Рекомендации по сборке программных пакетов и написанию к ним документации на платформе Astra Linux

2011

СОДЕРЖАНИЕ

1. РАЗРАБОТКА И СБОРКА ПАКЕТОВ

1.1. Типы пакетов

Для создания и работы с пакетами исходных кодов в ОС «Astra Linux» (далее по тексту — ОС) используется утилита `dpkg-source`.

Существует несколько форматов хранения пакетов с исходными кодами — «`1.0`» (используется утилитой `dpkg-source` по умолчанию), «`2.0`» и «`3.0`» (последний имеет несколько разновидностей — далее речь пойдет только об одной из них, а именно «`3.0 native`»).

При разработке собственных пакетов для ОС рекомендуется использовать формат «`3.0 (native)`», который почти совпадает с форматом «`1.0`», но при этом лишен некоторых его недостатков и поддерживает различные методы сжатия. Кроме того, в отличие от «`1.0`» формат «`3.0 (native)`» по умолчанию игнорирует файлы и директории, относящиеся к системам контроля версий, а также множество временных файлов (см. опцию `-I` утилиты `dpkg-source`). Таким образом, становится удобно использовать систему контроля версий, а сборку и отладку, порождающую временные файлы, можно производить прямо в дереве исходных кодов.

Формат «`3.0 native`».подразумевает, что пакет исходных кодов представляет собой набор из двух файлов:

- `<название_пакета>_<версия>.tar.gz`
- `<название_пакета>_<версия>.dsc`

Например пакет исходных кодов для текстового редактора vim будет выглядеть так:

- `vim_7.1.314.tar.gz`
- `vim_7.1.314.dsc`

где `vim` — название пакета, `7.1.314` — авторская версия (версия upstream source). Файл с именем `*.tar.gz` содержит первичные авторские исходные коды (upstream source) и все специфичные исходные коды для дистрибутива. Файл `*.dsc` содержит описание пакета исходных кодов.

Пакеты, специально разработанные для дистрибутива, или собственные пакеты (native) «дебианизируются» на этапе разработки и не содержат дополнительных изменений, так как разработчик первичной версии пакета обычно является одновременно и его сопровождающим. Если необходимо внести изменения в исходные коды, то разработчик делает это прямо в первичных исходных кодах проекта, увеличивая при этом

основную версию пакета. Для этого используется файл вида:

<название_пакета>_<версия>.tar.gz.

По умолчанию утилита dpkg-source использует формат «1.0». Чтобы явно задать формат пакета с исходным кодом можно использовать три способа:

- поле Format в файле debian/control;
- опция командной строки --format;
- содержимое файла debian/source/format.

Способы задания формата пакета исходных кодов перечислены в порядке приоритета, т.е. сначала утилита dpkg-source попытается использовать первый способ, затем второй, и только затем третий способ. Для разработки пакетов ПО для ОС рекомендуется использовать файл debian/source/format. Для задания формата пакета можно воспользоваться командой:

```
#echo "3.0 (native)" > debian/source/format
```

1.2. Формирование пакетов

Для формирования пакета с исходными текстами необходимо создать в дереве с исходными текстами программы специальный каталог debian, в котором расположены сценарии сборки пакета. Для начального создания каталога debian с шаблонами сценариев сборки deb-пакета необходимо выполнить команду, находясь в каталоге с исходными текстами программы:

```
dh_make -s -e builder@build -f ../test-1.1.tar.gz
```

где test-1.1.tar.gz — gzip-архив с исходными текстами программы.

В дальнейшем необходимо изучить структуру каталога debian и согласно документу Debian Policy Manual (http://www.debian.org/doc/debian-policy/) заполнить необходимые параметры конфигурационных файлов. В поле Maintainer файла debian/control разработчик указывает свое имя и существующий адрес электронной почты.

При разработке пакетов необходимо обратить внимание на правильное описание build и runtime-зависимостей.

Пакеты исходного кода должны выполнять требования, касающиеся обязательных полей в файле debian/control и проверять полученный пакет с помощью утилиты lintian (lintian -c <имя deb-пакета или dsc-файла>). Недоработки, помеченные утилитой lintian как предупреждения «W», допускаются, а ошибки «E» крайне рекомендуется исправить.

1.3. Сборка пакетов

Сборку пакетов из пакетов с исходными текстами следует выполнять в ОС при помощи команды:

```
dpkg-buildpackage -rfakeroot
```

Сборка должна в обязательном порядке выполняться под учетной записью пользователя.

В дальнейшем пакет ПО должен быть протестирован на корректную установку с соблюдением зависимостей в ОС.

После проверки установки пакета необходимо провести комплексное тестирование функционала перед поставкой пакета.

1.4. Разработка пакета с драйверами

Разработка пакета с драйверами возможна двумя способами:

- если драйвер содержит закрытые исходные тексты, то необходимо собирать драйвер описанным выше способом для каждой версии ядра;
- если драйвер использует открытые исходные тексты, то рекомендуется использовать `module-assistant`, так как при использовании `module-assistant` отсутствует зависимость от конкретной версии ядра.

Далее в документе с примерами описывается использование подсистемы `module-assistant`.

Для использования компонента `module-assistant` необходимо выполнить следующие подготовительные действия:

- создать сценарий сборки модуля ядра (`Makefile`);
- создать пакет с исходным текстом модуля ядра, сценарием сборки модуля ядра и сценарием сборки пакета;
- собрать пакет с исходным текстом ядра и сценарием сборки и установить его в систему;
- собрать пакет с исполняемым модулем ядра при помощи `module-assistant`.

Пример сценария сборки (`Makefile`) модуля ядра представлен в приложении 1.

Для создания пакета с исходными текстами необходимо выполнить операции, описанные выше, или выполнить его создание вручную. Служебные файлы будут располагаться в каталоге `debian` внутри каталога с исходными текстами модуля ядра. Файлом сценария сборки пакета будет являться файл `rules`. Пример файла см. в приложении 2. Для того чтобы `module-assistant` имел возможность собрать

исполняемый модуль, в файл `rules` необходимо добавить несколько определений и целей, в заголовке файла должны быть подключены модули `module-assistant` и определены некоторые переменные:

```
PACKAGE=simple-modules
MA_DIR ?= /usr/share/modass
-include $(MA_DIR)/include/generic.make
-include $(MA_DIR)/include/common-rules.make
```

Также должны быть определены цели, необходимые для работы `module-assistant` (цель, в которой выполняется сборка пакета — `binary-modules`):

```
kdist_config: prep-deb-files
kdist_clean: clean
    $(MAKE) $(MFLAGS) -f debian/rules clean
    rm -f *.o *.ko
binary-modules:
    dh_testroot
    dh_clean -k
    dh_installdirs lib/modules/$(KVERS)/misc

    $(MAKE) KERNEL_DIR=$(KSRC) KVERS=$(KVERS)

    install -m 0644 simple.$ko debian/$(PKGNAME)/lib/modules/$
(KVERS)/misc

    dh_installdocs
    dh_installchangelogs
    dh_compress
    dh_fixperms
    dh_installdeb
    dh_gencontrol -- -v$(VERSION)
    dh_md5sums
    dh_builddeb --destdir=$(DEB_DESTDIR)
    dh_clean -k
```

Для сборки пакета необходимо наличие файла `control` — файла описания пакета. Пример файла представлен в приложении 3.

Также необходимо создать файл `control.modules.in`. Данный файл необходим

для последующей сборки пакета с исполняемым кодом модуля ядра. Пример файла см. в приложении 4.

После создания всех необходимых служебных файлов выполнить сборку пакета с исходными текстами модуля ядра с помощью команды:

```
dpkg-buildpackage -rfakeroot
```

находясь в каталоге с исходными текстами. По окончании работы программы `dpkg-buildpackage` в вышележащем каталоге появится пакет с исходными текстами ядра. Установить полученный пакет при помощи команды:

```
dpkg -i <имя_пакета>
```

Собрать пакет с исполняемым модулем ядра, выполнив команду:

```
m-a build simple
```

где `simple` — название собираемого модуля. Пакет с исполняемым модулем ядра будет располагаться в каталоге `/usr/src/`.

Исходный текст тестового модуля см. в приложении 5.

Приложение 1 (Makefile)

```
obj-m := simple.o
mksm_tpm-objs := simple.o

CTAGS_FLAGS := -R
PACKAGE     := $(shell basename $(PWD))

KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD       := $(shell pwd)

all: modules

modules:
      $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

tags: $(wildcard *.c) $(wildcard *.h)
      ctags $(CTAGS_FLAGS) $^

clean:
      -@[ -n "`which dh_clean`" -a -d debian ] && { echo Running dh_clean;
dh_clean; }
      rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
      rm -rf Module.markers Module.symvers modules.order

distclean: clean
      rm -rf tags

dist: Makefile $(wildcard *.c) $(wildcard *.h) debian
      @echo "Creating distributive source package $(PACKAGE).tar.gz"
      @cd ..; tar --exclude=".*" -cvzf $(PACKAGE).tar.gz $(addprefix $(PACKAGE)/,$^)

export: Makefile $(wildcard *.c) $(wildcard *.h) debian
      @echo "Creating distributive source package $(PACKAGE)_$(shell date +%Y%m
%d).tar.gz"
      @cd ..; tar --exclude=".*" -cvzf $(PACKAGE)_$(shell date +%Y%m%d).tar.gz $
(addprefix $(PACKAGE)/,$^)

install: modules
      install -m 0755 -d $(DESTDIR)/lib/modules/$(shell uname -r)/misc
      install -m 0644 $(obj-m:.o=.ko) $(DESTDIR)/lib/modules/$(shell uname -r)/misc

deb:  clean
      dpkg-buildpackage -us -uc -rfakeroot -I".*" -Itags

.PHONY: clean distclean dist export install deb
```

Приложение 2 (rules)

```
#!/usr/bin/make -f
# -*- makefile -*-
# Sample debian/rules that uses debhelper.
# This file was originally written by Joey Hess and Craig Small.
# As a special exception, when this file is copied by dh-make into a
# dh-make output file, you may use that output file without restriction.
# This special exception was added by Craig Small in version 0.37 of dh-make.
#
# This version is for a hypothetical package that can build a kernel modules
# architecture-dependant package via make-kpkg, as well as an
# architecture-independent module source package, and other packages
# either dep/indep for things like common files or userspace components
# needed for the kernel modules.
```

```
# Uncomment this to turn on verbose mode.
#export DH_VERBOSE=1

# some default definitions, important!
#
# Name of the source package
psource:=simple-source

# The short upstream name, used for the module source directory
sname:=simple

### KERNEL SETUP
### Setup the stuff needed for making kernel module packages
### taken from /usr/share/kernel-package/sample.module.rules

# prefix of the target package name
PACKAGE=simple-modules
# modifieable for experiments or debugging m-a
MA_DIR ?= /usr/share/modass
# load generic variable handling
-include $(MA_DIR)/include/generic.make
# load default rules, including kdist, kdist_image, ...
-include $(MA_DIR)/include/common-rules.make

# module assistant calculates all needed things for us and sets
# following variables:
# KSRC (kernel source directory), KVERS (kernel version string), KDREV
# (revision of the Debian kernel-image package), CC (the correct
# compiler), VERSION (the final package version string), PKGNAME (full
# package name with KVERS included), DEB_DESTDIR (path to store DEBs)

# The kdist_config target is called by make-kpkg modules_config and
# by kdist* rules by dependency. It should configure the module so it is
# ready for compilation (mostly useful for calling configure).
# prep-deb-files from module-assistant creates the neccessary debian/ files
kdist_config: prep-deb-files

# the kdist_clean target is called by make-kpkg modules_clean and from
# kdist* rules. It is responsible for cleaning up any changes that have
# been made by the other kdist_commands (except for the .deb files created)
kdist_clean: clean
	$(MAKE) $(MFLAGS) -f debian/rules clean
	rm -f *.o *.ko
#
### end  KERNEL SETUP

configure: configure-stamp
configure-stamp:
	dh_testdir
	touch configure-stamp


build-arch: configure-stamp  build-arch-stamp
build-arch-stamp:
	dh_testdir

	$(MAKE)

	touch $@

k = $(shell echo $(KVERS) | grep -q ^2.6 && echo k)
```

```
# the binary-modules rule is invoked by module-assistant while processing the
# kdist* targets. It is called by module-assistant or make-kpkg and *not*
# during a normal build
binary-modules:
	dh_testroot
	dh_clean -k
	dh_installdirs lib/modules/$(KVERS)/misc

	$(MAKE) KERNEL_DIR=$(KSRC) KVERS=$(KVERS)

	install -m 0644 simple.$ko debian/$(PKGNAME)/lib/modules/$(KVERS)/misc

	dh_installdocs
	dh_installchangelogs
	dh_compress
	dh_fixperms
	dh_installdeb
	dh_gencontrol -- -v$(VERSION)
	dh_md5sums
	dh_builddeb --destdir=$(DEB_DESTDIR)
	dh_clean -k

build-indep:  configure-stamp build-indep-stamp
build-indep-stamp:
	dh_testdir

	touch $@

build: build-arch build-indep

clean:
	#dh_testdir
	#dh_testroot
	rm -f build-arch-stamp build-indep-stamp configure-stamp

# Add here commands to clean up after the build process.
	$(MAKE) clean

	dh_clean

install: DH_OPTIONS=
install: build
	dh_testdir
	dh_testroot
	dh_clean -k
	dh_installdirs

	dh_installdirs -p$(psource)              \
		usr/src/modules/$(sname)/debian

	cp *.c debian/$(psource)/usr/src/modules/$(sname)
	cp Makefile debian/$(psource)/usr/src/modules/$(sname)
	cp debian/*modules.in* \
		debian/$(psource)/usr/src/modules/$(sname)/debian
	echo "This is dummy file. It contents will be lost." > debian/$
(psource)/usr/src/modules/$(sname)/debian/control
	cp debian/rules debian/changelog debian/copyright \
		debian/compat debian/$(psource)/usr/src/modules/$(sname)/debian/
	cd debian/$(psource)/usr/src && tar c modules | bzip2 -9 > $(sname).tar.bz2 &&
rm -rf modules

	dh_install
```

```
# Build architecture-independent files here.
# Pass -i to all debhelper commands in this target to reduce clutter.
binary-indep: build install
      dh_testdir -i
      dh_testroot -i
      dh_installchangelogs  -i
      dh_installdocs -i
      dh_installexamples -i
      dh_installman -i
      dh_link -i
      dh_compress -i
      dh_fixperms -i
      dh_installdeb -i

      dh_installdeb -i
      dh_shlibdeps -i
      dh_gencontrol -i
      dh_md5sums -i
      dh_builddeb -i

# Build architecture-dependent files here.
binary-arch: build install
      dh_testdir -s
      dh_testroot -s
      dh_installdocs -s
      dh_installexamples -s
      dh_installmenu -s
      dh_installcron -s
#     dh_installman -s
      dh_installinfo -s
      dh_installchangelogs  -s
      dh_strip -s
      dh_link -s
      dh_compress -s
      dh_fixperms -s
#     dh_makeshlibs -s
      dh_installdeb -s
#     dh_perl -s
      dh_shlibdeps -s
      dh_gencontrol -s
      dh_md5sums -s
      dh_builddeb -s

binary: binary-indep # binary-arch
.PHONY: build clean binary-indep binary-arch binary install configure binary-modules
kdist kdist_configure kdist_image kdist_clean
```

Приложение 3 (control)

```
Source: simple
Section: admin
Priority: extra
Maintainer: builder <builder@rusbitech.ru>
Build-Depends: debhelper (>= 7), bzip2, coreutils, sed, make
Standards-Version: 3.7.3

Package: simple-source
Architecture: all
Depends: module-assistant, debhelper (>= 7), make, bzip2, coreutils, sed
Description: Source for test driver
```

Приложение 4 (control.modules.in)

```
Source: simple
Section: admin
Priority: optional
Maintainer: builder <builder@rusbitech.ru>
Build-Depends: debhelper (>= 7), gcc, linux-headers-_KVERS_, coreutils, sed, make
Standards-Version: 3.7.3

Package: simple-modules-_KVERS_
Architecture: any
Depends: linux-image-_KVERS_, module-init-tools
Provides: simple-modules
Description: Simple test driver
```

Приложение 5 (simple.c)

```
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
        printk("Test module hello.Run.\n");
        return 0;
}

void cleanup_module(void)
{
        printk(KERN_ALERT "Test module hello.Exit.\n");
```

2. РАЗРАБОТКА ДОКУМЕНТАЦИИ

2.1. Написание страниц руководства man

Для написания страниц руководства `man` к разработанным программам можно воспользоваться следующей справочной информацией:

http://www.linuxcenter.ru/lib/articles/programming/man_page_minihowto.phtml

http://www.linux.org.ru/books/HOWTO/Man-Page.html

http://security.opennet.ru/man.shtml?topic=man&category=7&russian

2.2. Документы для сертификации

Документы, необходимые для проведения сертификации:

1) описание программы (ГОСТ 19.402-78), содержащее основные сведения о составе, логической структуре и среде функционирования ПО, а также описание методов, приемов и правил эксплуатации средств технологического оснащения при создании ПО;

2) описание применения (ГОСТ 19.502-78), содержащее сведения о назначении ПО, области применения, применяемых методах, классе решаемых задач, ограничениях при применении, минимальной конфигурации технических средств, среде функционирования и порядке работы;

3) пояснительная записка (ГОСТ 19.404-79), содержащая основные сведения о назначении компонентов, входящих в состав ПО, параметрах обрабатываемых наборов данных (подсхемах баз данных), формируемых кодах возврата, описание используемых переменных, алгоритмов функционирования и т. п.;

4) исходные тексты программ (ГОСТ 19.401-78), входящих в состав ПО.

Контроль состава документации проводится путем сравнения перечня представленных документов с требованиями РД. При этом проверяется наличие в представленных документах обязательной (в соответствии с РД) информации, необходимой для проведения испытаний.

На основании проведенного контроля, при успешном прохождении пунктов проверок содержания и состава документации, описанных выше, делается вывод о соответствии документации требованиям РД.

Результатом контроля является отчет о соответствии нормативным документам приведенной программной документации.

Примеры отчетов представлены в таблицах 1—3.

Пример отчета о контроле содержания описания программы (ГОСТ 19.402-78).

Таблица 1

№ п/п	Требования	Результаты контроля
1.	В разделе «Общие сведения» должны быть указаны: – обозначение и наименование программы; – программное обеспечение, необходимое для функционирования программы	
2.	В разделе «Функциональное назначение» должны быть указаны классы решаемых задач и (или) назначение программы и сведения о функциональных ограничениях на применение	
3.	В разделе «Описание логической структуры» должны быть указаны: – алгоритм программы; – используемые методы; – структура программы с описанием функций составных частей и связи между ними. Описание логической структуры программы выполняют с учетом текста программы на исходном языке	
4.	В разделе «Используемые технические средства» должны быть указаны типы электронных вычислительных машин и устройств, которые используются при работе	
5.	В разделе «Вызов и загрузка» должны быть указаны: – способ вызова программы с соответствующего носителя данных; – входные точки в программу	
6.	В разделе «Входные данные» должны быть указаны: – характер, организация и предварительная подготовка входных данных; – формат, описание и способ кодирования входных данных	
7.	В разделе «Выходные данные» должны быть указаны: – характер и организация выходных данных; – формат, описание и способ кодирования выходных данных	

Пример отчета о контроле содержания описания применения (ГОСТ 19.502-78).

Таблица 2

№ п/п	Требования	Результаты контроля
1.	Должны быть заполнены разделы аннотации и содержания	
2.	В разделе «Назначение программы» должны быть указаны назначение, возможности программы, ее основные характеристики, ограничения, накладываемые на область применения программы	

№ п/п	Требования	Результаты контроля
3.	В разделе «Условия применения» должны быть указаны условия, необходимые для выполнения программы (требования к необходимым для данной программы техническим средствам и другим программам, общие характеристики входной и выходной информации, а также требования и условия организационного, технического и технологического характера и т.п.)	
4.	В разделе «Описание задачи» должны быть указаны определения задачи и методы ее решения	
5.	В разделе «Входные и выходные данные» должны быть указаны сведения о входных и выходных данных	

Пример отчета о контроле содержания пояснительной записки (ГОСТ 19.404-79).

Таблица 3

№ п/п	Требования	Результаты контроля
1.	Пояснительная записка должна содержать следующие разделы: – введение; – назначение и область применения; – технические характеристики; – ожидаемые технико-экономические показатели; – источники, использованные при разработке	
2.	В разделе «Введение» должны быть указаны наименование программы и (или) условное обозначение темы разработки, а также документы, на основании которых ведется разработка с указанием организации и даты утверждения	
3.	В разделе «Назначение и область применения» должно быть указано назначение программы, краткая характеристика области применения программы	
4.	Раздел «Технические характеристики» должен содержать следующие подразделы: – постановка задачи на разработку программы, описание применяемых математических методов и, при необходимости, описание допущений и ограничений, связанных с выбранным математическим материалом; – описание алгоритма и (или) функционирования программы с обоснованием выбора схемы алгоритма решения задачи, возможные взаимодействия программы с другими программами; – описание и обоснование выбора метода организации входных и выходных данных; – описание и обоснование выбора состава технических и программных средств на основании проведенных расчетов и (или) анализов, распределение носителей	

№ п/п	Требования	Результаты контроля
	данных, которые использует программа	
5.	В разделе «Ожидаемые технико-экономические показатели» должны быть указаны технико-экономические показатели, обосновывающие выбранный вариант технического решения, а также, при необходимости, ожидаемые оперативные показатели	
6.	В разделе «Источники, использованные при разработке» должны быть указаны перечень научно-технических публикаций, нормативно-технических документов и других научно-технических материалов, на которые есть ссылки в основном тексте	