MINIX 3. Портирование POSIX программ

Цилюрик О.И.

< olej@front.ru >

Редакция 1.07

от 03.06.2010

Оглавление

Аннотация....1
Введение....1
Версии системы....1
Обозначения в тексте....2
Компиляция и сборка POSIX программы / проекта....2
Тестовый проект....2
Компиляция отдельных программ....4
Полезные советы относительно компиляции....6
Оформление собственного проекта Autoconf/Automake....6
Перенос проекта, подготовленного Autoconf/Automake....9
Полезные советы конфигурирования....10
Оформление MINIX пакета....10
Дополнительные источники информации....13

Аннотация

Описываются некоторые приёмы перенесения проектов GNU и отдельных программ, соответствующих стандартам POSIX, в систему MINIX 3.

Введение

Портирование программного обеспечения в MINIX из POSIX совместимых систем (чаще всего из Linux или *BSD) является во многих случаях нетривиальным, но и не невыполнимым. Главная часть работы зачастую заключается в изменениях в `Makefile` или скриптов сборки. Перекодирование больших частей программного обеспечения требуется только либо для программ, которые частично или полностью выполняются в пространстве ядра, либо для программ, использующих POSIX API, не реализованные в MINIX (например, потоки, и все API вида `pthread_*()`).

При портировании любой программы в MINIX нам предстоит рассматривать последовательно несколько задачи:

- Компиляция и сборка POSIX проекта. На этом этапе программный код, написанный и проверенный для другой POSIX системы, вам предстоит собрать в совершенно не предусмотренной этим кодом среде — операционной системе MINIX.
- Оформление MINIX проекта. Теперь, добившись работоспособности программы, вам предстоит оформить (дополнить) программный код так, чтобы его работоспособность могла легко воспроизводиться в любой другой инсталляции MINIX.
- Скрипты (утилиты) установки и удаления программных пакетов в системе, такие, как существующие: `packman, binpackage` etc., или недостающие, которые нужно дописать.

Последний вопрос — чисто технический, тем более, что он сильно меняется от версии к версии системы. А вот первые 2 вопроса, как 2 последовательные фазы, и будут рассмотрены в дальнейшем тексте. Между этими 2-мя фазами обязательно будет ещё проверка работоспособности, тестирование программы, но это уже совсем не относится непосредственно к предмету нашего рассмотрения.

Версии системы

Версии MINIX3 в очень большой мере «волатильны» - разработчики часто вносят существенные изменения,

даже не считая должным отражать их даже в MAN страницах. Вся основная часть описания отрабатывалась на стабильной версии 3.1.6 (релиз 6084). В используемой вами версии могут быть, порой, довольно существенные отличия, но основные принципы при этом сохраняются.

Обозначения в тексте

В самом тексте, все примеры команд (скопированные с терминала) будет показываться моноширинным шрифтом. Кроме того, в большинстве случаев пользовательский ввод в записи команды будет показан жирным шрифтом, а ответный вывод от системы — обычным. Короткие цитаты из различных источников информации будут показываться *курсивом*. Также, если любое слово в тексте должно считаться термином в данном контексте: имя или расширение файла, имя переменной, название программного пакета — то термин будет показываться моноширинным шрифтом.

Компиляция и сборка POSIX программы / проекта

Эти рекомендации будут служить отправной точкой для разработчиков, желающих портировать некоторое POSIX приложение в MINIX 3. Сложность процесса определяется разнообразием исходных условий задачи, форм, в которых может быть представлен исходный программный код, подлежащий портированию:

- это может быть простейшая тестовая программа на языке C, написанная вашим соседом по лестничной клетке, не содержащая в своём составе даже файла сборки `Makefile`;
- это может быть (чаще всего) программный проект, подготовленный к сборке GNU инструментарием `Autoconf/Automake`;
- это может быть просто программный проект, подготовленный к сборке другой технологией (их число постоянно множится), пример такой технологии, достаточно часто уже встречаемой, `cmake`;
- это может быть программный проект (даже весьма объёмный), полностью написанный на скриптовом языке shell, синтаксические детали языков shell (`ash, bash, zsh`) различаются; кроме того могут различаться файловые пути различных компонент в системе;
- это может быть программный проект, написанный на другом языке программирования, отличающемся от C, в этом случае вам придётся проверить наличие такого языкового средства в системе, или установить его (самым ярким примером этого случая является C++, который потребует установки компилятора `gcc`);

В любом случае, компиляция и сборка чужого программного проекта всегда будет оставаться процессом поисковым, и не подлежит формальному описанию. Имейте при этом себе в виду, что процесс этот может завершиться неудачей, не взирая на любые затраченные усилия (он может просто потенциально не собираться для данной операционной системы). Весь дальнейший текст этого раздела — это только рекомендации, и не более, как вы можете упростить себе жизнь на этом пути.

К этому моменту рассмотрения будем считать, что вы уже скачали архив интересующего вас проекта в архиве форматов `*.tgz , *.tar.gz,` или `*.tar.bz2`, и разархивировали его в рабочий каталог.

Тестовый проект.

В качестве тестового проекта, который пройдёт через все разделы описания, я буду рассматривать портирование довольно известного пакета `dbug-2.0.0` (http://sourceforge.net/projects/dbug/files/) — объектной библиотеки внутреннего отладчика. В минимально достаточном объёме нам достаточно перенести в наш проект:

```
# ls
Makefile dbug.c  dbug.h
# make
gcc -g -Wall -c -o dbug.o dbug.c
LANG=ua; gar -r libdbug.a dbug.o
gar: creating libdbug.a
```

Для возможности тестирования того, что мы собрали, добавим к проекту тестовую задачу `fact_dbug.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "dbug.h"

unsigned long factorial( int value ) {.
   unsigned long res;
   DBUG_ENTER( "factorial" );
   DBUG_PRINT( "info", ( "Got argument: '%d'", value ) );...
   if( 1 == value || 0 == value ) DBUG_RETURN( 1 );
   res = value * factorial( value - 1 );
   DBUG_RETURN( res );
}

int main( int argc, char* argv[] ) {
   register unsigned long result;.
   int i;...
   DBUG_ENTER( "main" );
   DBUG_PROCESS( argv[ 0 ] );
   for( i = 1; i < argc && argv[ i ][ 0 ] == '-'; i++ ) {
      switch( argv[ i ][ 1 ] ) {
         case '#':
            DBUG_PUSH( &(argv[ i ][ 2 ]) );.
      }
   }
   if( 1 != argc - i ) {
      printf( "usage: %s [-#{d|t|O}] <NUMBER>\n", argv[ 0 ] );
      DBUG_RETURN( 1 )
   }
   result = factorial( atoi( argv[ i ] ) );
   printf( "%ld\n", result );
   DBUG_RETURN( 0 )
}
```

Всё, что есть `DBUG_*` - это и есть макросы пакета `dbug`, прикомпонованного к тестовой задаче статически (в MINIX 3 никакой иной компоновки не существует). Полный файл сборки `Makefile` теперь имеет вид :

```
CC=gcc -g -Wall
AR=gar
LIST=dbug fact_dbug
all:        $(LIST)
dbug:       dbug.h dbug.c
            $(CC) -c -o dbug.o dbug.c
            LANG=ua; $(AR) -r libdbug.a dbug.o
fact_dbug:  fact_dbug.c
            $(CC) -o $@  $< dbug.o
# or:       $(CC) -o $@  $< libdbug.a
```

```
clean:
rm -f *.o *.a $(LIST).
```

Всё, проект в смысле законченной задачи выполняющейся в MINIX 3, завершён:

```
# make
gcc -g -Wall -c -o dbug.o dbug.c
LANG=ua; gar -r libdbug.a dbug.o
gar: creating libdbug.a
gcc -g -Wall -o fact_dbug  fact_dbug.c dbug.o
# ./fact_dbug -#t 3
| >factorial
| | >factorial
| | | >factorial
| | | <factorial
| | <factorial
| <factorial
6
<main
# ./fact_dbug -#d 3
factorial: info: Got argument: '3'
factorial: info: Got argument: '2'
factorial: info: Got argument: '1'
6
```

Компиляция отдельных программ

Я начну с рассмотрения простейшего случая, когда вам необходимо собрать для MINIX относительно простую программу: один или несколько файлов программного кода, к которым прилагается, или даже не прилагается файл сборки `Makefile`. Это, конечно, ещё не портирование публичного POSIX проекта, но проблемы, которые предстоит решить здесь, они все перенесутся на более сложные проекты.

Возникающие проблемы порождены теми обстоятельствами, что:

- Программа собиралась в другой операционной системе (Linux, Solaris, FreeBSD, ...) или даже в MINIX но предыдущей версии: порядок расположения имён по файлам определений (`*.h`) и библиотекам (`*.a`) может отличаться, могут отличаться путевые имена компонент в системе;
- Компиляторы могут различаться, даже если это `gcc`, то почти наверняка различающиеся версии — при этом у компиляторов могут различаться синтаксические расширения, опции, прагмы...

Переходим к компиляции проекта. Первая попытка компиляции обычно сразу показывает сомнительные места, и уже изначально указывает вообще на возможность успешной сборки:

- В MINIX у вас есть два альтернативных компилятора C: `cc` и `gcc`, каждый с соответствующим набором инструментов, например `ar` и `gar` для сборки статических библиотек (и у них могут отличаться опции запуска). При переходе от одной линии инструментов к другой приходится корректировать имена утилит в `Makefile`. Освободиться от этого можно если присваивать имена обрабатывающих утилит переменным скрипта, которые уже позже использовать в вызовах команд, например:

  ```
  CC=gcc -Wall -g
  AR=gar
  ...
  ```

```
	$(CC) -c -o dbug.o dbug.c
	LAN=ua; $(AR) -r libdbug.a dbug.o
...
```

- Часто ошибки связаны с различными синтаксическими стандартами компиляторов C. Одной из самых частых ошибок бывает использование однострочных комментариев вида:

  ```
  // комментарий, допускаемый gcc, но не допускаемый ACK cc
  ```

  в отличие от многострочных:

  ```
  /* комментарий, допускаемый и gcc и ACK cc */
  ```

- Компиляторы GNU `gcc`, Solaris `cc` (Solaris Studio) из кода которых может производиться портрование, допускают синтаксические расширения (и значительные, и отличающиеся в каждом случае) относительно стандартов, и специфические прагмы компилятора — всё это не будет распознано в MINIX. Пример одного из существенных расширений `gcc` — это определения встроенных функций (внутрь описания других функций). Вот пример такого исключения расширения в случае реального портирования:

  ```
  void _db_enter_( const char *_func_, const char *_file_, uint _line_,
                   const char **_sfunc_, const char **_sfile_,
                /* uint * _slevel_, char ***_sframep_ __attribute__ ((unused))) */
                   uint * _slevel_, char ***_sframep_ )
  ```

  Чаще всего такие синтаксические расширения, свойственные только одному конкретному компилятору, удаётся просто безболезненно комментировать.

- Сообщения о неопределённых ссылках (определениях функций) периода компиляции указывают либо на то, что нужно включить дополнительные `#include` директивы (если удаётся найти определения требуемых функций в `*.h` файлах), либо сразу указывают на невозможность сборки, как в следующем примере (в MINIX не реализованы потоки, и связанные с ними мютексы `pthread_mutex_t`):

  ```
  efence.c:197: undefined reference to `pthread_mutex_trylock'
  ```

  Найдите требуемый (содержащий нужное имя) файл-хедер в каталоге `/usr/include` и во всех его подкаталогах, и добавьте директиву `#include` с именем найденного файла. При использовании компилятора `gcc` (особенно на языке C++) в путь поиска нужно включить и каталог `/usr/gnu/include` и его подкаталоги.

- Часто простейший способ выяснить, какого заголовочного файла в MINIX вам не хватает для требуемого имени файла — это выполнить команду справки относительно этого имени:

  ```
  $ man <имя>
  ```

  И в первых строках подсказки (если она найдётся) вы найдёте имя заголовочного файла, например, для имени `strcat` в справке присутствует строка:

  ```
  #include <string.h>
  ```

- Следующим этапом после успешной компиляции будут не разрешённые имена на этапе сборки. Например, такого вида:

  ```
  /usr/tmp/ccpQfRNO.o:(.text+0x9): undefined reference to `_mcount'
  ```

  Это означает, что в строке компиляции не достаёт явного указания имени библиотеки. Вам предстоит разыскать эту библиотеку и указать её в команде компилятора. Проверить библиотеку на наличие искомого символа можно так:

  ```
  # objdump -t libc.a | grep _strstr
  00000000 g       .text 0000 00 05 _strstr
  ```

  А проверить, содержит ли хоть одна библиотека в каталоге (без указания конкретной библиотеки, чтобы

не терять время на каталог) нужное имя можно так:

```
# objdump -t lib*.a | grep _strstr
00000000 g       .text 0000 00 05 _strstr
00000000         *UND* 0000 00 01 _strstr
```

Возвращаясь к тестовому примеру `dbug`, отметим какие изменения пришлось внести в оригинальный код, написанный много лет назад :

- в файле `dbug.h` добавить объявление синонимов типа:

```
#define uint unsigned int
#define ulong unsigned long
```

- в файле `dbug.c` закомментировать в уже показанной выше строке атрибут, и то, это понадобилось для совместимости с ACK `cc`, не `gcc`:

```
/* uint * _slevel_, char ***_sframep_ __attribute__ ((unused))) */
```

Легко видеть, что изменения — минимальные, можно считать, что нулевые.

Полезные советы относительно компиляции

Битовые поля не распознаются `cc` компилятором. При необходимости, вы можете использовать `gcc` компилятор или удалить битовые поля из структур - это безвредно, если данные структуры не должны строго соответствовать структуре оборудования или памяти.

Вызовы `ioctl()` и `setsockopt()` являются источниками проблем портирования. Вызовы функций (константы) могут быть определены, однако или не иметь вовсе, или иметь ограниченную реализацию, так что вы заранее не сможете проверить все эти вызовы. В частности, убедитесь, что каждый вызов должным образом проверен на возвращаемое сообщение об ошибке.

В MINIX вызовы `recv()` и `send()` не поддерживают флаги.

Нет способа запустить сервер, прослушивающий петлевой интерфейс на `localhost` (`127.0.0.1`), потому что в MINIX нет выделенного петлевого интерфейса. Если вы столкнулись с этой проблемой, то можете взамен использовать прослушивание с адреса `INADDR_ANY`, но отметьте при этом, что это будет работь только при работающем подключении к сети. Также помните о проблемах безопасности, связанных с принятием нелокальных сетевых соединений.

Используйте последнюю доступную версию MINIX. Это даёт расширенную функциональность библиотек, в них могут быть реализованы те элементы API, отсутствующие в предыдущих версиях.

Оформление собственного проекта Autoconf/Automake

Теперь мы рассмотрим как наш собственный проект, подобный тому, который рассмотрен выше, сделать конфигурируемым средствами `Autoconf/Automake`. После этого наш проект станет в значительной мере переносимым между POSIX операционными системами. В такой технологии мы можем подготовить к переносимости любой программный проект, начиная от простейшей программы «Hello world!» в одном файле.

Подготовку пакета мы продолжаем рассматривать на тестовом проекте. Я показываю самые упрощенные до примитивного шаги, но и этого достаточно для понимания хода работы. В реальности приходится использовать куда больше возможностей пакетов, обобщённо называемых `Autotools`, которые хорошо документированы, ссылки на документацию приведены в завершение текста. Возвращаем проект в исходное состояние (я чуть-чуть подправил `Makefile`, превратив его в `Makefile.in`, но об этом позже):

```
# ls
Makefile.in   dbug.c   dbug.h   fact_dbug.c
```

Прежде всего, нам предстоит составить файл `configure.in`, содержащий макросы для тестов проверок в новой операционной системе. Но мы не станем делать это сами, а воспользуемся утилитой `autoscan`, которая создаст нам `configure.scan` как прототип будущего `configure.h` :

```
# autoscan
# ls
Makefile   Makefile.in   autoscan.log   configure.scan   dbug.c   dbug.h   fact_dbug.c
# cat configure.scan
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.60)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([dbug.c])
AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h string.h sys/param.h unistd.h])
# Checks for typedefs, structures, and compiler characteristics.
AC_C_CONST
# Checks for library functions.
AC_FUNC_CHOWN
AC_FUNC_MALLOC
AC_FUNC_VPRINTF
AC_CHECK_FUNCS([bzero strrchr])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Обычно здесь вам предстоит добавить или убрать некоторые из тестов в этой последовательности, но я для краткости не стану этого делать:

```
# cp configure.scan configure.in
```

Далее создаём `config.h.in`, опять же воспользуемся для этого генератором заготовки `autoheader` :

```
# autoheader
# ls
Makefile.in   autom4te.cache   config.h.in   configure.in
configure.scan   dbug.c   dbug.h   fact_dbug.c
```

Здесь также предполагается, что вы станете сознательно править `config.h.in`. Но сейчас мы не станем этим заниматься.

И вот теперь время:

```
# autoconf
# ls -l conf*
-rw-r--r-- 1 root  operator    2183 Jun  3 19:58 config.h.in
-rwxr-xr-x 1 root  operator  159285 Jun  3 20:04 configure
```

```
-rw-r--r-- 1 root  operator     647 Jun  3 19:57 configure.in
-rw-r--r-- 1 root  operator     647 Jun  3 19:43 configure.scan
```

У нас появился файл `configure`, причём у него установлен флаг исполнимости. Но прежде, чем исполнять `configure` (для чего всё и делалось), рассмотрим файл прототипа `Makefile` - файл `Makefile.in` (я его получил чисто механически из использовавшегося выше `Makefile` проекта, путём небольших замен) :

```
CC=@CC@
COPT=-g -Wall
AR=gar
LIST=dbug fact_dbug
all:        $(LIST)
dbug:       dbug.h dbug.c
            $(CC) $(COPT) -c -o dbug.o dbug.c
            LANG=ua; $(AR) -r libdbug.a dbug.o
fact_dbug:  fact_dbug.c
            $(CC) $(COPT) -o $@  fact_dbug.c dbug.o
clean:
            rm -f *.o *.a $(LIST)
```

Фактически, это практически тот `Makefile`, который мы использовали при простой компиляции своего проекта, за одним принципиальным исключением: переменной `@CC@`. Значения переменным вида `@XXXX@` будет подставлять при выполнении скрипт `configure`, исходя из анализа тестов, которые он проводит над операционной системой. Выполняем (выполним так, как рекомендовано в MINIX при сборке пакетов, без учёта флагов исполнимости):

```
# sh -e ./configure
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
...
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
...
configure: creating ./config.status
config.status: creating Makefile
config.status: creating config.h
# ls
Makefile      config.h      config.log      configure      configure.scan   dbug.h
Makefile.in   config.h.in   config.status   configure.in   dbug.c           fact_dbug.c
```

Файл содержит детальное описание построенной конфигурации, в частности, значения переменных, которые использованы в `Makefile` :

```
## ----------------- ##
## Output variables. ##
## ----------------- ##
CC='gcc'
CFLAGS='-g -O2'
```

```
...
```

Всё! Из собственного компилируемого проекта мы получили конфигурируемый пакет:

```
# make
gcc -g -Wall -c -o dbug.o dbug.c
LANG=ua; gar -r libdbug.a dbug.o
gar: creating libdbug.a
gcc -g -Wall -o fact_dbug  fact_dbug.c dbug.o
```

Схема показана в примитивном виде, но она остаётся принципиально той же для любого самого сложного проекта.

Перенос проекта, подготовленного Autoconf/Automake

Теперь, когда мы уже умеем строить конфигурируемый пакет, мы можем переконфигурировать под MINIX любой публичный пакет (при условии, что это возможно в принципе). Подавляющее большинство публичных программных пакетов (например GNU), будут доступны вам именно в такой форме представления (хотя это и устаревшая технология и на смену ей подходят другие). Отличительно особенностью (зачастую, но не всегда) таких проектов (и их доля в общем числе проектов — подавляющая) то, что в каталоге проекта содержится файл `./configurre` (если нет, то бывает необходимо произвести те предварительные фазы, которые обсуждались выше).

Скачайте свежие экземпляры файлов `config.guess` и `config.sub` (см. приложение к тексту), они определяют машинную конфигурацию, и нужны для работы `./configure`, но в обрабатываемом пакете могут находиться весьма старые экземпляры `config.guess` и `config.sub`, в которых вообще не прописана такая операционная система как MINIX. Экземпляры `config.guess` и `config.sub` понадобятся вам один раз: дальше вы можете копировать эту свою копию файлов из одного пакета в другой. Установите своим экземплярам файлов флаг выполнимости:

```
# chmod a+x config.sub
# chmod a+x config.guess
```

Выполняем скрипт `./configure` :

```
# ./configure
creating cache ./config.cache
checking host system type... i686-pc-minix
checking for gcc... gcc
...
updating cache ./config.cache
creating ./config.status
creating Makefile
creating Make.defines
creating config.h
```

Это может быть весьма продолжительный процесс, но завершение его должно быть подобно тому, что показано: создание файла `Makefile`. В некоторых скриптах (особенно это касается Linux пакетов, где основной оболочкой практически всегда является `bash`) могут использоваться синтаксические конструкции, не поддерживаемые командной оболочкой `ash` (используется по умолчанию в MINIX). В этом случае используйте другую командную оболочку, такие, как `bash` или `zsh`:

```
# bash
bash-3.00# ./configure
loading cache ./config.cache
```

```
checking host system type... i686-pc-minix
...
```

Компиляция проекта обычно (но не обязательно) происходит по команде:

```
# make
...
```

Такая форма предполагает сборку цели `all`, определённой в `Makefile`. В более сложных случаях, может оказаться необходимым собрать некоторую предварительную цель, например библиотеку:

```
# make lib
...
```

Загляните в содержимое файла `Makefile` — там всё ясно относительно целей. В любом случае, выполнение команды `make` должно завершиться без ошибок (могут быть предупреждения, но и их появление нежелательно). Утилита `make` в составе MINIX имеет некоторые ограничения функциональности (в сравнении с GNU). Если она создаёт какие-либо сообщения об ошибках, можно попробовать более функциональную утилиту `gmake` (GNU `make`).

Полезные советы конфигурирования

Часто еще на этапе `./configure` можно указать пути установки параметром `--prefix:`

```
# ./configure --prefix=/usr/local
```

Многие опциональные параметры, которые можно переопределить на этапе `./configure` можно уточнить командой:

```
# ./configure --help
```

Если пакет устанавливает библиотеки для последующего использования другими программами, то поместите библиотеки, собранные с помощью `cc`, в `/usr/local/lib/ack`; а библиотеки, собранные с помощью `gcc`, в `/usr/local/lib/gcc/`.

Ещё на этапе компиляции собирайте команды в скрипт `build.minix`, в котором определяются конкретные пути, переменные и прочее, имеющие специфические значения в MINIX. Используйте его при повторных и тестовых сборках пакета. Это же скрипт понадобится вам при оформлении установочного пакета MINIX.

Оформление MINIX пакета

Если вам удалось оживить программный пакет перенесенный из другой системы, то оформление пакета MINIX — это уже формальная сторона. Целью этого действия должно стать то, что этот пакет может выполняться не только в вашей системе, но и может быть перенесен и установлен в аналогичной.

В MINIX нет никакой пакетной системы, в строгом её понимании, как например `rpm` в Linux, `qpr` в QNX, или `pkgsrc` в NetBSD (и множестве перенявших её системах). Многократно упоминаемая в обсуждениях установка пакетов в MINIX — это много лет известная в UNIX установка «разархивированием от корня». Бинарный пакет MINIX представляет собой дерево с размещёнными в нём файлами, которое копируется от пути '/'. Вот что представляет из себя пакет известного GNU командного калькулятора, портированного в MINIX как `bc-1.06.tar.bz2` (http://www.minix3.org/packages/i386/3.1.7/bc-1.06.tar.bz2), в MINIX нет команды `tree`, архив перенесен и проанализирован в Linux:

```
$ tree
`-- usr
    `-- local
        |-- bin
        |   |-- bc
        |   `-- dc
        |-- info
```

```
        |   |-- bc.info
        |   `-- dc.info
        `-- man
            `-- man1
                |-- bc.1
                `-- dc.1
6 directories, 6 files
```

Символьный пакет MINIX (обычно перенесенный из Linux) — это пакет исходных кодов в стандарте технологии `Automake/Autoconf`. И тот и другой формат после изготовления архивируется `bzip2`. Соглашения построения пакетов дополняются рядом договорных правил, чтобы сделать пакеты более единообразными для установки их утилитами-скриптами (как это называют в MINIX: «менеджеры пакетов») подобными `easypack` или `packman`.

Далее этот раздел и содержит перечисление таких правил (основных из числа изложенных основной командой разработчиков MINIX, с некоторыми моими комментариями). Хотя это и не оговорено специально в правилах, они относятся к оформлению пакетов исходных кодов, которые должны поступать на вход скриптам, выполняющим сборку программ и их установку.

Каждый пакет должен иметь уникальное имя, например `foo-1.2.4`, где часть, предшествующая дефису — название программы, и часть после дефиса — номер её версии. Вместе они образуют имя пакета.

Добавьте скрипт командной оболочки `build.minix` в корневой каталог вашего пакета. В правилах приводится простейший вид содержимого такого скрипта:

```
#!/bin/sh
make
make install
```

Гораздо содержательнее скрипт `build.minix`, который использовал Marco Slot в своей сборке GTK+, файл хоть и великоват, но стоит того, чтобы рассмотреть его полностью:

```
#!/bin/sh -e
# Marco Slot <marco@few.vu.nl>
if [ -f Makefile ] ; then
   /usr/gnu/bin/gmake uninstall || true
   /usr/gnu/bin/gmake clean     || true
   /usr/gnu/bin/gmake distclean || true
fi
export PATH="/usr/local/bin:$PATH"
AR="/usr/gnu/bin/gar"                                                  \
CC="/usr/gnu/bin/gcc"                                                   \
CFLAGS="-Wall -O2 -D_POSIX_SOURCE=1 -D_MINIX=1                          \
        -I/usr/X11R6-gcc/include -I/usr/local/include"                  \
CONFIG_SHELL="/bin/bigsh"                                               \
FONTCONFIG_CFLAGS="-I/usr/X11R6-gcc/include/fontconfig                  \
                   -I/usr/X11R6-gcc/include/freetype2"                  \
FONTCONFIG_LIBS="-L/usr/X11R6-gcc/lib -lfontconfig -lexpat -lfreetype -lz"   \
FREETYPE_CONFIG="/usr/X11R6/bin/freetype-config --prefix=/usr/X11R6-gcc"     \
GREP="/usr/local/bin/grep"                                              \
```

```
GNUMAKE="/usr/gnu/bin/gmake"                                                    \
LD="/bin/true"                                                                  \
LIBS="-L/usr/local/lib/gcc -L/usr/X11R6-gcc/lib"                                \
M4="/usr/gnu/bin/m4"                                                            \
PKG_CONFIG="/usr/local/bin/pkg-config --static"                                 \
PKG_CONFIG_PATH="/usr/local/lib/gcc/pkgconfig:/usr/X11R6-gcc/lib/pkgconfig"     \
RANLIB="/bin/true"                                                              \
SHELL="/bin/bigsh"                                                              \
lt_cv_sys_max_cmd_len="4096"                                                    \
/bin/bigsh configure                                                            \
   --prefix=/usr/local                                                          \
   --libdir=/usr/local/lib/gcc                                                  \
   --mandir=/usr/local/man                                                      \
   --x-libraries=/usr/X11R6-gcc/lib                                             \
   --x-includes=/usr/X11R6-gcc/include                                          \
   $*
find . -name Makefile | xargs perl -pi -e 's/--tag=CC//g'
/usr/gnu/bin/gmake
/usr/gnu/bin/gmake install
```

Основные действия этого скрипта: установить явно в нужные значения все переменные окружения, и только после этого выполнить сборку. Смысл такого (показанного) скрипта: написав для себя один раз такой скрипт, использовать его для сборки всех своих пакетов; лишние, не используемые, переменные окружения не повредят сборке.

Сценарий `build.minix` должен нормально завершаться если пакет установился правильно, иначе завершаться с ошибкой.

Все вызовы `build.minix` из устанавливающих скриптов будут осуществляться с опцией `sh -e`, так что неудачные команды будут прерывать сценарий. Для некоторых пакетов возможно потребуются дополнительные (конфигурирующие) команды, но в большинстве случаев пакет должен собираться и устанавливаться правильно выполнением:

```
$ sh -e build.minix
```

Процедура сборки не должна полагаться на то, что файлы имеют корректный режим доступа (как, например, исполняемые). Например, не используйте: `./configure` в `build.minix`, используйте: `sh -e configure`.

Примечание: именно из-за таких соображений, во многих пакетах MINIX сценарии `build.minix` и `./configure` не имеет установленных флагов исполнимости. По моему скромному разумению, пожелания этого абзаца вызывают глубокие сомнения, и, скорее всего, это что-то из области заблуждений или из детских комплексов разработчиков этих правил пакетирования.

Для завершения создания пакета (архивирования) выполните:

```
$ cd ..;  tar cf - foo-1.2.4  |  bzip2 >foo-1.2.4.tar.bz2
```

Для проверки корректности сборки в системе есть утилита:

```
$ binpackage
Usage: /usr/bin/binpackage sourcedir packagedir
```

Вы можете проверить сможет ли ваш исходный пакет собрать корректный бинарный пакет выполнив эту утилиту, `binpackage` запустит скрипт сборки, полученный пакет будет в `packagesdir/sourcedir.tar.bz2`.

Добавьте файл `.descr`, содержащий описание пакета в одну строку (до 50 символов). Когда `packman` показывает список доступных пакетов, он отображает эту краткую информацию после названия пакета.

Дальше я просто перечислю те из правил для разработчиков пакетов, которые не вызывают больших сомнений и вопросов:

- Устанавливайте `PREFIX=/usr/local` (при `./configure`) , но посмотрите следующее правило о библиотеках. Библиотеки должны устанавливаться в `/usr/local/lib/ack`, или в `/usr/local/lib/gcc`, или в оба эти каталога, в зависимости от используемых компилятора и компоновщика.
- Для компиляции программ вы можете использовать `cc` или `gcc`, хотя мы, по возможности предпочитаем `cc`. Эти два компилятора выдают разные предупреждения, и как правило они оба правы. Для C++ программ используйте g++.
- Команда `make install` должна использовать программу `install` для копирования вашей программы в `/usr/local/bin`. Этот метод будет работать даже тогда, когда пользователь вошёл в систему как `bin` (не как `root`).
- Каждая программа должна иметь MAN страницу в стандартном формате. Команда `make install` должна инсталлировать MAN страницы в соответствующие подкаталоги `/usr/local/man`. Большинство MAN страниц о программах помещаются в `/usr/local/man/man1`, но расширенные руководства пользователя должны быть в `/usr/local/man/man9`.
- Сделайте ваш `build.minix` сценарий таким, чтобы при повторном запуске он устанавливал всё с начала. Мало того, что это в значительной степени поможет после неудачной сборки, но наш механизм для создания бинарных пакетов опирается на это.

Дополнительные источники информации

1. Перевод «Autoconf. Создание скриптов для автоматической конфигурации. Редакция 2.13, Autoconf версии 2.13, Декабрь 1998», David MacKenzie и Ben Elliston :

http://www.linux.org.ru/books/GNU/autoconf/autoconf-ru_toc.html

2. Перевод «GNU Automake. Для версии 1.4, 10 January 1999», David MacKenzie и Tom Tromey :

http://public.ttknn.net/mirrors/www.linux.org.ru/books/GNU/automake/automake-ru_toc.html

3. Местоположение эталонных файлов `config.guess` и `config.sub` :

http://git.savannah.gnu.org/gitweb/?p=config.git;a=blob_plain;f=config.guess;hb=HEAD

http://git.savannah.gnu.org/gitweb/?p=config.git;a=blob_plain;f=config.sub;hb=HEAD