This specfile is provided as sample specfile for packages with programs.
It contains most of usual tags and constructions used in such specfiles.

%description -l ru_RU.UTF-8
Этот спек-файл является примером спек-файла для пакета с программой. Он содержит
основные теги и конструкции, используемые в подобных спек-файлах.

%prep
%setup
%patch0 -p1

%build
%configure
%make_build

%install
%makeinstall_std
%find_lang %name

%files -f %name.lang
```

```
%doc AUTHORS ChangeLog NEWS README THANKS TODO contrib/ manual/
%_bindir/*
%_mandir/*

%changelog
* Sat Sep 33 3001 Sample Packager <sample@altlinux.org> 1.0-alt1
- initial build
```

## Пункты преамбулы

В этой таблице перечислены элементы, используемые в разделе преамбулы файла спецификации RPM:

SPEC Директива	Определение
<b>Name</b>	Базовое имя пакета, которое должно совпадать с именем файла.
<b>Version</b>	Версия upstream-кода.
<b>Release</b>	Релиз пакета используется для указания номера сборки пакета при данной версии upstream-кода. Как правило, установите начальное <b>alt1</b> и увеличивайте его с каждым новым выпуском пакета, например: alt1, alt2, alt3 и т.д. Сбросьте значение до alt1 при создании новой версии программного обеспечения.
<b>Summary</b>	Краткое, в одну строку, описание пакета.
<b>License</b>	Лицензия на собираемое программное обеспечение.
<b>Group</b>	Используется для указания категории, к которой относится пакет. Указанная группа должна находиться в списке групп, известном RPM. Этот список располагается в файле <code>/usr/lib/rpm/GROUPS</code> , идущим вместе с пакетом <code>rpm</code> .
<b>URL</b>	Полный URL-адрес для получения дополнительной информации о программе. Чаще всего это ссылка на <b>GitHub</b> upstream-проекта для собираемого программного обеспечения.
<b>Source0</b>	Путь или URL-адрес к сжатому архиву исходного кода (не исправленный, исправления обрабатываются в другом месте). Этот раздел должен указывать на доступное и надежное хранилище архива, например, на upstream-страницу, а не на локальное хранилище сборщика. При необходимости можно добавить дополнительные исходные директивы, каждый раз увеличивая их количество, например: <code>Source1</code> , <code>Source2</code> , <code>Source3</code> и так далее.



<b>Patch0</b>	Название первого патча, который при необходимости будет применен к исходному коду. При необходимости можно добавить дополнительные директивы PatchX, увеличивая их количество каждый раз, например: Patch1, Patch2, Patch3 и так далее.
<b>BuildArch</b>	Если пакет не зависит от архитектуры, например, если он полностью написан на интерпретируемом языке программирования, установите для этого значение <b>BuildArch: noarch</b> . Если этот параметр не задан, пакет автоматически наследует архитектуру компьютера, на котором он собран, например <b>x86_64</b> .
<b>BuildRequires</b>	Разделённый запятыми или пробелами список пакетов, необходимых для сборки программы. Может быть несколько записей <b>BuildRequires</b> , каждая в отдельной строке в SPEC файле.
<b>Requires</b>	Разделённый запятыми или пробелами список пакетов, необходимых программному обеспечению для запуска после установки. Это его <b>зависимости</b> Может быть несколько записей <b>Requires</b> , каждая в отдельной строке в SPEC файле.
<b>ExcludeArch</b>	Если часть программного обеспечения не может работать на определенной архитектуре процессора, Вы можете исключить эту архитектуру здесь.

Директивы **Name**, **Version** и **Release** содержат имя RPM-пакета. Эти три директивы часто называют **N-V-R** или **NVR**, поскольку имена RPM-пакета имеют формат **NAME-VERSION-RELEASE**.

Вы можете получить пример **NAME-VERSION-RELEASE**, выполнив запрос с использованием **rpm** для конкретного пакета:

```
$ rpm -q rpmdevtools
rpmdevtools-8.10-alt2.noarch
```

Здесь **rpmdevtools** - это имя пакета, **8.10** - версия, а **alt2** - релиз. Последний маркер **noarch** - сведения об архитектуре. В отличие от **NVR**, маркер архитектуры не находится под прямым управлением сборщика, а определяется средой сборки **rpmbuild**. Исключением из этого правила является архитектурно-независимый пакет **noarch**.

## Составляющие основной части

В этой таблице перечислены элементы, используемые в теле файла спецификации RPM-пакета:

SPEC Директива	Определение
----------------	-------------

<code>%description</code>	Полное описание программного обеспечения, входящего в комплект поставки RPM. Это описание может занимать несколько строк и может быть разбито на абзацы.
<code>%prep</code>	Команда или серия команд для подготовки программного обеспечения к сборке, например, распаковка архива из Source0. Эта директива может содержать сценарий оболочки (shell скрипт).
<code>%build</code>	Команда или серия команд для фактической сборки программного обеспечения в машинный код (для скомпилированных языков) или байт-код (для некоторых интерпретируемых языков).
<code>%install</code>	Раздел, который во время сборки пакета эмулирует конечные пути установки файлов в систему. Команда или серия команд для копирования требуемых артефактов сборки из <code>%builddir</code> (где происходит сборка) в <code>%buildroot</code> каталог (который содержит структуру каталогов с файлами, подлежащими сборке). Обычно это означает копирование файлов из <code>~/rpmbuild/BUILD</code> в <code>~/rpmbuild/BUILDROOT</code> и создание необходимых каталогов <code>~/rpmbuild/BUILDROOT</code> . Это выполняется только при создании пакета, а не при установке пакета конечным пользователем. Подробности см. в разделе <a href="#">Работа со SPEC файлом</a> .
<code>%check</code>	Команда или серия команд для тестирования программного обеспечения. Обычно включает в себя такие вещи, как модульные тесты.
<code>%files</code>	Список файлов, которые будут установлены в системе конечного пользователя.
<code>%changelog</code>	Запись изменений, произошедших с пакетом между различными <code>Version</code> или <code>Release</code> сборками.

## ПРИМЕЧАНИЕ

Конструкция `%setup` в Sisyphus RPM использует флаг `-q` (quiet) по умолчанию. Запись `%setup -q` и `%setup` полностью идентичны. Если использовать конструкцию с флагом `-v`, то будет выведена дополнительная информация в логах сборки.

"Каркас" файла спецификации с кратким описанием составляющих:

```
Name: mypackage
Version: 1.0
Release: 1
Summary: Однострочное описание пакета.

%description
Общее описание пакета. Примерно 1 абзац.

%prep
# Подготовка исходных файлов для сборки

%build
```

```
# Компиляция и сборка пакета

%install
# Установка файлов в директорию сборки
install -D -m 0644 %{SOURCE1} %{buildroot}/путь/к/файлу/example.txt

%files
# Перечисление файлов и директорий, которые будут включены в пакет
%{buildroot}/путь/к/файлу/example.txt
```

Пример выше - достаточно простой. **Срес-файл** может быть куда больше, включать больше секций и команд. В некоторых случаях один **срес-файл** используется для сборки множества пакетов.

# RPM Макросы

Макросы RPM — это прямые текстовые подстановки, которые происходят путем замены определенных выражений и условий на соответствующий текст во время процесса сборки пакета. Имена макросов начинаются с символа `%`. Они представляют собой сокращенные псевдонимы для часто используемых фрагментов текста.

Для чего нужны макросы:

- **Обеспечить желаемую функциональность:** Пакеты в репозитории Сизиф должны отвечать определённым правилам, для этого `spec`-файлы должны обеспечивать выполнение этих правил.
- **Помощь разработчику:** `spec`-файлы пишут люди, следовательно, их работу нужно свести к минимуму, который и требует участия человека. Майнтейнер не должен копировать блоки кода из файла в файл, так как данная работа занимает время, силы, и чревата ошибками. Для таких случаев существуют **макросы**. Если какой-то код появляется в разных `spec`-файлах более одного раза, то надо написать макрос(ы).
- **Сделать `spec`-файлы более читабельными:**

Людам, пересобирающим пакет, или собирающим новый аналогичный пакет, опираясь на другие `spec`-файлы, будет удобнее, если в наименовании, расположении и использовании различных элементов `spec`-файлов будет определенный порядок.

Просмотреть список доступных макросов и их значения можно, выполнив команду:

```
rpm --showrc
```

Получить значение, раскрываемое макросом можно, использовав команду `rpm --eval {<имя_макроса>}`.

У нас есть макрос `_%sysconfdir`. Раскроем его:

```
$ rpm --eval %_sysconfdir  
  
/etc
```

Макросы можно использовать внутри других макросов. Так, например, если название архива исходных текстов проекта формируется из его имени и версии (директивы `Name` и `Version` транслируются в определённые макросы, о чём будет рассказано ниже), то директива задания пути к файлу может выглядеть следующим образом:

```
Source0: %{name}-%{version}.tar.gz
```

# Макросы путей системных каталогов

В этой таблице представлены макросы системных путей

Макрос	Замена	Описание
<code>%_usr</code>	<code>/usr</code>	-----
<code>%_var</code>	<code>/var</code>	-----
<code>%_bindir</code>	<code>/usr/bin</code>	----
<code>%_sbin</code>	<code>/usr/sbin</code>	----
<code>%_libexecdir</code>	<code>/usr/lib</code>	----
<code>%_localstatedir</code>	<code>/var/lib</code>	----
<code>%_datadir</code>	<code>/usr/share</code>	----
<code>%_tmpfilesdir</code>	<code>/lib/tmpfiles.d</code>	----
<code>%_desktopdir</code>	<code>/usr/share/application</code>	----
<code>%_</code>		----

## Макросы меню

<code>%_menudir</code>	<code>/usr/lib/menu</code>	----
<code>%_iconsdir</code>	<code>/usr/share/icons</code>	----
<code>%_miconsdir</code>	<code>/usr/share/icons/hicolor</code> <code>/16x16/apps</code>	----
<code>%_liconsdir</code>	<code>/usr/share/icons/hicolor</code> <code>/48x48/apps</code>	----

## Другие системные макросы

<code>%_initdir</code>	<code>/etc/rc.d/init.d</code>	----
<code>%_lockdir</code>	<code>/var/lock</code>	----
<code>%_logdir</code>	<code>/var/log</code>	----

## Прочие макросы

<code>%intel</code>	<code>i386 i486 i586 i686 i786</code> <code>i886 i986 pentium2</code> <code>pentium3 pentium4</code>	Список архитектур <i>intel</i> , совместимых с <i>i386</i>
<code>%amd</code>	<code>k6 athlon athlon_xp</code>	Список архитектур <i>amd</i> , совместимых с <i>i386</i> ,

<code>%i386</code>	<code>i386 i486 i586 i686 i786 i886 i986 pentium2 pentium3 pentium4 k6 athlon athlon_xp</code>	Список всех архитектур, совместимых с <code>i386</code> ;
--------------------	--	---

<code>%_</code>		----
<code>%_</code>		----
<code>%_</code>		----
<code>%_</code>		----
<code>%_</code>		----
<code>%_</code>		----
<code>%_</code>		----

Подробный список predefined макросов Вы можете найти на страницах: [Predefined macros](#) и [Macros by category](#).

# Пользовательские макросы

# Инструмент Gear

## Вступление

Gear (Get Every Archive from git package Repository) - это инструмент для создания пакетов RPM из репозитория git

система для работы с произвольными архивами программ. В качестве хранилища данных gear использует `git`, что позволяет работать с полной историей проекта.

Основной смысл хранения исходного кода пакетов в `git-репозитории` заключается в более эффективной и удобной совместной разработке, а также в минимизации используемого дискового пространства для хранения архива репозитория за длительный срок и минимизации трафика при обновлении исходного кода.

Идея gear заключается в том, чтобы с помощью одного файла с простыми правилами (для обработки которых достаточно `sed` и `git`) можно было бы собирать пакеты из произвольно устроенного `git-репозитория`, по аналогии с `hasher`, который был задуман как средство для сборки пакетов из произвольных 'rpm-пакетов'.

## Структура репозитория

Хотя `gear` и не накладывает ограничений на внутреннюю организацию `git-репозитория` (не считая требования наличия файла с правилами), есть несколько соображений о том, как более эффективно и удобно организовывать `git-репозитории`, предназначенные для хранения исходного кода пакетов.

### Одна сущность — один репозиторий

Не стоит помещать в один репозиторий несколько разных пакетов, за исключением случаев, когда у этих пакетов есть общий пакет-предок.

- Плюсы: Соблюдение этого правила облегчает совместную работу над пакетом, поскольку неперегруженный репозиторий легче клонировать и в целом инструмент `git` больше подходит для работы с такими репозиториями.
- Минусы: Несколько сложнее выполнять операции `fetch` и `push` в случае, когда репозитория, которые надо обработать, много. Впрочем, `fetch/push` в цикле выручает.

### Несжатый исходный код

Сжатый разными средствами (`gzip`, `bzip2` и т.п.) исходный код лучше хранить в `git-репозитории` в несжатом виде.

- Плюсы: Изменение файлов, которые помещены в репозиторий в сжатом виде, менее удобно отслеживать штатными средствами (`git diff`). Поскольку `git` хранит объекты в сжатом виде, двойное сжатие редко приводит к экономии дискового пространства. Наконец, алгоритм, применяемый для минимизации трафика при обновлении репозитория по протоколу `git`, более эффективен на несжатых данных.



- Минусы: Поскольку некоторые виды сжатия одних и тех же данных могут приводить к разным результатам, может уменьшиться степень первозданности (нативности) исходного кода.

### Распакованный исходный код

Исходный код, запакованный архиваторами (`tar`, `cpio`, `zip` и т.п.), лучше хранить в `git`-репозитории в распакованном виде.

- Плюсы: Существенно удобнее вносить изменения в конечные файлы и отслеживать изменения в них, заметно меньше трафик при обновлении.
- Минусы: Поскольку `git` из информации о владельце, правах доступа и дате модификации файлов хранит только исполняемость файлов, любой архив, созданный из репозитория, будет по этим параметрам отличаться от первозданного. Помимо потери нативности, изменение прав доступа и даты модификации может теоретически повлиять на результат сборки пакета. Впрочем, сборку таких пакетов, если они будут обнаружены, всё равно придётся исправить.

### Форматированный changelog

В `changelog` релизного `commit` имеет смысл включать соответствующий текст из `changelog` пакета, как это делают утилиты `gear-commit` (обёртка к `git commit`, специально предназначенная для этих целей) и `gear-srpmimport`. В результате можно будет получить представление об изменениях в очередном релизе пакета, не заглядывая в `src`-файл самого пакета.

## Правила экспорта

С одной стороны, для того, чтобы `srpm`-пакет мог быть импортирован в `git`-репозиторий наиболее удобным для пользователя способом, язык правил, согласно которым производится экспорт из коммита репозитория (в форму, из которой можно однозначно изготовить `srpm`-пакет или запустить сборку), должен быть достаточно выразительным.

С другой стороны, для того, чтобы можно было относительно безбоязненно собирать пакеты из чужих `gear`-репозиториях, этот язык правил должен быть достаточно простым.

Файл правил экспорта (по умолчанию в `.gear/rules`) состоит из строк формата:

```
директива: параметры
```

Параметры разделяются пробельными символами.

Директивы позволяют экспортировать:

1. Любой файл из дерева, соответствующего коммиту;
2. Любой каталог из дерева, соответствующего коммиту в виде `tar`- или `zip`-архива;
3. `nified diff` между любыми каталогами, соответствующими коммитам.

Файлы на выходе могут быть сжаты разными средствами (`gzip`, `bzip2` и т.п.). В качестве коммита может быть указан как целевой коммит (значение параметра `-t` утилиты `gear`), так

и любой из его предков при соблюдении условий, гарантирующих однозначное вычисление полного имени коммита-предка по целевому коммиту.

(Правила экспорта из gear-репозитория описаны детально в [gear-rules](#).)(ссылка под редактуру)

## Основные типы устройства gear-репозитория

Правила экспорта реализуют основные типы устройства gear-репозитория следующим образом:

### Архив с модифицированным исходным кодом

С помощью простого правила

```
tar: .
```

Всё дерево исходного кода экспортируется в один tar-архив. Если у проекта есть upstream, публикующий tar-архивы, то добавление релиза в имя tar-архива, например, с помощью правила:

```
tar: . name=@name@-@version@-@release@
```

позволяет избежать коллизий.

### Архив с немодифицированным исходным кодом и патчем, содержащем локальные изменения

Если дерево с немодифицированным исходным кодом хранится в отдельном подкаталоге, а локальные изменения хранятся в gear-репозитории в виде отдельных патч-файлов, то правила экспорта могут выглядеть следующим образом:

```
tar: package_name
copy: *.patch
```

Такое устройство репозитория получается при использовании утилиты [gear-srpmimport](#), предназначенной для быстрой миграции от srpm-файла к gear-репозиторию.

### Смешанные типы

Вышеперечисленные типы устройства gear-репозитория являются основными, но не исчерпывающими. Правила экспорта достаточно выразительны для того, чтобы реализовать всевозможные сочетания основных типов и создать полнофункциональный gear-репозиторий на любой вкус.

## Быстрый старт Gear

## Создание gear-репозитория путём импорта созданного ранее srpm-пакета.

Пусть у нас есть srpm-пакет `foobar-1.0-alt1.src.rpm`, и, к примеру, в нём находится следующее:

```
$ rpm -qpl foobar-1.0-alt1.src.rpm
foobar-1-fix.patch
foobar-2-fix.patch
foobar.icon.png
foobar-1.0.tar.bz2
foobar-plugins.tar.gz
```

Для того чтобы сделать из него gear-репозиторий, нам нужно:

1. Создать каталог, в котором будет располагаться наш архив:

```
$ mkdir foobar
$ cd foobar
```

2. Создать новый git-репозиторий:

```
$ git init
Initialized empty Git repository in .git/
```

Получившийся пустой git-репозиторий будет выглядеть примерно следующим образом:

```
$ ls -dlog .*
drwxr-xr-x 4 4096 Aug 12 34:56 .
drwxr-xr-x 6 4096 Aug 12 34:56 ..
drwxr-xr-x 8 4096 Aug 12 34:56 .git
```

Таким образом, git-репозиторий готов для импорта srpm-пакета.

3. В проекте `gear` есть утилита `gear-srpmimport`, предназначенная для автоматизации импортирования srpm-пакета в git-репозиторий:

```
$ gear-srpmimport foobar-1.0-alt1.src.rpm
Committing initial tree deadbeefdeadbeefdeadbeefdeadbeefdeadbeef
gear-srpmimport: Imported foobar-1.0-alt1.src.rpm
gear-srpmimport: Created master branch
```

После выполнения импорта git-репозиторий будет выглядеть следующим образом:

```
$ ls -Alog
drwxr-xr-x 1 4096 Aug 12 34:56 .gear
drwxr-xr-x 1 4096 Aug 12 34:56 .git
-rw-r--r-- 1 6637 Aug 12 34:56 foobar.spec
drwxr-xr-x 3 4096 Aug 12 34:56 foobar
drwxr-xr-x 3 4096 Aug 12 34:56 foobar-plugins
-rw-r--r-- 1 791 Aug 12 34:56 foobar-1-fix.patch
-rw-r--r-- 1 3115 Aug 12 34:56 foobar-2-fix.patch
-rw-r--r-- 1 842 Aug 12 34:56 foobar.icon.png
```

4. При необходимости в файл правил можно вносить изменения. Например, можно убрать сжатие исходников (соответствующие изменения следует вносить и в `.gear/rules`).

## Создание gear-репозитория на основе готового git-репозитория

1. Создать и добавить в git-репозиторий spec-файл.
2. Создать и добавить в git-репозиторий файл с правилами `.gear/rules`.

## Сборка пакета из gear-репозитория

1. Сборка пакета при помощи hasher осуществляется командой `gear-hsh`:

```
$ gear-hsh
```

2. Чтобы собрать старый пакет, который не содержит определения тега `Package` в spec-файле, следует отключить соответствующую проверку:

```
$ gear-hsh --no-sisyphus-check=gpg,packager
```

3. Сборка пакета при помощи `rpmbuild(8)` осуществляется командой `gear-rpm`:

```
$ gear-rpm -ba
```

## Фиксация изменений в репозитории

1. Для того, чтобы сделать `commit` очередной сборки пакета, имеет смысл воспользоваться утилитой `gear-commit`, которая помогает сформировать список изменений на основе записи в spec-файле:

```
$ gear-commit -a
```

2. Прежде чем сделать первый `commit`, не забудьте сконфигурировать ваш адрес. Это можно сделать глобально несколькими способами, например, прописав

соответствующие значения в `~/.gitconfig`:

```
$ git config --global user.name 'Your Name'  
$ git config --global user.email '<login>@altlinux.org'
```

Для отдельно взятого git-репозитория сконфигурировать адрес можно, прописав соответствующие значения в `.git/config` этого git-репозитория:

```
$ git config user.name 'Your Name'  
$ git config user.email '<login>@altlinux.org'
```

# Hasher start

## Что такое hasher?

**Hasher** — это инструмент безопасной и воспроизводимой сборки пакетов. Все пакеты репозитория **Сизиф** собираются с его помощью.

## Принцип действия

**Hasher** — инструмент для сборки пакетов в «чистой» и контролируемой среде. Это достигается с помощью создания в chroot минимальной сборочной среды, установки туда указанных в source-пакете сборочных зависимостей и сборке пакета в свежесозданной среде. Для сборки каждого пакета сборочная среда создаётся заново.

Такой принцип сборки имеет несколько следствий:

1. Все необходимые для сборки зависимости должны быть указаны в пакете. Для облегчения поддержки сборочных зависимостей в актуальном состоянии в **Сизифе** придуман инструмент под названием `buildreq`,
2. Сборка не зависит от конфигурации компьютера пользователя, собирающего пакет, и может быть повторена на другом компьютере,
3. Изолированность среды сборки позволяет с лёгкостью собирать на одном компьютере пакеты для разных дистрибутивов и веток репозитория — для этого достаточно лишь направить `hasher` на различные репозитории для каждого сборочного окружения.

## Настройка Hasher

Установка Hasher:

```
# apt-get install hasher
```

Добавление пользователя:

Hasher использует специальных вспомогательных пользователей и группу `hashman` для своей работы, поэтому каждого пользователя, желающего использовать `hasher`, перед началом работы нужно зарегистрировать:

```
# hasher-useradd USER
```

Настройка сборочной среды:

Для работы `hasher` требуется создать директорию, в которой будет строиться сборочная среда:

```
$ mkdir ~/.hasher
```

Рабочий каталог (в данном случае `~/.hasher`) должен быть доступен на запись пользователю, запускающему сборку.

Кроме того, его нельзя располагать на файловой системе, которая смонтирована с опциями `noexec` или `nodew` — в таких условиях `hasher` не сможет создать корректное сборочное окружение.

Сборочное окружение можно создать явно:

```
$ hsh --initroot-only ~/.hasher
```

Явное создание необязательно — при необходимости оно будет произведено при первой сборке пакета.

Hasher берёт пакеты для установки из АРТ-источников. По умолчанию в сборочную среду копируется список источников, указанный в конфигурации АРТ хост-системы; также можно явно задать дополнительные репозитории, указав альтернативный файл конфигурации АРТ:

```
$ hsh --apt-config=branch4.1-apt.conf --initroot-only ~/.hasher
```

В таком файле конфигурации необходимо указать расположение файла с АРТ-источниками:

```
$ hsh --apt-config=branch4.1-apt.conf --initroot-only ~/.hasher
```

## Сборка в hasher

Сборка происходит от обычного пользователя, добавленного с помощью `hasher-useradd`:

```
hsh ~/.hasher /home/work/rpm/package.src.rpm
```

При удачной сборке полученные пакеты будут лежать в `~/.hasher/репо/<платформа>/RPMS.hasher/`, в противном случае на `stdout` будет выведена информация об ошибках сборки.

Создаваемый `hasher` репозиторий является обычным АРТ-репозиторием и может быть использован в `sources.list[3]`. Также он будет использован при дальнейшей сборке пакетов (это поведение можно регулировать ключом `--without-stuff`).

Если вы держите сборочную среду в `tmpfs` (см. ниже), каталог `~/.hasher/репо`, вероятно, не переживёт перезагрузку системы. Репозиторий можно переместить в постоянное место, указав в настройочном файле `hasher` параметр `~/.hasher/config`

def\_repo=постоянное\_хранилище (или вызвав hasher с ключом --repo).

## Сборочные зависимости

Сборочные зависимости RPM делятся на два вида:

1. необходимые для корректного создания src.rpm из spec-файла (содержащие определения RPM-макросов, используемых в spec-файле),
2. все остальные (необходимые для непосредственной сборки).

Поскольку **hasher** собирает пакеты из src.rpm (не считая поддержки **gear**), то для сборки необходимо иметь в хост-системе установленные сборочные зависимости первого типа. Большинство таких зависимостей (но пока не все) содержатся в пакетах с названием **rpm-build-\***.

Поскольку сборка src.rpm либо завершается неудачно (при отсутствии сборочной зависимости первого типа), либо корректно, то собирать src.rpm-пакеты в хост системе можно с помощью **--nodeps**:

```
rpm -bs --nodeps package.spec
```

Сам **hasher**, в отличие от **gear**, не предъявляет никаких требований к разделению сборочных зависимостей на первый и второй тип. Однако для совместимости с **gear** и для улучшения описания spec-файла рекомендуется распределять их так:

- В поле BuildRequires(pre) помещать сборочные зависимости, требуемые для сборки src.rpm,
- В поле BuildRequires — все остальные.

### ПРИМЕЧАНИЕ

в поле BuildRequires(pre) нельзя использовать макросы.

## man hasher

Для получение подробной справки и пояснении команд - воспользуйтесь мануалом **hasher**:

```
$ man hsh
```

## Монтирование файловых систем внутри hasher

Некоторым приложениям для сборки требуется смонтированная файловая система (например, /proc). **hasher** поддерживает монтирование дополнительных файловых систем в сборочную среду.

Монтирование происходит при одновременном выполнении следующих четырех условий:



- файловая система описана в файле `/etc/hasher-priv/fstab`, либо является одной из предопределённых: `/proc`, `/dev/pts`, `/sys`; в конфигурации `hasher-priv (/etc/hasher-priv/system)` ФС указана в опции `allowed_mountpoints`;
- файловая система указана в опции `--mountpoints` при запуске `hasher`, либо, аналогично, в ключе `known_mountpoints` конфигурационного файла `hasher (~/.hasher/config)`;
- файловая система указана сборочной зависимостью (например, `BuildReq: /proc`) собираемого пакета, прямой или косвенной (через зависимости сборочных зависимостей пакета).

# Примеры сборки пакетов с использованием инструментов АЛЬТ

Для примера сборки пакета будем использовать программу для вывода системных уведомлений о текущей дате и времени. Ссылка на github-репозиторий с исходными текстами программ на языках C++ ([Notification](#)) и Python ([DBusTimer\\_Example](#))

Структура репозитория для данных программ идентична: Главный файл (.cpp или .py) и два юнита systemd (.service и .timer)

Вдаваться в подробности написания кода мы не будем, так как основная цель - сборка пакета, а не разработка приложения.

Файл .timer - юнит systemd, который при истечении заданного времени будет вызывать скрипт .py, который выводит уведомление о дате и времени. После срабатывания таймер снова начинает отсчёт до запуска скрипта.

Файл .service - содержит описание, расположение скрипта .py и интерпретатора, который будет обрабатывать скрипт.

## Подготовка пространства

Первым шагом Вам необходимо клонировать репозиторий в Вашу рабочую директорию, используя команду `git clone` (адрес репозитория `DBusTimer_Example` из ссылки выше):

```
$ git clone https://github.com/danila-Skachedubov/DBusTimer_example.git
```

```
Cloning into 'DBusTimer_example'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 5 (delta 0), reused 5 (delta 0), pack-reused 0
Receiving objects: 100% (5/5), done.
```

В рабочей директории появится каталог с названием проекта: `DBusTimer_Example`.

Создадим каталог `.gear` и перейдём в него.

```
DBusTimer_example $ mkdir .gear
DBusTimer_example $ cd .gear/
.gear $
```

В каталоге `.gear` создадим два файла: правила для `gear` - `rules` и `spec` файл - `dbustimer.spec`

```
.gear $ touch rules dbustimer.spec
```

После всех изменений содержание каталога проекта примет следующий вид:

```
DBusTimer_example $ ls -la
итого 28
drwxr-xr-x  4 sova domain users 4096 апр 20 14:20 .
drwxr-xr-x 10 sova domain users 4096 апр 20 14:06 ..
drwxr-xr-x  2 sova domain users 4096 апр 20 14:24 .gear
drwxr-xr-x  8 sova domain users 4096 апр 20 14:06 .git
-rwxr-xr-x  1 sova domain users  413 апр 20 14:06 script_dbus.py
-rw-r--r--  1 sova domain users  186 апр 20 14:06 script_dbus.service
-rw-r--r--  1 sova domain users  106 апр 20 14:06 script_dbus.timer
DBusTimer_example $ ls .gear/
dbustimer.spec  rules
```

## Написание spec файла и правил Gear

Следующим этапом сборки будет написание **spec** файла и правил для gear.

В каталоге `.gear` откроем файл **rules**. Заполним его следующим содержимым:

```
tar: .
spec: .gear/dbustimer.spec
```

Первая строка указывает, что проект будет упакован в **.tar** архив. Вторая строка указывает путь к расположению **.spec** файла. На этом этапе редактирование **rules** заканчивается.

Перейдём к написанию **.spec** файла.

В заголовке или шапке спек файла находятся секции Name, Version, Release, Summary, License, Group, BuildArch, BuildRequires, Source0.

Заполнив данные секции, заголовок спек файла примет вид:

```
Name: dbustimer
Version: 0.4
Release: alt1

Summary: Display system time
License: GPLv3+
Group: Other
BuildArch: noarch

BuildRequires: rpm-build-python3
```

Стандартная схема Name-Version-Release, содержащая в себе имя пакета, его версию и релиз сборки. Поле Summary включает в себя краткое описание пакета. License - лицензия, под которой выпускается данное ПО. В данном случае - GPLv3. Группа - категория, к которой

относится пакет. Так как это тестовый пакет для примера, выставим группу "Other". BuildRequires - пакеты, необходимые для сборки. Так как исходный код написан на python3, нам необходим пакет `rpm-build-python3` с макросами для сборки скриптов Python. Source0 - путь к архиву с исходниками (`%name-%version.tar`). На этом заголовок `.spec` файла заканчивается.

Далее - тело, или основная часть `.spec` файла. В ней описывается сам процесс сборки и инструкции к преобразованию исходных файлов.

Начнём с заполнения полей `%description` и `%prep`.

```
%description
This program displays notifications about the system time with a frequency of one
hour.

%prep
%setup -q
```

В секции `%description` находится краткое описание программы. Секция `%prep` отвечает за подготовку программы к сборке. Макрос `%setup` распаковывает исходный код перед компиляцией.

В секции `%install` описаны инструкции, как установить файлы пакета в систему конечного пользователя.

Вместо того, чтобы писать пути установки файлов вручную, будем использовать predefined макросы: `%python3_sitelibdir_noarch` будет раскрываться в путь `/usr/lib/python3/site-packages`. По этому пути будет создан каталог с именем пакета, в который будет помещён файл `script_dbus.py` с правами доступа 755.

Аналогичная операция будет проведена с файлами `script_dbus.timer` и `script_dbus.service`. Они должны быть установлены по пути `/etc/xdg/systemd/user`. Так как макроса, раскрывающегося в данный путь нет, будет использован макрос `%_sysconfdir`, который раскрывается в путь `/etc`.

```
%install

mkdir -p \
    %buildroot%python3_sitelibdir_noarch/%name/
install -Dm0755 script_dbus.py \
    %buildroot%python3_sitelibdir_noarch/%name/

mkdir -p \
    %buildroot%_sysconfdir/xdg/systemd/user/
cp script_dbus.timer script_dbus.service \
    %buildroot%_sysconfdir/xdg/systemd/user/
```

Команда `mkdir -p \ %buildroot%python3_sitelibdir_noarch/%name/` создаёт каталог `dbustimer` в окружении `buildroot` по пути `/usr/lib/python3/site-packages`

Следующим действием происходит установка файла `script_dbus.py` с правами 755 в каталог `/usr/lib/python3/site-packages/dbustimer/` в окружении `buildroot`.

Аналогично создаётся каталог `%buildroot%_sysconfdir/xdg/systemd/user/`, в который копируются файлы `.service` и `.timer`

### Секция `%files`

```
%files
%python3_sitelibdir_noarch/%name/script_dbus.py
/etc/xdg/systemd/user/script_dbus.service
/etc/xdg/systemd/user/script_dbus.timer
```

В секции `%files` описано, какие файлы и каталоги с соответствующими атрибутами должны быть скопированы из дерева сборки в `rpm`-пакет, а затем будут копироваться в целевую систему при установке этого пакета. Все три файла из пакета будут распакованы по путям, описанным в секции `%install`.

Секция `%changelog`. Здесь описаны изменения внесённые в ПО, патчи, изменения методологии сборки

```
%changelog
* Thu Apr 13 2023 Danila Skachedubov <dan@altlinux.org> 0.4-alt1
- Update system
- Changed access rights
```

После всех манипуляций Ваш `.spec` файл будет выглядеть следующим образом:

```
Name: dbustimer
Version: 0.4
Release: alt1

Summary: Display system time
License: GPLv3+
Group: Other
BuildArch: noarch

BuildRequires: rpm-build-python3

Source0: %name-%version.tar

%description
This program displays notifications about the system time with a frequency of one hour.
```

```

%prep
%setup

%install

mkdir -p \
    %buildroot%python3_sitelibdir_noarch/%name/
install -Dm0755 script_dbus.py \
    %buildroot%python3_sitelibdir_noarch/%name/

mkdir -p \
    %buildroot%_sysconfdir/xdg/systemd/user/
cp script_dbus.timer script_dbus.service \
    %buildroot%_sysconfdir/xdg/systemd/user/

%files
%python3_sitelibdir_noarch/%name/script_dbus.py
/etc/xdg/systemd/user/script_dbus.service
/etc/xdg/systemd/user/script_dbus.timer

%changelog
* Thu Apr 13 2023 Danila Skachedubov <dan@altlinux.org> 0.4-alt1
- Update system
- Changed access rights

```

Сохраним файл и перейдём в основную директорию нашего проекта.

Теперь необходимо добавить созданные нами файлы на отслеживание git. Сделать это можно с помощью команды:

```
$ git add .gear/rules .gear/dbustimer.spec
```

После добавление файлов на отслеживание, запустим сборку с помощью инструментов gear и hasher следующей командой:

```
$ gear-hsh --no-sisyphus-check --commit -v
```

Если сборка прошла успешно, собранный пакет `dbustimer-0.4-alt1.noarch.rpm` будет находиться в каталоге `~/hasher/repo/x86_64/RPMS.hasher/`.

## Описание пакета с исходными текстами на C++

Ссылка на GitHub репозиторий: [Notification](#).

Данная программа выводит системное уведомление о текущей дате и времени в формате: `День недели, месяц, число, чч:мм:сс, год`.

В репозитории находятся следующие файлы:

1. `.gear` - каталог с правилами `gear` и `.spec` файлом
2. `Makefile` — набор инструкций для программы `make`, которая собирает данный проект.
3. `notify.cpp` - исходный код программы
4. `notify.service` - юнит данной программы для `systemd`
5. `notify.timer` - юнит `systemd`, запускающий вывод уведомления о дате и времени с периодичностью в один час.

В каталоге `.gear` находятся два файла:

1. `rules` - правила для упаковки архива для `gear`
2. `notify.spec` - файл спецификации для сборки пакета

Остановимся подробнее на этих двух файлах.

Перейдём к содержанию файла `rules`

```
tar: .  
spec: .gear/notify.spec
```

Первая строка - указания для `gear`, в какой формат упаковать файлы для последующей сборки. В данном проекте архив будет иметь вид `name-version.tar`.

Вторая строка - путь к `.spec` файлу с инструкциями по сборке текущего пакета.

```
Name: notify  
Version: 0.1  
Release: alt1  
  
Summary: Display system time every hour  
License: GPLv3+  
Group: Other  
  
BuildRequires: make  
BuildRequires: gcc-c++  
BuildRequires: libsystemd-devel Работа с ключами разработчика.  
  
Создание заявки  
  
Source0: %name-%version.tar  
  
%description  
This test program displays system date and time every hour via notification  
  
%prep  
%setup -q
```

```

%build
%make_build

%install

mkdir -p \
    %buildroot/bin/
install -Dm0644 %name %buildroot/bin/

mkdir -p \
    %buildroot%_sysconfdir/xdg/systemd/user/
cp %name.timer %name.service \
    %buildroot%_sysconfdir/xdg/systemd/user/

%files
/bin/%name
/etc/xdg/systemd/user/%name.service
/etc/xdg/systemd/user/%name.timer

%changelog
* Thu Apr 13 2023 Sergey Okunkov <sok@altlinux.org> 0.1-alt1
- Finished my task

```

В заголовке или "шапке" .spec файла описаны следующие поля:

```

Name: notify
Version: 0.1
Release: alt1

Summary: Display system time every hour
License: GPLv3+
Group: Other

BuildRequires: make
BuildRequires: gcc-c++
BuildRequires: libsystemd-devel

Source0: %name-%version.tar

```

Стандартная схема Name-Version-Release, содержащая в себе имя пакета, его версию и релиз сборки. Поле Summary включает в себя краткое описание пакета. License - лицензия, под которой выпускается данное ПО. В данном случае - GPLv3. Группа - категория, к которой относится пакет. Так как это тестовый пакет для примера, выставим группу "Other". BuildRequires - пакеты, необходимые для сборки. Так как исходный код написан на c++, нам необходим компилятор **g + +**, система сборки программы - **make** и библиотека для работы с модулями systemd - **libsystemd-devel**. Source0 - путь к архиву с исходниками (%name-%version.tar). На этом заголовок .spec файла заканчивается.



Тело .spec файла, или же его основная часть.

```
%description
This test program displays system date and time every hour via notification

%prep
%setup -q

%build
%make

%install

mkdir -p \
    %buildroot/bin/
install -Dm0644 %name %buildroot/bin/

mkdir -p \
    %buildroot%_sysconfdir/xdg/systemd/user/
cp %name.timer %name.service \
    %buildroot%_sysconfdir/xdg/systemd/user/

%files
/bin/%name
/etc/xdg/systemd/user/%name.service
/etc/xdg/systemd/user/%name.timer
```

**Секция %description** - описание того, что делает программа. В данном примере - вывод системного уведомления с датой и временем.

**Секция %prep.** Макрос %setup с флагом `-q` распаковывает архив, описанный в секции Source0.

В секции %build происходит *сборка исходного кода*. Так как в примере присутствует Makefile для автоматизации процесса сборки, то в секции будет указан макрос %make\_build, использующий Makefile для сборки программы.

### Секция %install

Здесь происходит эмуляция конечных путей при установке файлов в систему. Мы переносим файл в buildroot в те пути, куда файлы будут помещены после установки пакета в систему пользователя. Так как файла три, для каждого пропишем конечный путь:

1. `notify` - скомпилированный бинарный файл. В Unix-подобных системах бинарные файлы располагаются в каталоге `/bin`. `mkdir -p %buildroot/bin` - строка, в которой создаётся каталог `bin` в окружении `buildroot`. Следующая строка - `install -Dm0644 %name %buildroot/bin/` - установка бинарного файла `notify` в каталог `%buildroot/bin/` с разрешениями 644.
2. `%name.timer`, `%name.service` - юниты systemd. Данные юниты относятся к пользовательским

и находятся в `/etc/xdg/systemd/user/`. Как и для предыдущего файла, создадим в окружении `buildroot` каталог `mkdir -p %buildroot%_sysconffdir/xdg/systemd/user/`. В пути использован макрос `%_sysconffdir`, который заменяется путём `/etc`. Следующая строка `cp %name.timer %name.service %buildroot%_sysconffdir/xdg/systemd/user/` - переносит данные файлы по заданному пути в окружении `buildroot`.

### Секция `%files`

Описывает какие файлы и директории будут скопированы в систему при установке пакета.

```
/bin/%name  
/etc/xdg/systemd/user/%name.service  
/etc/xdg/systemd/user/%name.timer
```