

# 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 .....	9
Инструмент Gear .....	12
Вступление .....	12
Структура репозитория .....	12
Правила экспорта .....	13
Основные типы устройства gear-репозитория .....	14
Быстрый старт Gear .....	14
Создание gear-репозитория путём импорта созданного ранее srpm-пакета .....	14
Создание gear-репозитория на основе готового git-репозитория .....	16
Сборка пакета из gear-репозитория .....	16
Фиксация изменений в репозитории .....	16
Подготовка программного обеспечения для упаковки .....	18
Что такое Исходный код? .....	18
Как создаются программы .....	19
Изначально скомпилированный код .....	19
Интерпретируемый код .....	19
Сборка программного обеспечения из исходного кода .....	19
Изначально скомпилированный код .....	19
Интерпретируемый код .....	20
Программное обеспечение для исправления ошибок .....	22
Установка Произвольных Артефактов .....	24
Использование команды install .....	24
Использование команды make install .....	25
Подготовка исходного кода для упаковки .....	26
Создание Tarball с исходным кодом .....	26
bello .....	26
pello .....	27
cello .....	27
Примеры сборки пакетов с использованием инструментов Альт .....	29

Подготовка пространства .....	29
Написание спрес файла и правил Gear .....	30
Описание пакета с исходными текстами на C++ .....	33
Программное обеспечение для исправления ошибок .....	37
Система контроля версий .....	37
Дополнительные материалы .....	39
Подпись пакетов .....	39
Добавление подписи к пакету .....	39
Замена подписи пакета .....	40
Подпись во время сборки .....	41
Подробнее о макросах .....	42
Определение Ваших Собственных Макросов .....	42
%setup .....	43
%files .....	46
Встроенные макросы .....	46
RPM Макросы, предоставляемые дистрибутивом .....	46
Пользовательские макросы .....	46
Eroch, Скриптлеты и Триггеры .....	47
Eroch .....	47
Скриптлеты и Триггеры .....	47
Условные обозначения RPM .....	51
Синтаксис условий RPM .....	51
Примеры условий RPM .....	51
Join .....	54
Процедура принятия в Team .....	54
Сбор информации .....	54
Создание заявки .....	55
Создание SSH и GPG ключей .....	56
SSH Key .....	56
GPG Key .....	56
Hasher start .....	58
Что такое hasher? .....	58
Принцип действия .....	58
Настройка Hasher .....	58
Сборка в hasher .....	59
Сборочные зависимости .....	60
man hasher .....	60

# 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-asciidoctor-doc-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.

Существует два типа RPM-пакетов:

- SRPM-пакеты (исходники) - архив с расширением `.src.rpm`
- RPM-пакеты - архив с расширением `.rpm`.

SRPM и RPM-пакеты имеют общий формат, но имеют разное содержимое и служат разным целям. SRPM содержит исходный код, при необходимости патчи к нему и `срес-файл`, в котором описывается, как собрать исходный код в RPM-пакет.

RPM-пакет содержит исполняемые файлы, созданные из исходного кода и патчей, если таковые имелись, библиотеки и метаданные. 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.

## 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
```

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 появляются готовые RPM-пакеты ( <i>.rpm</i> ) в подкаталогах для разных архитектур, например, в подкаталогах <i>x86_64</i> и <i>noarch</i> .
SOURCES	Здесь находятся архивы исходного кода и патчи. Утилита <i>rpmbuild</i> ищет их здесь.
SPECS	Здесь хранятся спес-файлы.
SRPMS	Здесь находятся пакеты с исходниками ( <i>.src.rpm</i> ).

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>
# %_gpg_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 SSH и GPG ключей](#).

#### 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>
# %_pgp_name<-->joe@email.address
```

# Инструмент Gear

## Вступление

Gear (Get Every Archive from git package Repository) - 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 инструментарий `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 первоначального. 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 `spec`-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 -ldog .*
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 спес-файл).

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

1. Создать и добавить в git-репозиторий спес-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 Packager в спес-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 спес-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'
```

# Подготовка программного обеспечения для упаковки

Эта глава посвящена исходному коду и созданию программного обеспечения, которые являются необходимой основой для сборки RPM.

## Что такое Исходный код?

**Исходный код** - это понятные для человека инструкции к компьютеру, в которых описывается, как выполнить вычисления. Исходный код выражается с помощью [языка программирования](#).

В этом руководстве представлены три версии программы **Hello World**, каждая из которых написана на разных языках программирования. Программы, написанные на этих трех разных языках, упаковываются по-разному и охватывают три основных варианта использования RPM упаковщиком ПО.

### ПРИМЕЧАНИЕ

Существуют тысячи языков программирования. В этом документе представлены только три из них, но их достаточно для концептуального обзора.

Программа **Hello World**, написанная на [bash](#):

**bello**

```
#!/bin/bash

printf "Hello World\n"
```

**Hello World**, написанная на [Python](#):

**pello.py**

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

print("Hello World")
```

**Hello World**, написанная на [C](#):

**cello.c**

```
#include <stdio.h>

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

```
}
```

Целью каждой из трех программ является вывод строки **Hello World** в командной строке.

#### ПРИМЕЧАНИЕ

Знание того, как программировать, не обязательно для упаковщика программного обеспечения, но полезно.

## Как создаются программы

### Изначально скомпилированный код

### Интерпретируемый код

#### Программы с прямой интерпретацией

#### Программы, скомпилированные в байт-код

## Сборка программного обеспечения из исходного кода

### Изначально скомпилированный код

#### Ручная сборка

#### Автоматическая сборка

Вместо того, чтобы создавать исходный код вручную, вы можете автоматизировать сборку. Это обычная практика, используемая в крупномасштабном программном обеспечении. Автоматизация сборки осуществляется путем создания **Makefile** и последующим запуском утилиты **GNU make**.

Чтобы настроить автоматическую сборку, создайте файл с именем **Makefile** в том же каталоге, что и **cello.c**:

#### Makefile

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

clean:
    rm cello
```

Теперь, чтобы собрать программу, просто запустите **make**:

```
$ make
```

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

Поскольку сборка уже создана, **make clean** очистит её, а затем снова запустит **make**:

```
$ make clean
rm cello

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

Опять же, попытка сборки после другой сборки ничего не даст:

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

Наконец, программа выполнится:

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

Теперь Вы скомпилировали программу как вручную, так и с помощью инструмента сборки.

## Интерпретируемый код

Следующие два примера демонстрируют компиляцию в байт-код программы, написанной на **Python** и прямую интерпретацию программы, написанной на **bash**.

### ПРИМЕЧАНИЕ

В двух приведенных ниже примерах **#!** строка в верхней части файла называется **shebang** и не является частью исходного кода.

**shebang** позволяет использовать текстовый файл в качестве исполняемого файла: загрузчик системной программы анализирует строку, содержащую **shebang**, чтобы получить путь к бинарному исполняемому файлу, который затем используется в качестве интерпретатора языка программирования.

### Компиляция в байт-кода

В этом примере Вы скомпилируете в байт-код **pello.py** - программу, написанную на Python, который затем выполняется виртуальной машиной Python. Исходный код Python также может быть напрямую интерпретирован, но исполнение байт-кода быстрее. Следовательно, RPM упаковщики предпочитают упаковывать скомпилированный байт-код для распространения среди конечных пользователей.

**pello.py**

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

print("Hello World")
```

Процедура программ в байт-код отличается для разных языков. Это зависит от языка, виртуальной машины языка, а также инструментов и процессов, используемых с этим языком.

#### ПРИМЕЧАНИЕ

[Python](#) часто компилируется в байт-код, но не так, как описано здесь. Следующая процедура направлена не на то, чтобы соответствовать стандартам сообщества, а на то, чтобы быть простой. Для получения практических рекомендаций по Python см. раздел [Упаковка и распространение программного обеспечения](#).

Компиляция `pello.py` в байт-код:

```
$ python -m compileall pello.py

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

Выполните байт-код в `pello.pyc`:

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

#### Напрямую интерпретированный код

В этом примере Вы будете интерпретировать программу `bello` написанную на встроенном языке оболочки `bash`.

`bello`

```
#!/bin/bash

printf "Hello World\n"
```

Для программ, написанных на языках сценариев оболочки, таких как `bash`, используется прямая интерпретация. Следовательно, Вам нужно только сделать файл с исходным кодом исполняемым и запустить его:

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



# Программное обеспечение для исправления ошибок

**Patch** - это исходный код, который исправляет другой исходный код. Он отформатирован как *diff*, потому что представляет разницу между двумя версиями текста. Разница создаётся с помощью утилиты **diff**, которая затем применяется к исходному коду с помощью утилиты **patch**.

## ПРИМЕЧАНИЕ

Разработчики программного обеспечения часто используют системы контроля версий, такие как **git**, для управления своей кодовой базой. Такие инструменты предоставляют свои собственные методы создания различий или исправления программного обеспечения.

В следующем примере мы создаем исправление из исходного кода с помощью **diff**, а затем используем **patch**. Исправление используется в следующих разделах при создании RPM и работе со *.spec*-файлом. [Работа со SPEC файлами](#).

Как исправление связано с упаковкой RPM? В упаковке, вместо того, чтобы просто изменять исходный код, мы сохраняем его и используем на нем исправления.

Чтобы создать патч для **cello.c**:

1. Сохраним исходный код:

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

Это наиболее распространённый способ сохранить файл исходного кода.

2. Изменим **cello.c**:

```
#include <stdio.h>

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

3. Сгенерируем патч используя утилиту **diff**:

## ПРИМЕЧАНИЕ

Мы используем несколько общих аргументов для утилиты **diff**. Для получения дополнительной информации о них см. руководство по использованию **diff**.

```
$ diff -Naur cello.c.orig cello.c
--- cello.c.orig      2016-05-26 17:21:30.478523360 -0500
```

```
+++ cello.c      2016-05-27 14:53:20.668588245 -0500
@@ -1,6 +1,6 @@
#include<stdio.h>

int main(void){
-   printf("Hello World!\n");
+   printf("Hello World from my very first patch!\n");
    return 0;
}
```

Строки, начинающиеся с `-` удалятся из исходного кода и заменятся на строки, начинающихся с `+`.

4. Сохраним патч в файл:

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

5. Восстановим исходный код `cello.c`:

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

Мы сохраняем исходный файл `cello.c`, потому что при создании RPM используется исходный файл, а не измененный. Дополнительные сведения см. в разделе [Работа со СПЕС файлами](#).

Чтобы исправить `cello.c` с помощью `cello-output-first-patch.patch`, перенаправьте патч-файл `patch` командой:

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

Содержимое `cello.c` теперь отражает изменения:

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

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

Чтобы собрать и запустить отредактированную `cello.c`:

```
$ make clean
rm cello
```

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

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

Вы создали патч, отредактировали программу, собрали отредактированную программу и запустили её.

## Установка Произвольных Артефактов

Большим преимуществом [Linux](#) и других Unix-подобных систем является [Стандарт иерархии файловой системы](#). Он указывает, в каком каталоге должны быть расположены файлы. Файлы, установленные из пакетов RPM, должны быть размещены в соответствии с ИФС. Например, исполняемый файл должен находиться в каталоге, который находится в переменной [PATH](#).

В контексте этого руководства, *Произвольный артефакт* - это все, что устанавливается из RPM в систему. Для RPM и для системы это может быть скрипт, бинарный файл, скомпилированный из исходного кода пакета, предварительно скомпилированный бинарный файл или любой другой файл.

Мы рассмотрим два популярных способа размещения *произвольных артефактов* в системе: с помощью команды `install` и с помощью команды `make install`.

### Использование команды `install`

Иногда с помощью инструментов автоматизации сборки, таких как [GNU make](#) не является оптимальным - например, если упакованная программа проста. В этих случаях упаковщики часто используют команду `install` (предоставляемая системе [coreutils](#)), которая помещает артефакт в указанный каталог в файловой системе с указанным набором разрешений.

В приведенном ниже примере будет использоваться файл `bello`, который мы ранее создали в качестве произвольного артефакта, зависящего от нашего метода установки. Обратите внимание, что Вам либо понадобятся разрешения [sudo](#), либо запустите эту команду от имени root, исключая часть команды `sudo`.

В этом примере `install` помещает файл `bello` в `/usr/bin` с разрешениями, общими для исполняемых скриптов:

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

Теперь `bello` находится в каталоге, который указан в переменной `$PATH`. Таким образом, Вы можете запустить `bello` из любого каталога, не указывая его путь:

```
$ cd ~
```

```
$ bello
Hello World
```

## Использование команды `make install`

Популярным автоматизированным способом установки программного обеспечения в систему является использование команды `make install`. Вы указываете, как установить произвольные артефакты в систему в файле `Makefile`.

**ПРИМЕЧАНИЕ** | Обычно `Makefile` пишется разработчиком, а не упаковщиком.

Добавьте секцию `install` в `Makefile`:

`Makefile`

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

clean:
    rm cello

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

Переменная `$(DESTDIR)` является встроенной в `GNU make` и обычно используется для указания установки в каталог, отличный от корневого каталога.

Теперь вы можете использовать `Makefile` не только для сборки программного обеспечения, но и для его установки в систему.

Для сборки и установки программы `cello.c`:

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

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

Теперь `cello` находится в каталоге, который указан в переменной `$PATH`. Таким образом, Вы можете запустить `cello` из любого каталога, не указывая его полный путь.

```
$ cd ~

$ cello
```

Вы установили артефакт сборки в выбранное место в системе.

## Подготовка исходного кода для упаковки

### ПРИМЕЧАНИЕ

Код, созданный в этом разделе, можно найти [здесь](#).

Разработчики часто распространяют программное обеспечение в виде сжатых архивов исходного кода, которые затем используются для создания пакетов. В этом разделе Вы создадите такие архивы.

### ПРИМЕЧАНИЕ

Создание архивов исходного кода обычно выполняется не RPM-упаковщиком, а разработчиком. Упаковщик работает с готовым архивом исходного кода.

Программное обеспечение должно распространяться с [лицензией](#). Для примера мы будем использовать лицензию [GPLv3](#). Текст лицензии помещается в файл **LICENSE** для каждой из примеров программ. Упаковщику RPM необходимо иметь дело с файлами лицензий при упаковке.

## Создание Tarball с исходным кодом

В приведенных ниже примерах мы помещаем каждую из трех программ **Hello World** в архив, сжатый с помощью [gzip](#). Программное обеспечение часто выпускается таким образом, чтобы позже быть упакованным для распространения.

### bello

Проект *bello* реализует **Hello World** в [bash](#). Реализация содержит только сценарий оболочки **bello**, поэтому результирующий архив **tar.gz** будет содержать только один файл, кроме файла **LICENSE**. Давайте предположим, что это версия программы - **0.1**

Подготовьте проект *bello* для распространения:

1. Поместите файлы в один каталог:

```
$ mkdir /tmp/bello-0.1  
  
$ mv ~/bello /tmp/bello-0.1/  
  
$ cp /tmp/LICENSE /tmp/bello-0.1/
```

2. Создайте архив и переместите его в **~/rpmbuild/SOURCES/**:

```
$ cd /tmp/
```

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

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

## pello

Проект *pello* реализует **Hello World** на **Python**. Реализация содержит только программу **pello.py**, так что результирующий архив **tar.gz** будет содержать только один файл, кроме файла **LICENSE**. Предположим, что это версия программы - **0.1.1**

Подготовьте проект *pello* для распространения:

1. Поместите файлы в один каталог:

```
$ mkdir /tmp/pello-0.1.1

$ mv ~/pello.py /tmp/pello-0.1.1/

$ cp /tmp/LICENSE /tmp/pello-0.1.1/
```

2. Создайте архив для распространения и переместите его в **~/rpmbuild/SOURCES/**:

```
$ cd /tmp/

$ tar -cvzf pello-0.1.1.tar.gz pello-0.1.1
pello-0.1.1/
pello-0.1.1/LICENSE
pello-0.1.1/pello.py

$ mv /tmp/pello-0.1.1.tar.gz ~/rpmbuild/SOURCES/
```

## cello

Проект *cello* реализует **Hello World** на **C**. Реализация содержит только файлы **cello.c** и **Makefile**, поэтому результирующий архив **tar.gz** будет содержать только два файла, помимо файла **LICENSE**. Давайте предположим, что это версия программы - **1.0**

Обратите внимание, что **patch** не распространяется в архиве вместе с программой. Упаковщик RPM применяет исправление при создании RPM. Патч будет помещен в каталог **~/rpmbuild/SOURCES/** рядом с **.tar.gz** архивом.

Подготовьте проект *cello* для распространения:

1. Поместите файлы в один каталог:

```
$ mkdir /tmp/cello-1.0  
  
$ mv ~/cello.c /tmp/cello-1.0/  
  
$ mv ~/Makefile /tmp/cello-1.0/  
  
$ cp /tmp/LICENSE /tmp/cello-1.0/
```

2. Создайте архив для распространения и переместите его в `~/rpmbuild/SOURCES/`:

```
$ cd /tmp/  
  
$ tar -cvzf cello-1.0.tar.gz cello-1.0  
cello-1.0/  
cello-1.0/Makefile  
cello-1.0/cello.c  
cello-1.0/LICENSE  
  
$ mv /tmp/cello-1.0.tar.gz ~/rpmbuild/SOURCES/
```

3. Добавьте патч:

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

Теперь исходный код готов к упаковке в RPM.

# 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 spec 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 предопределённые 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%_sysconffdir/xdg/systemd/user/
cp script_dbus.timer script_dbus.service \
    %buildroot%_sysconffdir/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 находится 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 .спес 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.спес - файл спецификации для сборки пакета

Остановимся подробнее на этих двух файлах.

Перейдём к содержанию файла **rules**

```
tar: .  
спес: .gear/notify.спес
```

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 .спес 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%_sysconfdir/xdg/systemd/user/`. В пути использован макрос `%_sysconfdir`, который заменяется путём `/etc`. Следующая строка `cp %name.timer %name.service %buildroot%_sysconfdir/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
```

## Программное обеспечение для исправления ошибок

**Patch** - это исходный код, который исправляет другой исходный код. Он отформатирован как *diff*, потому что представляет разницу между двумя версиями текста. Разница создаётся с помощью утилиты **diff**, которая затем применяется к исходному коду с помощью утилиты **patch**.

### ПРИМЕЧАНИЕ

Разработчики программного обеспечения часто используют системы контроля версий, такие как **git**, для управления своей кодовой базой. Такие инструменты предоставляют свои собственные методы создания различий или исправления программного обеспечения.

В следующем примере мы создаем исправление из исходного кода с помощью **diff**, а затем используем **patch**. Исправление использует в следующих разделах при создании RPM и работе со `.spec`-файлом.

## Система контроля версий

При работе с RPMs желательно использовать **Системы контроля версий** (VCS), такую как **git**, для управления компонентами программного обеспечения, которое мы упаковываем. Следует отметить, что хранение бинарных файлов в системе контроля версий нецелесообразно, поскольку это резко увеличивает размер исходного репозитория, поскольку эти инструменты разработаны для обработки различий в файлах (часто оптимизированных для текстовых файлов), чему не поддаются бинарные файлы, поэтому обычно сохраняется весь бинарный файл целиком. Существуют некоторые утилиты, популярные среди upstream проектов с открытым исходным кодом, которые решают эту проблему, либо сохраняя SPEC файл, где исходный код находится в VCS (т. е. - он не находится в сжатом архиве для распространения), либо помещая в VCS только SPEC-файл и патчи и загружается сжатый архив upstream исходного кода в так называемый «кэш просмотра».



В этом разделе мы рассмотрим два различных варианта использования системы контроля версий `git` для управления содержимым, которое в конечном итоге будет преобразовано в пакет RPM. Первый называется `tito`, второй - `dist-git`.

#### ПРИМЕЧАНИЕ

Вам нужно будет установить пакет `git` в Вашу систему, он понадобится нам для изучения данного раздела.

# Дополнительные материалы

В этой главе рассматриваются темы, которые выходят за рамки вводного руководства, но часто полезны в реальной упаковке RPM.

## Подпись пакетов

Подпись пакета - это способ защитить пакет для конечного пользователя. Безопасная транспортировка может быть достигнута с помощью реализации протокола HTTPS. Такой метод используют, когда пакет загружается непосредственно перед установкой. Однако пакеты часто загружаются заранее и хранятся в локальных репозиториях перед их использованием. Пакеты подписываются, чтобы гарантировать, что никакая третья сторона не сможет изменить содержимое пакета.

Существует три способа подписи пакета:

- [Добавление подписи к уже существующему пакету.](#)
- [Замена подписи на уже существующем пакете.](#)
- [Подпись пакета во время сборки.](#)

### Добавление подписи к пакету

В большинстве случаев пакеты создаются без подписи. Подпись добавляется непосредственно перед выпуском пакета.

Чтобы добавить другую подпись к пакету, используйте опцию `--addsign`. Наличие более чем одной подписи позволяет зафиксировать путь владения пакетом от разработчика пакета до конечного пользователя.

В качестве примера подразделение компании создает пакет и подписывает его ключом подразделения. Затем штаб-квартира компании проверяет подпись пакета и добавляет корпоративную подпись к пакету, заявляя, что подписанный пакет является подлинным.

С двумя подписями пакет попадает к продавцу. Продавец проверяет подписи и, если они проверяются, также добавляет свою подпись.

Теперь пакет отправляется в компанию, которая желает его развернуть. Проверив каждую подпись на упаковке, они знают, что это подлинная копия, не изменившаяся с момента её первого создания. В зависимости от внутреннего контроля внедряющей компании, они могут добавить свою собственную подпись, чтобы заверить своих сотрудников в том, что пакет получил их корпоративное одобрение.

Вывод из команды `--addsign`:

```
$ rpm --addsign blather-7.9-1.i386.rpm
Enter pass phrase:

Pass phrase is good.
```

```
blather-7.9-1.i386.rpm:
```

Для проверки подписей пакета с несколькими подписями:

```
$ rpm --checksig blather-7.9-1.i386.rpm
blather-7.9-1.i386.rpm: size pgp pgp md5 OK
```

Два обозначения **pgp** в выходных данных команды **rpm --checksig** показывают, что пакет был подписан дважды.

RPM позволяет добавлять одну и ту же подпись несколько раз. Параметр **--addsign** не проверяет наличие нескольких идентичных подписей.

```
$ rpm --addsig blather-7.9-1.i386.rpm
Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
$ rpm --addsig blather-7.9-1.i386.rpm
Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
$ rpm --addsig blather-7.9-1.i386.rpm
Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
$ rpm --checksig blather-7.9-1.i386.rpm
blather-7.9-1.i386.rpm: size pgp pgp pgp md5 OK
```

На выходе команды **rpm --checksig** отображается четыре подписи.

## Замена подписи пакета

Чтобы изменить открытый ключ без необходимости пересобирать каждый пакет, используйте опцию **--resign**.

```
$ rpm --resign blather-7.9-1.i386.rpm
Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
```

Использование опции **--resign** с несколькими пакетами:

```
$ rpm --resign b*.rpm
Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
bother-3.5-1.i386.rpm:
```

## Подпись во время сборки

Чтобы подписать пакет во время сборки, используйте команду `rpmbuild` с параметром `--sign`. Для этого необходимо ввести кодовую фразу PGP.

Для примера:

```
$ rpmbuild -ba --sign blather-7.9.spec
Enter pass phrase:

Pass phrase is good.
* Package: blather
...
Binary Packaging: blather-7.9-1
Finding dependencies...
...
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/blather-7.9-1.i386.rpm
...
Source Packaging: blather-7.9-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/blather-7.9-1.src.rpm
```

Сообщение "Generating signature" появляется как в бинарном, так и в исходном разделах упаковки. Число, следующее за сообщением, указывает на то, что добавленная подпись была создана с использованием PGP.

### ПРИМЕЧАНИЕ

При использовании опции `--sign` в `rpmbuild`, используйте только аргументы `-bb` или `-ba` для сборки пакета. Аргумент `-ba` обозначает сборку бинарных **и** исходных пакетов.

Чтобы проверить подпись пакета, используйте команду `--checksig`. Для примера:

```
$ rpm --checksig blather-7.9-1.i386.rpm
blather-7.9-1.i386.rpm: size pgp md5 OK
```

## Сборка нескольких пакетов

При сборке нескольких пакетов используйте следующий синтаксис, чтобы избежать многократного ввода кодовой фразы PGP. Например, при сборке пакетов **blather** и **bother**, подпишите их, следуя примеру ниже:

```
$ rpmbuild -ba --sign b*.spec
    Enter pass phrase:

Pass phrase is good.
* Package: blather
...
Binary Packaging: blather-7.9-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/blather-7.9-1.i386.rpm
...
Source Packaging: blather-7.9-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/blather-7.9-1.src.rpm
...
* Package: bother
...
Binary Packaging: bother-3.5-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/bother-3.5-1.i386.rpm
...
Source Packaging: bother-3.5-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/bother-3.5-1.src.rpm
```

## Подробнее о макросах

Существует множество встроенных макросов RPM, и мы рассмотрим некоторые из них в следующем разделе, однако исчерпывающий список можно найти на странице [RPM Official Documentation](#).

### Определение Ваших Собственных Макросов

Вы можете определить свои собственные макросы. Ниже приводится выдержка из [RPM Official Documentation](#), в которой содержится исчерпывающая информация о возможностях макросов.

Чтобы определить макрос, используйте:

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

Все пробелы, окружающие `\`, удаляются. Имя может состоять из буквенно-цифровых символов и символа `_`, и должно иметь длину не менее 3 символов. Макрос без поля `(opts)` является “простым” в том смысле, что выполняется только рекурсивное расширение макроса. Параметризованный макрос содержит поле `(opts)` field. The `opts` - (строка в круглых скобках) передается точно так же, как и в `getopt(3)` для обработки `argc/argv` в начале вызова макроса.

#### ПРИМЕЧАНИЕ

Более старый SPEC файлы RPM могут использовать шаблон макроса `%define <name> <body>`. Различия между макросами `%define` и `%global` заключаются в следующем:

- `%define` имеет локальную область действия, что означает, что он применяется только к указанной части SPEC файла. Кроме того, тело макроса `%define` расширяется при использовании.
- `%global` имеет глобальную область действия, что означает, что он применяется ко всему SPEC файлу. Кроме того, тело макроса `%global` расширяется во время определения.

Пример:

```
%global githash 0ec4e58
%global python_sitelib %(%{__python} -c "from distutils.sysconfig import
get_python_lib; print(get_python_lib())")
```

#### ПРИМЕЧАНИЕ

Макросы всегда оцениваются, даже в комментариях. Иногда это небезопасно. Но во втором примере мы выполняем команду `python`, чтобы получить содержимое макроса. Эта команда будет выполняться даже тогда, когда Вы прокомментируете макрос, или когда Вы вводите имя макроса в `%changelog`. Чтобы закомментировать макрос, используйте `%%`. Например: `%%global`.

## %setup

Макрос `%setup` можно использовать для сборки пакета с помощью `tarball`. Стандартное поведение макроса `%setup` можно увидеть в выходных данных `rpmbuild`. В начале каждой фазы макрос выводит `Executing(%something)`. Например:

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

Выходные данные оболочки устанавливаются с включенным `set -x`. Чтобы просмотреть содержимое `/var/tmp/rpm-tmp.DhddsG`, используйте опцию `--debug`, поскольку `rpmbuild` удаляет временные файлы после успешной сборки. Здесь отображается настройка переменных

среды, например:

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

Макрос `%setup` гарантирует, что мы работаем в правильном каталоге, удаляет остатки предыдущих сборок, распаковывает исходный архив и устанавливает некоторые привилегии по умолчанию. Существует несколько вариантов настройки поведения макроса `%setup`.

### `%setup -q`

Параметр `-q` ограничивает детализацию макроса `%setup`. Вместо `tar -xof` выполняется только `tar -xvvo`. Этот параметр должен быть использован в качестве первого.

### `%setup -n`

В некоторых случаях каталог из расширенного архива имеет другое имя, чем ожидалось `%{name}-%{version}`. Это может привести к ошибке макроса `%setup`. Имя каталога должно быть указано параметром `-n directory_name`.

Например, если имя пакета `cello`, но исходный код заархивирован в `hello-1.0.tgz` и содержит каталог `hello/`, содержимое SPEC файла должно быть следующим:

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

### `%setup -c`

Параметр `-c` можно использовать, если архив исходного кода не содержит никаких подкаталогов и после распаковки файлы из архива заполняют текущий каталог. Опция `-c` создает каталог и переходит к расширению архива. Наглядный пример:

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

Каталог не изменяется после расширения архива.

## %setup -D и -T

Параметр **-D** отключает удаление каталога исходного кода. Этот параметр полезен, если макрос **%setup** используется несколько раз. По сути, параметр **-D** означает, что следующие строки не используются:

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

Параметр **-T** отключает расширение хранилища исходного кода, удаляя следующую строку из скрипта:

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

## %setup -a и -b

Параметры **-a** и **-b** расширяют определённые источники.

- Параметр **-b** (расшифровывается как **before**) расширяет определенные источники перед входом в рабочий каталог.
- Параметр **-a** (расшифровывается как **after**) расширяет эти источники после входа. Их аргументами являются исходные номера из преамбулы файла спецификации.

Например, допустим, что архив **cello-1.0.tar.gz** содержит пустой каталог **examples**, и примеры поставляются в отдельных **examples.tar.gz** tarball архивах, и они разархивируются в каталог с тем же именем. В этом случае используйте **-a 1**, так как мы хотим разархивировать **Source1** после входа в рабочий каталог:

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

Но если бы примеры были в отдельном **cello-1.0-examples.tar.gz** tarball архиве, который расширяется до **cello-1.0/examples**, используйте параметры **-b 1**, поскольку **Source1** должен быть разархивирован перед входом в рабочий каталог:

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

Вы также можете использовать комбинацию всех этих опций.



## %files

Общие “расширенные” макросы RPM, необходимые в разделе **%files**:

Макрос	Описание
<b>%license</b>	Макрос идентифицирует файл, указанный в списке, как файл ЛИЦЕНЗИИ, и он будет установлен и помечен как таковой RPM. Пример: <b>%license LICENSE</b>
<b>%doc</b>	Этот макрос идентифицирует файл, указанный как документация, и он будет установлен и помечен RPM как таковой. Это часто используется не только для документации об упаковываемом программном обеспечении, но и для примеров кода и различных элементов, которые должны сопровождать документацию. Пример: <b>%doc README</b>
<b>%dir</b>	Макрос указывает, что путь является каталогом, которым должен владеть этот RPM. Это важно, чтобы манифест RPM-файла точно знал, какие каталоги очищать при удалении. Пример: <b>%dir %{_libdir}/%{name}</b>
<b>%config</b>	Указывает, что следующий файл является файлом конфигурации и поэтому не должен перезаписываться (или заменяться) при установке или обновлении пакета, если файл был изменен по сравнению с исходной контрольной установкой. В случае внесения изменений файл будет создан с добавлением <b>.rpmnew</b> в конец имени файла при обновлении или установке, чтобы ранее существующий или измененный файл в целевой системе не был изменен. Пример: <b>%config(noreplace) %{_sysconfdir}/%{name}/%{name}.conf</b>

## Встроенные макросы

В Вашей системе есть много встроенных макросов RPM, и самый быстрый способ просмотреть их все - это просто выполнить команду **rpm --showrc**. Обратите внимание, что это будет содержать много выходных данных, поэтому его часто используют в сочетании с каналом для **grep**.

Вы также можете найти информацию о макросах RPM, которые поставляются непосредственно с версией RPM Вашей системы, просмотрев выходные данные **rpm -ql rpm**, обратив внимание на файлы с названием **macros** в структуре каталогов.

## RPM Макросы, предоставляемые дистрибутивом

Различные дистрибутивы будут предоставлять разные наборы рекомендуемых макросов RPM в зависимости от языковой реализации упаковываемого программного обеспечения или конкретных рекомендаций рассматриваемого дистрибутива. [Предопределённые макросы](#)

## Пользовательские макросы

Вы можете переопределить макросы в файле **~/rpmmacros**. Любые внесенные вами изменения повлияют на каждую сборку на Вашем компьютере.

Существует несколько макросов, которые Вы можете использовать для переопределения

**%\_topdir /opt/some/working/directory/rpmbuild**

Вы можете создать этот каталог, включая все подкаталоги, с помощью утилиты **rpmdev-setuptree**. Значение этого макроса по умолчанию равно **~/rpmbuild**.

**%\_smp\_mflags -l3**

Этот макрос часто используется для передачи в Makefile, например: **make %{\_smp\_mflags}**, и для задания количества одновременных процессов на этапе сборки. По умолчанию для него задано значение **-jX**, где X - количество ядер. Если Вы измените количество ядер, Вы можете ускорить или замедлить сборку пакетов.

Хотя Вы можете определить любые новые макросы в файле **~/rpmmacros** это не рекомендуется, поскольку эти макросы не будут присутствовать на других компьютерах, где пользователи могут захотеть попытаться пересобрать Ваш пакет.

## Epoch, Скриптлеты и Триггеры

В мире SPEC файлов RPM существуют различные разделы, которые считаются продвинутыми, поскольку они влияют не только на файл спецификации, способ сборки пакета, но и на конечный компьютер, на который устанавливается результирующий RPM. В этом разделе мы рассмотрим наиболее распространенные из них, такие как Epoch, Скриптлеты и триггеры.

### Epoch

Первым в списке стоит **Epoch**, epoch - это способ определения взвешенных зависимостей на основе номеров версий. Его значение по умолчанию равно 0, если директива **Epoch** не указана в SPEC файле. Это не рассматривалось в разделе "SPEC файл" этого руководства, потому что почти всегда вводить значени Epoch - плохая идея, поскольку это искажает то, что вы обычно ожидаете от RPM при сравнении версий пакетов.

Например, если был установлен пакет **foobar** с **Epoch: 1** и **Version: 1.0**, а кто-то другой упаковал **foobar** с **Version: 2.0**, но просто опустил директиву **Epoch** либо потому, что они не знали о её необходимости, либо просто забыли, эта новая версия никогда не будет считаться обновлением, потому что версия Epoch превалирует над традиционным маркером Name-Version-Release, который означает управление версиями для RPM-пакетов.

Этот подход обычно используется только в случае крайней необходимости для решения проблемы с порядком обновления, которая может возникнуть как побочный эффект программного обеспечения из upstream репозитория, изменяющего схемы нумерации версий или версии, включающие буквенные символы, которые не всегда можно надёжно сравнить на основе кодирование.

### Скриптлеты и Триггеры

В пакетах RPM существует ряд директив, которые можно использовать для внесения необходимых или желаемых изменений в систему во время установки RPM. Они

называются **скриптлеты**.

Один из основных примеров того, когда и почему Вы хотели бы это сделать, это когда установлена системная служба RPM и она предоставляет [systemd файл](#). Во время установки нам нужно будет уведомить [systemd](#) о появлении нового модуля, чтобы системный администратор мог выполнить команду, аналогичную `systemctl start foo.service` после установки вымышленного `foo` (который в этом примере предоставляет демон). Аналогично, нам нужно было бы отменить это действие при деинсталляции, чтобы администратор не получал ошибок из-за того, что бинарный файл демона больше не установлен, но файл модуля все еще существует в запущенной конфигурации systemd.

Существует небольшая горстка распространенных скриптлетов, они похожи на “заголовки разделов”, такие как `%build` or `%install`, в том смысле, что они определяются многострочными сегментами кода, часто написанными, как стандартный сценарий оболочки [POSIX](#) , но могут быть на нескольких разных языках программирования. Исчерпывающий список этих доступных языков можно найти в *Официальной документации RPM*.

Следующие скриптлеты:

Директива	Описание
<code>%pre</code>	Скриптлет, который выполняется непосредственно перед установкой пакета в целевую систему.
<code>%post</code>	Скриптлет, который выполняется сразу после установки пакета в целевой системе.
<code>%preun</code>	Скриптлет, который выполняется непосредственно перед удалением пакета из целевой системы.
<code>%postun</code>	Скриптлет, который выполняется сразу после удаления пакета из целевой системы.

Также часто для этой функции существуют макросы RPM. В нашем предыдущем примере мы обсуждали необходимость получения [systemd](#) уведомления о новом [unit file](#), , это легко обрабатывается макросами скриптлетов systemd, как Мы можем видеть из приведенного ниже примера вывода. Более подробную информацию об этом можно найти в [Fedora systemd Packaging Guidelines](#).

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

```

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

$ rpm --eval %{systemd_post}

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

$ rpm --eval %{systemd_postun}

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

$ rpm --eval %{systemd_preun}

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

```

Еще один элемент, который обеспечивает еще более детальный контроль над транзакцией RPM в целом, - это то, что известно как **триггеры**. По сути, это то же самое, что и скриптлет, но выполняется в очень определенном порядке операций во время транзакции установки или обновления RPM, что позволяет более точно контролировать весь процесс.

Порядок, в котором выполняется каждый из них, и подробная информация о котором - приведена ниже.

```

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

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

```

```

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

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

old-%triggerpostun
any-%triggerpostun (%triggerpostun from other packages set off by old un
                    install)

...
all-%posttrans

```

Вышеуказанные элементы взяты из прилагаемой документации RPM, найденной в [/usr/share/doc/rpm/triggers](#) на системах Fedora и [/usr/share/doc/rpm-4.\\*/triggers](#) в системах RHEL 7 и CentOS 7.

### Использование скриптов без оболочки в SPEC файле

Параметр скриптлета `-p`, в SPEC файле позволяет вызывать определенный интерпретатор вместо стандартного `-p /bin/sh`. Наглядным примером является скрипт, который выводит сообщение после установки `pello.py`.

1. Откройте файл `pello.spec`.
2. Найдите следующую строку:

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

Под этой строкой вставьте следующий код:

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

3. Создайте свой пакет в соответствии с главой [Сборка RPM](#).
4. Установите Ваш пакет:

```
# dnf install /home/<username>/rpmbuild/RPMS/noarch/pello-0.1.1-1.fc27.noarch.rpm
```

Результатом выполнения этой команды является следующее сообщение после установки:

```

Installing      : pello-0.1.1-1.fc27.noarch                1/1
Running scriptlet: pello-0.1.1-1.fc27.noarch                1/1
This is python code

```

## ПРИМЕЧАНИЕ

- Чтобы использовать скрипт Python 3: Напишите строку `%post -p /usr/bin/python3` под строкой `install -m` in a SPEC file.
- Чтобы использовать скрипт Lua: Напишите строку `%post -p <lua>` под строкой `install -m` в SPEC файле.
- Таким образом, в SPEC файле может быть указан любой интерпретатор.

## Условные обозначения RPM

Условные обозначения RPM позволяют условно включать различные разделы SPEC файла.

Чаще всего условные обозначения имеют дело с:

- разделами, относящимися к конкретной архитектуре
- разделами, относящимися к конкретной операционной системе
- проблемами совместимости между различными версиями операционных систем
- существованием и определением макросов

### Синтаксис условий RPM

Если *expression* истинно, то выполните какое-нибудь действие:

```
%if expression
...
%endif
```

Если *expression* истинно, то выполните какое-нибудь действие, в другом случае выполните другое действие:

```
%if expression
...
%else
...
%endif
```

### Примеры условий RPM

#### Обозначение `%if`

```
%if 0%{?rhel} == 6
sed -i '/AS_FUNCTION_DESCRIBE/ s/^/#/' configure.in
sed -i '/AS_FUNCTION_DESCRIBE/ s/^/#/' acinclude.m4
%endif
```

Это условие обрабатывает совместимость между RHEL6 и другими операционными системами с точки зрения поддержки макроса AS\_FUNCTION\_DESCRIBE. Когда пакет создается для RHEL, определяется макрос `%rhel`, и он расширяется до версии RHEL. Если его значение равно 6, что означает, что пакет создан для RHEL 6, то ссылки на AS\_FUNCTION\_DESCRIBE, который не поддерживается RHEL6, удаляются из сценариев автоконфигурации.

```
%if 0%{?el6}
%global ruby_sitearch %(ruby -rrbconfig -e 'puts Config::CONFIG["sitearchdir"]')
%endif
```

Это условие регулирует совместимость между Fedora версии 17 и новее и RHEL 6 с точки зрения поддержки макроса `%ruby_sitearch` макро. Fedora версии 17 и никогда не определяет `%ruby_sitearch` по умолчанию, но RHEL6 не поддерживает этот макрос. Условие проверяет, является ли операционная система RHEL 6. Если это так, `%ruby_sitearch` определяется явно. Обратите внимание, что `0%{?el6}` имеет то же значение, что и `0%{?rhel} == 6` из предыдущего примера, и он проверяет, построен ли пакет на RHEL 6.

```
%if 0%{?fedora} >= 19
%global with_rubypick 1
%endif
```

Это условие обрабатывает поддержку инструмента выбора ruby. Если операционная система Fedora версии 19 или новее, поддерживается rubypick.

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

Это условие обрабатывает определение макросов. Если заданы макросы `%milestone` или `%revision`, переопределяется макрос `%ruby_archive`, который определяет имя вышестоящего файла архива.

### Специальные варианты обозначения `%if`

Условные обозначения `%ifarch`, `%ifnarch` и `%ifos` являются специализированными вариантами условных обозначений `%if`. Эти варианты обычно используются, поэтому у них есть свои собственные макросы.

#### Обозначение ``%ifarch``

Условие `%ifarch` используется для начала блока SPEC файла, который зависит от архитектуры. За ним следует один или несколько спецификаторов архитектуры, каждый из которых разделен запятыми или пробелами.

```
%ifarch i386 sparc
...
%endif
```

Все содержимое SPEC файла между **%ifarch** и **%endif** обрабатывается только на 32-разрядных архитектурах AMD и Intel или системах на базе Sun SPARC.

**Условное обозначение %ifnarch**

Условие **%ifnarch** имеет обратную логику, чем условие **%ifarch**.

```
%ifnarch alpha
...
%endif
```

Все содержимое SPEC файла между **%ifnarch** и **%endif** обрабатывается только в том случае, если оно не выполняется в системе на основе Digital Alpha/AXP.

**Условие %ifos**

Условие **%ifos** используется для управления обработкой на основе операционной системы сборки. За ним может следовать одно или несколько имен операционной системы.

```
%ifos linux
...
%endif
```

Все содержимое SPEC файла между **%ifos** и **%endif** обрабатывается только в том случае, если сборка была выполнена в системе Linux.



# Join

Если Вы считаете, что какого-то пакета в Сизифе не хватает, или какой-то пакет заслуживает большего внимания и готовы заняться им — значит, настало время присоединиться к команде [ALT Linux Team](#).

Процедура Join - это процесс вступления в ALT Linux Team, результатом которого является возможность непосредственно участвовать в разработке репозитория Сизиф. После прохождения Join Вы станете мейнтейнером.

Кто такой мейнтейнер? У каждого пакета в репозитории Сизиф есть один или несколько мейнтейнеров — это те участники ALT Linux Team, которые собирают новые версии пакета, следят за ошибками, которые заводят пользователи в багтрекере и общаются с upstream-разработчиками программы.

## Процедура принятия в Team

Для принятия человека в Team создаётся небольшая команда заинтересованных в этом тимовцев:

1. Секретарь команды. В его обязанность входит отслеживать стадии принятия и выполнять административные действия.
2. Ментор вступающего. Он помогает кандидату со вступлением, отвечает на его вопросы и принимает решение о готовности кандидата.
3. Рецензент работы вступающего. Он проводит независимую оценку готовности вступающего по результатам его работы и подтверждает его готовность.
4. Сам кандидат.

## Сбор информации

Для принятия в [ALT Linux Team](#) необходима следующая информация:

1. Имя ментора — участника команды, имеющего желание помогать в процессе приёма в Team. Менторов можно искать в [списках рассылки](#), канале [Telegram](#), или узнать у Ваших знакомых членов сообщества ALT Linux Team, если таковые есть, готовы ли они стать Вашим ментором.
2. Псевдоним участника (имя пользователя). Выбирается им самим. Имя должно начинаться с буквы, содержать только строчные латинские буквы и цифры, быть не короче трёх символов.

### ПРИМЕЧАНИЕ

Псевдоним — это фактически Ваше второе имя в команде. Так Вас будут называть в глаза и за глаза, по нему на Вас будут ссылаться. Поэтому псевдоним лучше выбирать короткий, запоминающийся и не отягощённый мусором. Например, [yoda](#) — удобный псевдоним, а [travellingwilburys1998](#) — неудобный. Список уже занятых имён можно посмотреть в пакете [alt-gpgkeys](#).

1. Адрес почты, на который будет производиться пересылка с адреса `псевдоним@altlinux.org`;
2. SSH-ключ (ED25519 или RSA >= 4096bit). Принимающему нужна публичная часть ключа. Этот ключ будет использоваться для SSH-доступа на ресурсы Sisyphus ([git.alt](https://git.altlinux.org) и другие);
3. GPG-ключ (RSA >= 4096bit). В ключе должны быть имя в формате "<First name> <Last name>" и uid вида `псевдоним@altlinux.org`. Принимающему нужна публичная часть ключа. Этот ключ будет использоваться для подписи пакетов и для удостоверения личности в почте.

Если у Вас ещё нет SSH- или GPG-ключа, прочтите статью [Работа с ключами разработчика](#).

## Создание заявки

Когда Вы нашли своего куратора процедуры join - ментора, Вам необходимо зафиксировать начало вашей работы. На этом этапе вам понадобится [BugZilla](#). В данном случае bugzilla выступает площадкой для процедуры вступления в ALT Linux Team. Здесь ведётся официальный диалог с участниками, Вы можете посмотреть прогресс других кандидатов, узнать, какие задачи они взяли. Введите в поиске [join](#) и увидите список, относящийся к этой тематике.

Для начала необходимо пройти регистрацию на [BugZilla](#).

После того, как Вы прошли регистрацию и нашли ментора, можно начинать регистрировать заявку. Регистрация заявки происходит следующим образом: на главной странице сайта ALT Bugzilla выберите "[Зарегистрировать ошибку](#)" или "[Новая ошибка](#)". Будущие участники ALT Linux Team регистрируются в разделе [Team Accounts: ALT Linux Team: присоединение](#). При наведении мыши будут видны подсказки к заполнению полей формы. Клик по любому из пунктов формы приведёт Вас на страницу [Поля ошибок](#) с описанием терминов. Выберите компонент *Join*. Если Вы посмотрите другие компоненты, то убедитесь, что они не так популярны. При выборе уровня важности задачи учитывайте, что майнтейнеры просматривают ошибки в том числе по этому фильтру. Также будет видно, к какому разделу относится ваша задача. (Важность определяете Вы сами, исходя из своих возможностей и целеполагания. Скорее, это параметр для Вас.) Поставим здесь средний приоритет **P3**. Платформа определяется автоматически. Здесь же Вы можете изменить значения, в том числе поставить all, и научиться собирать пакеты для любых платформ.

Отметим новое состояние нашей заявки.

Поля *Исполнитель* и *Приёмка* заполнятся автоматически. В пункте *Подписчики* укажем всех, кто заинтересован в решении этой задачи. В случае join - это ментор.

В стандартной форме bugzilla есть поле для заполнения адреса URL, с которым может быть связана ошибка. Так как мы регистрируем по сути наше участие в прохождении Join, отметьте страницу на вики [Join](#)

Поле Аннотация предназначено для краткого описания сути ошибки. Так как мы заводим эту ошибку в рамках вступления в ALT Linux Team, напомним join с указанием предполагаемой почты: `join user@`.

Пример комментария к открытой заявке на вступление в ALT Linux Team.

```
Псевдоним: sova
Почта: Valentin Sokolov <sova@altlinux.org>
Адрес пересылки почты: sokolovvaly.158@gmail.com
Имя ментора: Grigory Ustinov
Почта ментора: grenka@altlinux.org
```

Хочу научиться собирать пакеты.

На данном этапе необходимо прикрепить SSH и GPG ключи. О том, как их создать и прикрепить к заявке - читайте ниже.

## Создание SSH и GPG ключей

### SSH Key

Для того, чтобы создать SSH ключ, можно воспользоваться командой:

```
$ ssh-keygen -t ed25519
```

На вопрос о файле сохранения ключа можно ничего не отвечать, а просто нажать Enter, чтобы принять путь файла ключа по умолчанию. Далее будет задан вопрос о пароле к ключу. Укажите придуманный пароль для защиты ключа.

В каталоге `~/.ssh` появятся файлы:

- `id_ed25519` - секретный ключ
- `id_ed25519.pub` - публичный ключ

<b>ПРИМЕЧАНИЕ</b>	НИКОГДА И НИКОМУ НЕ ПЕРЕСЫЛАЙТЕ СЕКРЕТНЫЙ КЛЮЧ
-------------------	--

### GPG Key

Для того, чтобы создать GPG-ключ, можно воспользоваться командой:

```
$ gpg --gen-key
```

В процессе ответа на вопросы выберете тип ключа `RSA` и `RSA`, длина ключа должна быть не менее 4096 bit, поэтому впишем 4096. Срок действия ключа - без ограничения срока действия. Для идентификации Вашего ключа необходим ID пользователя. Программа создаст его из Вашего имени, комментария и адреса электронной почты в виде: `Baba Yaga (pensioner) yaga@deepforest.ru`.

Теперь экспортируем публичную часть ключа для отправки, используя команду:

```
$ gpg --armor --export псевдоним@altlinux.org
```

Для копирования публичного ключа в файл Вы можете использовать следующую команду:

```
$ gpg --armor --export псевдоним@altlinux.org >>public.key
```

Экспортировать приватную часть ключа можно с помощью команды:

```
$ gpg --armor --export-secret-keys псевдоним@altlinux.org >>private.key
```

#### ПРИМЕЧАНИЕ

**ВНИМАНИЕ** Никогда и ни при каких условиях никому не сообщайте и не присылайте Ваши ПРИВАТНЫЕ ключи. Сохраните Ваши приватные и публичные ключи на отдельном носителе для лучшей сохранности.

Для более подробного изучения темы создания и редактирования ключей рекомендуется перейти по ссылке: [Работа с ключами разработчика](#).

После того, как Вы создали ключи, загрузите их публичные части в комментарий к Вашей заявке. На странице заявки есть отдельный пункт **Приложить файл**.

Поздравляем! Вы создали заявку на вступление в ALT Linux Team. Её закрытие будет означать либо успешное прохождение процедуры и получение доступов к работе с Сизифом и вступление в Team, либо отказ в нём.

Следующие шаги по прохождению процедуры Join будут описаны позже, на данный момент документация находится на стадии разработки.

# 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 `nodelv` — 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 настройках файла `hasher .hasher/config` параметр

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

## Сборочные зависимости

### man hasher

Следующий абзац будет посвящён мануалу hasher, который доступен при выполнении следующей команды:

```
$ man hsh
```

Данный мануал переведён и адаптирован под русскоязычного пользователя.

ПРИМЕР КОМАНДЫ СБОРКИ ИЗ SRC.RPM

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