GNU Automake

Этот документ описывает пакет GNU Automake версии 1.16.1 (26.02.2018), применяемый для создания стандартизованных файлов Makefile на основе шаблонов.

Оглавление

1. Введение

Automake служит инструментом для автоматического создания файла Makefile.in из файла Makefile.am. Каждый файл Makefile.am содержит набор определений переменных make[1], в которые иногда добавляются правила. Создаваемые файлы Makefile.in соответствуют стандартам GNU Makefile.

Документ GNU Makefile Standards достаточно велик, сложен и может изменяться. Цель Automake заключается в упрощении поддержки файлов Makefile для сопровождающих пакеты людей и перенесение этой нагрузки на сопровождающих пакет Automake.

Обычно входными данными Automake являются просто определения переменных, а результатом - файлы Makefile.in.

Automake вносит в проекты ряд ограничений. Например, предполагается, что проект использует программу Autoconf. Имеются также ограничения для содержимого configure.ac.

Для Automake требуется пакет perl, применяемый при создании Makefile.in. Однако создаваемые с помощью Automake дистрибутивы полностью соответствуют стандартам GNU и не требуют для сборки наличия perl.

2. Введение в Autotools

Automake является частью пакета Autotools, включающего файлы configure, configure.ac, Makefile.in, Makefile.am, aclocal.m4 и др., часть которых создаётся Autoconf или Automake. Целью этого раздела является ознакомление с механизмами работы и связями между этими файлами. Обучающие материалы, подготовленные Alexandre Duret-Lutz, доступны по ссылке.

2.1. Введение в систему сборки GNU

Очевидно, что разработчикам программ требуется система сборки. В среде Unix такая среда реализуется на основе команды make, а рецепт (задание) для сборки содержится в файле Makefile. Этот файл представляет собой набор правил для сборки файлов пакета. Например, программа prog может быть собрана путём запуска компоновщика для файлов main.o, foo.o и bar.o, а файл main.o - путём запуска компилятора для main.c и т. д.. При каждом запуске make считывается Makefile, проверяется наличие и время изменения упомянутых в нем файлов и принимается решение о сборке (повторной сборке), после чего выполняются соответствующие команды.

[1] *Эти переменные называют также макросами make, однако здесь термин «макрос» относится к макросам Autoconf.*

Когда пакет нужно собрать не на той платформе, где он был разработан, файл Makefile обычно приходится редактировать. Например, на платформе сборки компилятор может называться иначе или использовать другие опции. В 1991 г. David J. MacKenzie, устав от настройки Makefile для 20 разных платформ, с которыми приходилось работать, написал простой shell-сценарий configure для автоматического создания файлов Makefile. Для компиляции пакета теперь было достаточно ввести команду

```
./configure && make.
```

Сегодня этот процесс стандартизован в проекте GNU. Документ GNU Coding Standards указывает, что в каждый проект GNU следует включать сценарий configure, и описывает его минимальный интерфейс. Для файлов Makefile также имеется ряд соглашений. В результате получилась унифицированная система сборки, которая позволяет установить почти любой пакет похожим способом. В простейшем варианте это последовательность команд

```
./configure && make && make install
```

Эту систему сборки назвали GNU Build System, поскольку она «выросла» из проекта GNU. Однако система используется множеством других пакетов, поскольку единые соглашения обеспечивают ряд преимуществ.

Пакет Autotools включает набор инструментов создания системы сборки GNU для конкретного пакета. Программа Autoconf работает в основном с configure, а Automake - с файлами Makefile. Систему сборки GNU можно создать и без этих инструментов, однако это довольно обременительно и часто ведёт к ошибкам.

2.2 Примеры использования GNU Build System

Здесь рассмотрено несколько примеров использования системы сборки GNU. Эти примеры можно выполнить самостоятельно, воспользовавшись файлом amhello-1.0.tar.gz из состава Automake. При наличии Automake в системе этот файл будет в каталоге PREFIX/share/doc/automake/, где PREFIX указывает путь установки, заданный в конфигурации (по умолчанию /usr/local, но Automake в большинстве дистрибутивов GNU/Linux устанавливается в /usr). Если пакет Automake не установлен в системе, можно найти копию файла в каталоге doc/ пакета Automake.

В части примеров представлены расширения GNU Build System, включённые в систему сборки, создаваемую Autotools.

2.2.1. Базовая установка

Базовая процедура установки имеет вид

```
~ % tar zxf amhello-1.0.tar.gz
~ % cd amhello-1.0
~/amhello-1.0 % ./configure
...
config.status: creating Makefile
config.status: creating src/Makefile
...
~/amhello-1.0 % make
...
~/amhello-1.0 % make check
...
~/amhello-1.0 % su
Password:
/home/adl/amhello-1.0 # make install
...
/home/adl/amhello-1.0 # exit
~/amhello-1.0 % make installcheck
...
```

Сначала пользователь распаковывает архив. Здесь и далее для простоты используется команда tar zxf, применимая не во всех системах. В системах без GNU tar взамен следует применять команду gunzip -c amhello-1.0.tar.gz | tar xf -.

Затем пользователь переходит в созданный каталог для запуска сценария configure, которые проверяет различные переменные системы и создаёт файлы Makefile. В этом примере создаётся два Makefile, но в реальности такие файлы обычно создаются в каждом каталоге пакета.

После этого можно запускать make. Эта команда будет собирать все программы, библиотеки и сценарии для пакета., размещая файлы в заданные места дерева исходного кода. В примере компилируется программа hello.

Команда make check запускает тесты для собранного пакета. Этот этап не обязателен, но зачастую имеет смысл проверить, как ведёт себя программа, ещё до её установки. В приведённом примере make check ничего не делает.

После сборки и тестирования пакета можно установить его в системе. Установка включает копирование программ, библиотек, заголовочных файлов, сценариев и других данных в нужные каталоги системы. Для этого служит команда make install. По умолчанию все будет устанавливаться в каталог /usr/local - исполняемые файлы в /usr/local/bin, библиотеки - в /usr/local/lib и т. д. Обычно эти каталоги не доступны для рядового пользователя, поэтому нужно получить полномочия root. В примере команда make install будет копировать программу hello в /usr/local/bin и файл README - в /usr/local/share/doc/amhello.

Последней операцией является проверка установки с помощью команды make installcheck, которая может запускать тесты для установленных файлов (make check выполняет тесты в дереве исходных кодов, а make installcheck - в реальной системе). Эти тесты могут отличаться от предшествующих, например, могут включать проверки, которые невозможны в дереве кода. В тестах могут также использоваться иные файлы, нежели при команде make check. Тесты позволяют проверить полноту и корректность установки.

В настоящее время далеко не все пакеты включают тесты installcheck, поэтому проверку используют редко.

2.2.2. Стандартные цели Makefile

До этого были рассмотрены 4 способа применения команды make в системе сборки GNU - make, make check, make install и make installcheck. Слова check, install и installcheck, передаваемые в качестве аргумента команды make, называют целями (target). Команда make является сокращением make all, а цель all применяется по умолчанию в GNU Build System. Ниже приведён список наиболее распространённых целей, указанных в GNU Coding Standards.

make all
Собирает программы, библиотеки, документацию и т. п. (совпадает с make).
make install
Устанавливает все нужное, копируя файлы из дерева кода в системные каталоги.
make install-strip
Делает то же, что make install, а затем вырезает отладочные символы (они могут пригодиться при диагностике).
make uninstall
Противоположна make install и удаляет установленные файлы (должна быть запущена из того же дерева, которое служило для установки).
make clean
Удаляет из дерева сборки файлы, созданные командой make all.
make distclean
Удаляет также файл ./configure.
make check
Запускает тесты, если они имеются.
make installcheck
Проверяет установленные программы или библиотеки, если такая проверка поддерживается.
make dist
Заново создаёт архив PACKAGE-VERSION.tar.gz из файлов исходного кода.

2.2.3. Переменные для стандартных каталогов

Стандарт кодирования GNU задаёт также иерархию переменных для указания каталогов установки.

Переменная	*Значение по умолчанию*
prefix	/usr/local
exec_prefix	${prefix}
bindir	${exec_prefix}/bin
libdir	${exec_prefix}/lib
includedir	${prefix}/include
datarootdir	${prefix}/share
datadir	${datarootdir}
mandir	${datarootdir}/man
infodir	${datarootdir}/info
docdir	${datarootdir}/doc/${PACKAGE}

Роли большинства этих каталогов понятны из имён. В пакетах каждый из устанавливаемых файлов помещается в один из таких каталогов. Например, в пакете amhello-1.0 программа hello помещается в BINDIR - каталог для исполняемых файлов. По умолчанию это каталог /usr/local/bin, но пользователь может указать иное имя при вызове сценария configure. Файл README будет помещён в каталог DOCDIR (по умолчанию /usr/local/share/doc/amhello).

Пользователь может изменить каталоги установки, как показано ниже.

```
~/amhello-1.0 % ./configure --prefix ~/usr
...
~/amhello-1.0 % make
...
~/amhello-1.0 % make install
```

Это приведёт к установке ~/usr/bin/hello и ~/usr/share/doc/amhello/README.

Список переменных для каталогов можно получить с помощью команды ./configure --help.

2.2.4. Стандартные переменные конфигурации

Стандарт кодирования GNU определяет также список конфигурационных переменных, используемых при сборке.

CC
команда компилятора C.
CFLAGS
флаги компилятора C.
CXX
команда компилятора C++.
CXXFLAGS
флаги компилятора C++.
LDFLAGS
флаги компоновщика.
CPPFLAGS
флаги препроцессора C/C++.

Сценарий configure обычно выполняет всю работу по установке нужных значений этих переменных, но иногда может потребоваться переопределение значений. Например, в системе может быть установлено несколько компиляторов и нужный вам может отличаться от принятого по умолчанию.

Ниже показано, как можно использовать configure для выбора компилятора gcc-3, использования заголовочных файлов ~/usr/include при компиляции и библиотек ~/usr/lib - при сборке.

```
~/amhello-1.0 % ./configure --prefix ~/usr CC=gcc-3 \
CPPFLAGS=-I$HOME/usr/include LDFLAGS=-L$HOME/usr/lib
```

Список переменных конфигурации можно получить с помощью команды ./configure --help.

2.2.5. Переопределение принятых по умолчанию значений в config.site

При установке нескольких пакетов с общими параметрами удобно создать файл для хранения таких настроек. Если в системе существует файл PREFIX/share/config.site, сценарий configure будет использовать его (команда source).

Воспользуемся приведённой ниже командой настройки конфигурации

```
~/amhello-1.0 % ./configure --prefix ~/usr CC=gcc-3 \
CPPFLAGS=-I$HOME/usr/include LDFLAGS=-L$HOME/usr/lib
```

В предположении установки множества пакетов в ~/usr и использования общих определений CC, CPPFLAGS и LDFLAGS, можно автоматизировать процесс, создав файл ~/usr/share/config.site вида

```
test -z "$CC" && CC=gcc-3
test -z "$CPPFLAGS" && CPPFLAGS=-I$HOME/usr/include
test -z "$LDFLAGS" && LDFLAGS=-L$HOME/usr/lib
```

После этого сценарий configure при каждом использовании префикса ~/usr будет использовать приведённый выше файл config.site и задавать указанные в нем переменные.

```
~/amhello-1.0 % ./configure --prefix ~/usr
configure: loading site script /home/adl/usr/share/config.site
...
```

2.2.6. Параллельная сборка (VPATH)

Система сборки GNU различает деревья исходного кода и сборки. Корнем дерева кодов является каталог, где находится сценарий configure. Здесь хранятся файлы кода, которые могут быть размещены в разных подкаталогах.

Деревом сборки является каталог, из которого запускается сценарий configure. В этом каталоге размещаются все объектные файлы, программы, библиотеки и другие файлы, собранные из исходного кода. В дереве сборки обычно используются подкаталоги, автоматически создаваемые системой сборки.

Если configure выполняется из своего каталога, деревья исходного кода и сборки объединяются как в показанном выше примере сборки. Но можно эти деревья разделить, как показано ниже.

```
~ % tar zxf ~/amhello-1.0.tar.gz
~ % cd amhello-1.0
~/amhello-1.0 % mkdir build && cd build
~/amhello-1.0/build % ../configure
...
~/amhello-1.0/build % make
...
```

Такие конфигурации часто называют деревом параллельной сборки (parallel builds или VPATH builds), хотя этот термин может вводить в заблуждение. Поэтому дальше будет применяться термин сборка VPATH1. Такой вариант позволяет использовать одно дерево исходных кодов для сборки разных конфигураций. Например,

```
~ % tar zxf ~/amhello-1.0.tar.gz
~ % cd amhello-1.0
~/amhello-1.0 % mkdir debug optim && cd debug
~/amhello-1.0/debug % ../configure CFLAGS='-g -O0'
...
~/amhello-1.0/debug % make
...
~/amhello-1.0/debug % cd ../optim
~/amhello-1.0/optim % ../configure CFLAGS='-O3 -fomit-frame-pointer'
...
~/amhello-1.0/optim % make
...
```

На сетевых файловых системах может применяться похожая модель для сборки на разных машинах. Например, исходные коды могут размещаться в каталоге, доступном хостам HOST1 и HOST2, которые могут использовать разные платформы.

```
~ % cd /nfs/src
/nfs/src % tar zxf ~/amhello-1.0.tar.gz
```

на первом хосте можно создать локальный каталог сборки build, как показано ниже.

```
[HOST1] ~ % mkdir /tmp/amh && cd /tmp/amh
[HOST1] /tmp/amh % /nfs/src/amhello-1.0/configure
...
[HOST1] /tmp/amh % make && sudo make install
...
```

Здесь предполагается, что имя устанавливающего программы указано в конфигурации sudo, что позволяет ему использовать команду make install с правами root (это удобней использования su, описанного выше).

На втором хосте можно поступить также (может быть даже в то же самое время).

```
[HOST2] ~ % mkdir /tmp/amh && cd /tmp/amh
[HOST2] /tmp/amh % /nfs/src/amhello-1.0/configure
...
[HOST2] /tmp/amh % make && sudo make install
...
```

В этом случае ничто не препятствует сделать каталог /nfs/src/amhello-1.0 доступным лишь для чтения. Фактически сборки VPATH позволяет работать даже с исходными кодами, размещёнными на недоступных для записи средах, например, CD-ROM.

2.2.7. Раздельная установка

В предыдущем примере два хоста использовали общее дерево исходных кодов, но компиляция и установка выполнялись на каждом хосте раздельно. Система сборки GNU поддерживает также сетевые варианты, где часть устанавливаемых файлов совместно используется несколькими хостами. Это достигается путём разделения зависимых и независимых от архитектуры файлов и создания двух целей Makefile для установки этих компонент. Этими целями служат install-exec для зависимых от архитектуры файлов и install-data - для остальных. Использованная выше команда make install может считаться комбинацией make install-exec install-data.

С точки зрения системы сборки GNU различие между зависимыми и независимыми от архитектуры файлами основано лишь на переменной каталога, указанного для установки. В представленном выше списке переменные, основанные на EXEC-PREFIX, указывают зависимые от архитектуры компоненты, чьи файлы устанавливаются командой make install-exec. Другие каталоги относятся к архитектурно-независимым компонентам, устанавливаемым make install-data.

Ниже показано, как можно было разделить установку для двух хостов с размещением пакета непосредственно в /usr и использованием общего каталога /usr/share. На первом хосте можно использовать команды

```
[HOST1] ~ % mkdir /tmp/amh && cd /tmp/amh
[HOST1] /tmp/amh % /nfs/src/amhello-1.0/configure --prefix /usr
...
[HOST1] /tmp/amh % make && sudo make install
...
```

На втором хосте достаточно установить лишь зависимые от архитектуры файлы, как показано ниже.

```
[HOST2] ~ % mkdir /tmp/amh && cd /tmp/amh
[HOST2] /tmp/amh % /nfs/src/amhello-1.0/configure --prefix /usr
...
[HOST2] /tmp/amh % make && sudo make install-exec
...
```

В пакетах с проверкой установки имеет смысл команда make installcheck для тестирования корректности инсталляции.

2.2.8. Кросс-компиляция

Кросс-компиляцией называют сборку пакетов на платформе, которая отличается от целевой платформы, где будут работать пакет. Здесь важно различать платформу сборки (build), где происходит компиляция, и целевую (host) платформу, где программы будут использоваться. Ниже показаны опции configure для задания платформ.

--build=BUILD
Система, на которой собирается пакет.
--host=HOST
Система, где будут работать собранные программы и библиотеки.

При указании опции --host сценарий configure будет искать набор инструментов кросс-компиляции для этой платформы. Обычно в именах таких инструментов название целевой платформы служит префиксом. Например, кросс-инструменты для MinGW32 могут называться i586-mingw32msvc-gcc, i586-mingw32msvc-ld, i586-mingw32msvc-as и т. п.

Ниже приведён пример сборки amhello-1.0 для i586-mingw32msvc на компьютере GNU/Linux.

```
~/amhello-1.0 % ./configure --build i686-pc-linux-gnu --host i586-mingw32msvc
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for i586-mingw32msvc-strip... i586-mingw32msvc-strip
checking for i586-mingw32msvc-gcc... i586-mingw32msvc-gcc
checking for C compiler default output file name... a.exe
checking whether the C compiler works... yes
checking whether we are cross compiling... yes
checking for suffix of executables... .exe
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether i586-mingw32msvc-gcc accepts -g... yes
checking for i586-mingw32msvc-gcc option to accept ANSI C...
...
~/amhello-1.0 % make
...
~/amhello-1.0 % cd src; file hello.exe
hello.exe: MS Windows PE 32-bit Intel 80386 console executable not relocatable
```

Обычно при кросс-компиляции требуется указать опции --host и --build. Единственным исключением является случай сборки кросс-компилятора, где должна использоваться ещё одна опция.

--target=TARGET
При сборке инструментов кросс-компиляции задаёт систему, для которой будет предназначен инструмент. Например, при установке GCC (GNU Compiler Collection) можно использовать --target=TARGET для указания сборки GCC как кросс-компилятора для TARGET. Комбинация --build и --target указывает кросс-компиляцию кросс-компилятора. Обычно это называют канадской кросс-компиляцией (Canadian cross).

2.2.9. Переименование программ при установке

Система сборки GNU позволяет автоматически переименовывать исполняемые файлы и страницы man перед их установкой. Это особенно удобно при установке пакетов GNU в системы, где уже есть иная реализация пакета, которую нужно сохранить. Например, можно установить GNU tar как gtar, если в системе уже имеется tar. Для решения этой задачи имеется три опции.

--program-prefix=PREFIX
Добавляет префикс PREFIX к именам устанавливаемых программ.
--program-suffix=SUFFIX
Добавляет суффикс SUFFIX к именам устанавливаемых программ.
--program-transform-name=PROGRAM
Запускает команду sed PROGRAM для имён. установленных программ.

Приведённые ниже команды будут устанавливать hello как /usr/local/bin/test-hello.

```
~/amhello-1.0 % ./configure --program-prefix test-
...
~/amhello-1.0 % make
...
~/amhello-1.0 % sudo make install
...
```

2.2.10. Сборка двоичных пакетов с использованием DESTDIR

В системе сборки GNU команды make install и make uninstall не вполне соответствуют потребностям системных администраторов, устанавливающих или обновляющих пакеты на многочисленных хостах. Словом, GNU Build System не является заменой менеджеру пакетов. Таким менеджерам нужно знать, какие файлы устанавливаются пакетом, поэтому make install им не подходит.

Можно использовать переменную DESTDIR для поэтапной установки. Пакет следует настраивать для установки в нужное место (например, --prefix /usr), но при запуске make install указывать в переменной DESTDIR абсолютное имя каталога для копирования устанавливаемых файлов. В этом каталоге можно легко просмотреть установленные файлы,

а затем скопировать в нужное место. Например, можно создать двоичный пакет с «моментальным» снимком (snapshot) всех устанавливаемых файлов.

```
~/amhello-1.0 % ./configure --prefix /usr
...
~/amhello-1.0 % make
...
~/amhello-1.0 % make DESTDIR=$HOME/inst install
...
~/amhello-1.0 % cd ~/inst
~/inst % find . -type f -print > ../files.lst
~/inst % tar zcvf ~/amhello-1.0-i686.tar.gz `cat ../files.lst`
./usr/bin/hello
./usr/share/doc/amhello/README
```

После этого файл amhello-1.0-i686.tar.gz будет готов для распаковки в корневой каталог (/) на разных хостах. Использование `cat ../files.lst` вместо . в качестве аргумента tar предотвращает изменение подкаталогов (не нужно менять время изменения каталогов /, /usr/ и т. п.).

Отметим, что при сборке пакетов для нескольких архитектур может потребоваться использование команд make install-data и make install-exec (2.2.7. Раздельная установка).

2.2.11. Подготовка дистрибутива

Выше уже была упомянута команда make dist, позволяющая собрать все нужные исходные файлы и части системы сборки для создания архива PACKAGE-VERSION.tar.gz. Более полезной командой является make distcheck, создающая архив PACKAGE-VERSION.tar.gz как dist, но обеспечивающая дополнительные операции.

- попытка полной компиляции пакета, распаковки недавно созданных архивов, запуск команд make, make check, make install, а также make installcheck и make dist;
- тестирование сборки VPATH для доступных лишь на чтение исходных кодов (2.2.6. Параллельная сборка (VPATH));
- гарантирует, что команды make clean, make distclean и make uninstall не пропустят файлов;
- проверка установки с DESTDIR (2.2.10. Сборка двоичных пакетов с использованием DESTDIR).

Все эти действия выполняются во временном каталоге, поэтому полномочий root не требуется. Отметим, что местоположение и структура этого каталога (размещение временных файлов, именование временных каталогов, уровень вложенности и т. п.) определяются реализацией и могут меняться.

Выпуск пакета с отказом make distcheck означает, что какое-то из перечисленных выше действий не работает. Поэтому хорошим тоном является выпуск пакетов лишь после успешного прохождения make distcheck. Это не говорит о безупречности пакета, но позволяет надеяться на меньшее число ошибок.

2.2.12. Автоматическая проверка зависимостей

Проверка зависимостей выполняется как побочный результат компиляции. При каждой сборке исходного кода система сборки рассчитывает список зависимостей (в C имеются заголовочные файлы, включаемые в собираемый код). Позднее при запуске команды make для зависимостей, в которых файлы изменились, выполняется повторная сборка.

Automake по умолчанию создаёт код для автоматического отслеживания зависимостей, если явно не указано иное. При выполнении сценария configure, можно видеть проверку зависимостей каждым компилятором (разные механизмы).

```
~/amhello-1.0 % ./configure --prefix /usr
...
checking dependency style of gcc... gcc3
...
```

Поскольку расчёт зависимостей является лишь побочным результатом компиляции, при первой сборке пакета сведений о зависимостях не будет. Поэтому при первой сборке отслеживание зависимостей не имеет смысла и его можно отключить.

--disable-dependency-tracking

Ускоряет первоначальную сборку.

Некоторые компиляторы не предлагают практического способа получить список зависимостей в качестве побочного результата компиляции и требуется отдельный запуск (или другой инструмент) для отслеживания зависимостей. Снижение производительности за счёт этого достаточно велико, чтобы по умолчанию эти методы были отключены. При необходимости их можно включить опцией --enable-dependency-tracking сценария configure.

--enable-dependency-tracking

Не отключать медленное отслеживание зависимостей.

2.2.13. Вложенные пакеты

Автоматическая настройка Autotools может работать с пакетами, содержащими в себе другие пакеты, настраиваемые таким же способом. Предположим, что пакет A содержит библиотеку в одном из своих каталогов и эта библиотека B является полным пакетом со своей системой сборки GNU. Сценарий configure пакета A будет запускать configure пакета B в процессе своей работы. В результате будут собраны и установлены оба пакета A и B.

Таким способом можно собрать несколько пакетов в один приём. В GCC это используется широко, позволяя установщикам настраивать, собирать и устанавливать множество пакетов сразу, а разработчикам - независимо создавать такие пакеты.

При настройке вложенных пакетов опции configure верхнего уровня рекурсивно передаются вложенным сценариям configure. Если пакет не понимает ту или иную опцию, он просто игнорирует её, предполагая, что она предназначена для другого пакета.

Опции, поддерживаемые всеми включёнными пакетами можно увидеть по команде configure --help=recursive.

2.3. Пакет Autotools

Пакет GNU Autotools выполняет значительную часть работы, требуемой для настройки и сборки пакета, создавая переносимую, полную и самодостаточную систему сборки GNU с помощью простых инструментов. Самодостаточность означает, что подготовленной системе сборки уже не требуется GNU Autotools.

Однако в некоторых случаях использовать Autotools не удастся (например, при использовании несовместимой системы сборки). Инструменты Autotools будут работать лишь при поддержке GNU Build System.

2.4. Сборка небольшого пакета

В этом разделе описано создание пакета amhello-1.0 с нуля.

2.4.1. Создание amhello-1.0.tar.gz

Рассмотрим процесс создания архива amhello-1.0.tar.gz. Пакет достаточно прост и включает лишь 5 файлов. Если не хочется набирать текст этих файлов, можно воспользоваться архивом amhello-1.0.tar.gz из пакета Automake.

- Файл src/main.c содержит код программы hello. Он помещён в каталог src/, поскольку по мере создания проекта будут организованы каталоги man/ для руководства, data/ - для данных и т. п.

```
~/amhello % cat src/main.c
#include <config.h>
#include <stdio.h>

int
main (void)
{
  puts ("Hello World!");
  puts ("This is " PACKAGE_STRING ".");
  return 0;
}
```

- Файл README содержит краткое описание нашего небольшого пакета.

```
~/amhello % cat README
This is a demonstration package for GNU Automake.
Type 'info Automake' to read the Automake manual.
```

- Файлы Makefile.am и src/Makefile.am содержат инструкции Automake для двух каталогов.

```
~/amhello % cat src/Makefile.am
bin_PROGRAMS = hello
hello_SOURCES = main.c
~/amhello % cat Makefile.am
SUBDIRS = src
dist_doc_DATA = README
```

- Файл configure.ac содержит инструкции Autoconf для создания сценария configure.

```
~/amhello % cat configure.ac
AC_INIT([amhello], [1.0], [bug-automake@gnu.org])
AM_INIT_AUTOMAKE([-Wall -Werror foreign])
AC_PROG_CC
AC_CONFIG_HEADERS([config.h])
AC_CONFIG_FILES([
 Makefile
 src/Makefile
])
AC_OUTPUT
```

Когда эти 5 файлов созданы, можно запустить Autotools для организации системы сборки. Для этого служит команда autoreconf.

```
~/amhello % autoreconf --install
configure.ac: installing './install-sh'
configure.ac: installing './missing'
configure.ac: installing './compile'
src/Makefile.am: installing './depcomp'
```

На этом создание системы сборки завершается.

В дополнение к трём указанным в выводе autoreconf сценариям будут созданы файлы configure, config.h.in, Makefile.in и src/Makefile.in. Три последних файла являются шаблонами, на основе которых сценарий configure создаст файлы config.h, Makefile и src/Makefile. Запустим этот сценарий.

```
~/amhello % ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
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 suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating config.h
config.status: executing depfiles commands
```

Файлы Makefile, src/Makefile и config.h созданы и можно начать сборку. Например,

```
~/amhello % make
...
~/amhello % src/hello
Hello World!
```

```
This is amhello 1.0.
~/amhello % make distcheck
...
==========================================
amhello-1.0 archives ready for distribution:
amhello-1.0.tar.gz
==========================================
```

Отметим, что команда autoreconf нужна лишь при отсутствии GNU Build System. Если позднее изменить инструкции в Makefile.am или configure.ac, соответствующие части системы сборки будут автоматически обновлены при выполнении команды make.

Сценарий autoreconf вызывает autoconf, automake и выполняет множество других команд в нужном порядке. Команда autoconf создаёт файл configure на основе configure.ac, а automake - файлы Makefile.in из Makefile.am и configure.ac.

2.4.2. Файл configure.ac

Рассмотрим содержимое файла configure.ac.

```
AC_INIT([amhello], [1.0], [bug-automake@gnu.org])
AM_INIT_AUTOMAKE([-Wall -Werror foreign])
AC_PROG_CC
AC_CONFIG_HEADERS([config.h])
AC_CONFIG_FILES([
 Makefile
 src/Makefile
])
AC_OUTPUT
```

Этот файл читает команда autoconf (для создания configure) и automake (для создания файлов Makefile.in). В файле содержится последовательность макросов M4, которые преобразуются в код оболочки (shell) для сценария configure. Синтаксис файла описан в руководстве Autoconf.

Макросы с префиксом AC_ относятся к Autoconf, а с AM_ - к Automake и описаны в соответствующих руководствах.

Первые две строки configure.ac инициализируют Autoconf и Automake. Макрос AC_INIT принимает в качестве параметров имя пакета, номер версии и адрес для отправки сообщений об ошибках (этот адрес выводится в конце по команде ./configure --help). При создании своих пакетов следует указывать корректный адрес. Аргументом AM_INIT_AUTOMAKE служит список опций automake (17. Смена поведения Automake). Опции -Wall и -Werror запрашивают у automake вывод всех предупреждений как ошибок. Речь идёт о предупреждениях Automake, таких как сомнительные инструкции в Makefile.am. Это не имеет отношения к вызовам компилятора, даже если он поддерживает опции с такими именами. Использование -Wall -Werror безопасно на начальном этапе работы с пакетом и позволяет увидеть все проблемы. Позднее можно сузить вывод предупреждений. Опция foreign говорит Automake, что пакет не следует стандартам GNU. В пакеты GNU всегда следует включать такие файлы как ChangeLog, AUTHORS и т. п. Наличие этой опции указывает automake, что не следует выводить предупреждений об отсутствии этих файлов.

Строка AC_PROG_CC заставляет сценарий configure найти компилятор C и указать в переменной CC его имя. Файл src/Makefile.in, созданный Automake, использует переменную CC для сборки hello. Когда сценарий configure создаёт файл src/Makefile из src/Makefile.in, он будет включать в файл найденное значение CC. Если у Automake запрошено создание файла Makefile.in, использующего CC, но в configure.ac не определена эта строка, будет предложено добавить вызов AC_PROG_CC.

Вызов AC_CONFIG_HEADERS([config.h]) заставляет сценарий configure создать файл config.h, собирая операторы #define, заданные другими макросами в configure.ac. В нашем случае некоторые из них уже определены в макросе AC_INIT. Ниже приведён фрагмент файла config.h после работы сценария configure.

```
...
/* Define to the address where bug reports for this package should be sent. */
#define PACKAGE_BUGREPORT "bug-automake@gnu.org"

/* Define to the full name and version of this package. */
#define PACKAGE_STRING "amhello 1.0"
...
```

Файл src/main.c включает config.h и поэтому может использовать PACKAGE_STRING. В реальных проектах файл config.h может быть большим с операторами #define для каждого свойства, проверенного в системе.

Макрос AC_CONFIG_FILES объявляет список файлов, которые должен создать сценарий configure по шаблонам *.in. Automake также просматривает этот список в поиске файлов Makefile.am, которые нужно обработать. Об этом важно помнить при добавлении каталогов в проект, указывая в списке соответствующие файлы Makefile.

Строка AC_OUTPUT является завершающей командой, которая фактически отвечает за создание файлов, зарегистрированных в AC_CONFIG_HEADERS и AC_CONFIG_FILES.

В новых проектах можно начать с этого простого файла configure.ac и добавить в него другие нужные проверки. Команда autoscan может предложить некоторые проверки, которые могут потребоваться для пакета.

2.4.3. Файл Makefile.am

Файл src/Makefile.am содержит инструкции Automake для сборки и установки программы hello.

```
bin_PROGRAMS = hello
hello_SOURCES = main.c
```

Синтаксис Makefile.am не отличается от синтаксиса Makefile. Когда automake обрабатывает Makefile.am этот файл целиком копируется в выходной файл Makefile.in, на основе которого затем создаётся файл Makefile сценарием configure. При этом для некоторых определений переменных создаются правила сборки и другие переменные. Зачастую файлы Makefile.am содержат лишь список переменных, показанных выше, но могут включать также другие определения переменных и правил, которые automake будет пропускать без обработки.

Переменные с суффиксом _PROGRAMS перечисляют программы, которые следует собрать с помощью финального файла Makefile. На языке Automake переменная _PROGRAMS называется первичной (primary). Другими первичными переменными являются _SCRIPTS, _DATA, _LIBRARIES и т. п., соответствующие разным типам файлов.

Префикс bin в bin_PROGRAMS говорит automake, что полученную программу следует установить в BINDIR. Система сборки GNU использует набор переменных для указания целевых каталогов и позволяет пользователям управлять ими (2.2.3. Переменные для стандартных каталогов). Любую из таких переменных можно указать в качестве префикса для primary (без dir в конце), чтобы сказать automake, куда нужно поместить файлы.

Программы собираются из файлов исходного кода, поэтому для каждой программы PROG, указанной в переменной _PROGRAMS, automake будет искать другую переменную с именем PROG_SOURCES, указывающую исходные файлы (их может быть много для последующей компиляции и компоновки).

Automake также знает, что исходные файлы нужно включить в создаваемый архив (в отличие от сборки). Поэтому побочным результатом объявления hello_SOURCES является включение файла main.c в архив по команде make dist.

На верхнем уровне файл Makefile.am включает дополнительные определения.

```
SUBDIRS = src
dist_doc_DATA = README
```

Специальная переменная SUBDIRS указывает все каталоге, которые команде make следует просмотреть перед обработкой текущего каталога. Таким образом, эта строка отвечает за сборку src/hello даже при запуске make из каталога верхнего уровня. Она также заставляет команду make install установить src/hello перед установкой README.

Строка dist_doc_DATA = README включает распространение файла README и его установку в DOCDIR. Файлы, указанные в _DATA, не становятся автоматически частью архива, создаваемого командой make dist, поэтому для их распространения используется префикс dist_. Однако для файла README это не требуется и automake будет автоматически распространять файл README (список автоматически распространяемых файлов можно увидеть по команде automake --help).

3. Основные идеи

3.1. Базовые операции

Automake читает файлы Makefile.am и создаёт Makefile.in. Некоторые переменные и правила из Makefile.am указывают Automake генерировать более специализированный код. Например, определение bin_PROGRAMS ведёт к генерации правил для компиляции и компоновки программ.

Определения переменных и правила в основном копируются дословно из Makefile.am в создаваемый файл и определения переменных помещаются до правил. Это позволяет добавлять практически любой код в создаваемый файл Makefile.in. Например, дистрибутив Automake включает нестандартное правило для цели git-dist, которую сопровождающий Automake использует для создания дистрибутивов из системы контроля версий.

Отметим, что большинство расширений GNU make не распознаётся Automake и использование таких расширений в Makefile.am ведёт к ошибкам или путаному поведению. Особым исключением является поддержка оператора добавления в конце +=, который добавляет указанный справа аргумент в конец заданной слева переменной. Automake транслирует этот оператор в обычный оператор =, поэтому += работает с любой программой make.

Automake пытается сохранить комментарии, сгруппированные с примыкающими правилами или определениями переменных.

В общем случае Automake не очень хорошо разбирает необычные конструкции Makefile, поэтому рекомендуется избегать таких конструкций, а также «творческого» использования пробелов. Например, символы <TAB> не могут помещаться между именем цели и следующим символом :, а в назначениях переменных не следует применять <TAB> для отступа. Использование сложных макросов в именах целей может создавать проблемы. Например,

```
% cat Makefile.am
$(FOO:=x): bar
% automake
Makefile.am:1: bad characters in variable name '$(FOO'
Makefile.am:1: ':='-style assignments are not portable
```

Правило в Makefile.am обычно переопределяет правило с похожим именем, автоматически созданное программой automake. Хотя это поддерживаемая функция, её лучше избегать, поскольку иногда сгенерированные правила очень специфичны..

Точно так же переменная, определённая в Makefile.am или подставленная AC_SUBST из configure.ac переопределяет любую переменную, которую обычно создаёт automake. Эта функция более полезна, чем возможность переопределения правила. Следует помнить, что многие переменные, созданные automake, предназначены лишь для внутреннего использования и их имена могут измениться в будущих выпусках.

При проверке определения переменной Automake рекурсивно проверяет упомянутые в определении переменные. Например, увидев содержимое foo_SOURCES в приведённом фрагменте

```
xs = a.c b.c
foo_SOURCES = c.c $(xs)
```

Automake будет использовать файлы a.c, b.c и c.c как содержимое foo_SOURCES.

Automake также допускает форму комментария, которая не копируется на выход. Строки, начинающиеся с символов ## (допускаются пробелы в начале), Automake полностью игнорирует. Первая строка Makefile.am обычно содержит текст

```
## Process this file with automake to produce Makefile.in
```

3.2. Строгость

Хотя программа Automake предназначена для сопровождающих пакеты GNU, она прилагает усилия по адаптации к требованиям тех, кто хочет её использовать, но не готов следовать всем соглашениям GNU. Для этого в Automake поддерживается три уровня строгости, определяющие уровень проверки в Automake соответствия стандартам.

foreign

Automake проверяет лишь те элементы, которые абсолютно необходимы для корректной работы. Например, стандарты GNU требуют наличия файла NEWS, в этом режиме файл не проверяется. На этом уровне также

отключены некоторые предупреждения, используемые по умолчанию (например, по части переносимости). Название режима показывает, что он относится к программам, не совсем соответствующим стандартам GNU.

gnu

Automake будет по возможности проверять соответствие стандартам GNU. Режим принят по умолчанию.

gnits

Automake проверяет соответствие ещё не написанным «стандартам Gnits», основанным на стандартах GNU, но более детальных. Рекомендуется не использовать этот режим, не будучи разработчиком стандартов Gnits, пока стандарт не опубликован.

3.3. Схема именования

Переменные Automake обычно следуют схеме унифицированного именования (uniform naming scheme), которая упрощает решение вопросов о сборке и установке программ (и производных элементов). Эта схема также позволяет определить время запуска configure для выбора собираемых компонент.

Во время работы make некоторые переменные используются для определения объектов, которые нужно собрать. Имена переменных собираются из нескольких частей путём конкатенации. Часть, информирующую automake о том, что нужно собирать, называют первичной (primary). Например, первичная переменная PROGRAMS содержит список программ, которые нужно компилировать и компоновать.

Для определения мест установки применяется другой набор имён., которые служат префиксами для primary и указывают стандартные каталоги (имена их заданы в стандартах GNU - 2.2.3. Переменные для стандартных каталогов). Automake расширяет этот список переменными pkgdatadir, pkgincludedir, pkglibdir и pkglibexecdir, в которые добавлен суффикс $(PACKAGE). Например, pkglibdir определяется как $(libdir)/$(PACKAGE).

Для каждого первичного имени имеется дополнительная переменная с префиксом EXTRA_, которая служит для перечисления объектов, сборка которых не обязательная и определяется сценарием configure. Эти переменные нужны , поскольку Automake требуется заранее знать список объектов, которые могут быть собраны, для генерации файла Makefile.in, который будет работать в всех случаях. Например, cpio выбирает собираемые программы во время выполнения configure. Некоторые программы устанавливаются в bindir, а другие - в sbindir, как показано ниже.

```
EXTRA_PROGRAMS = mt rmt
bin_PROGRAMS = cpio pax
sbin_PROGRAMS = $(MORE_PROGRAMS)
```

Указание первичной переменной без префикса (например, PROGRAMS) приведёт к ошибке. Отметим, что базовый суффикс dir не включается в имена, т. е. используется переменная bin_PROGRAMS, а не bindir_PROGRAMS.

К каждом из каталогов можно размещать лишь определённые объекты. Automake будет фиксировать попытки некорректного размещения и сообщать об ошибках (однако имеются способы обхода этого запрета). Кроме того, Automake замечает опечатки в именах стандартных каталогов.

Иногда стандартных каталогов недостаточно, даже с расширениями Automake. В частности, иногда нужно поместить файлы в подкаталог определённого стандартного каталога. Для этого Automake позволяет расширять список возможных каталогов установки. Заданный префикс (например, zar) будет действительным, если определена переменная с таким именем и суффиксом dir (например, zardir). Ниже приведён пример установки файла file.xml в каталог $(datadir)/xml.

```
xmldir = $(datadir)/xml
xml_DATA = file.xml
```

Это свойство можно использовать для переопределения проверок работоспособности, выполняемых Automake для обнаружения подозрительных каталогов и привязок. Например Automake может возвращать ошибку, показанную ниже.

```
# Forbidden directory combinations, automake will error out on this.
pkglib_PROGRAMS = foo
doc_LIBRARIES = libquux.a
```

Но эта ошибка исчезнет в приведённом ниже варианте.

```
# Work around forbidden directory combinations. Do not use this
# without a very good reason!
my_execbindir = $(pkglibdir)
my_doclibdir = $(docdir)
my_execbin_PROGRAMS = foo
my_doclib_LIBRARIES = libquux.a
```

Подстрока exec в переменной my_execbindir позволяет установить файлы в нужное время (2.2.7. Раздельная установка). Специальный префикс noinst_ указывает, что объект следует собрать, но не нужно устанавливать. Обычно это применяется для объектов, которые нужны для сборки остальной части пакета (например, библиотеки или вспомогательного сценария). Специальный префикс check_ указывает, что объекты не следует собирать, пока не используется команда make check. Такие объекты не устанавливаются совсем.

К основным именам переменных относятся PROGRAMS, LIBRARIES, LTLIBRARIES, LISP, PYTHON, JAVA, SCRIPTS, DATA, HEADERS, MANS и TEXINFOS. Некоторые из этих имён. допускают префиксы, управляющие другими аспектами поведения automake. К таким префиксам относятся dist_, nodist_, nobase_ и notrans_ (8.4. Переменные для программ и библиотек).

3.4. Ограничение размера команд

В большинстве unix-подобных систем размер аргументов команды и содержимого окружения при создании новых процессов ограничены (см., например, http://www.in-ulm.de/~mascheck/various/argmax/), что применимо и к командам, выполняемым make. POSIX требует, чтобы предельный размер составлял не менее 4096 байтов и в большинстве современных систем предельный размер достаточно велик или неограничен.

Для создания переносимых файлов Makefile, в которых не нарушается это ограничение, нужно ограничивать размер списков. К сожалению это не совсем прозрачно в Automake и могут потребоваться дополнительные действия. Обычно для этого достаточно вручную разделять списки и использовать для каждого своё имя каталога установки. Например,

```
data_DATA = file1 ... fileN fileN+1 ... file2N
```

можно записать в форме

```
data_DATA = file1 ... fileN
data2dir = $(datadir)
data2_DATA = fileN+1 ... file2N
```

и Automake будет обрабатывать отдельно два списка при выполнении команды make install. Выбор имён. каталогов для раздельной установки описан в параграфе 2.2.7. Раздельная установка. Отметим, что команда make dist по-прежнему будет работать лишь в средах с большим размером строки команд.

В Automake применяется несколько стратегий предотвращения длинных строк. Например, при добавлении к именам файлов префикса ${srcdir}/ может возникать ситуация, подобная упомянутым выше спискам $(data_DATA) и ограничивается число аргументов, передаваемых внешним командам.

К сожалению в некоторых системах команды make могут автоматически использовать префиксы VPATH, такие как ${srcdir}/, для имён. файлов в дереве исходных кодов. В таких случаях пользователю имеет смысл перейти на работу с GNU Make или воздержаться от сборок VPATH.

Для программ и библиотек, собираемых из множества источников можно применять удобные промежуточные архивы для сокращения размера элементов (8.3.5. Вспомогательные библиотеки Libtool).

3.5. Именование производных переменных

Иногда имена переменных Makefile выводятся из текста, предоставленного сопровождающим. Например, имя программы в _PROGRAMS может переопределяться в переменную _SOURCES. В таких случаях Automake «канонизирует» текст. Все символы в именах за исключением букв, цифр @ и _ превращаются в _ при создании ссылок на переменные. Например, для программы sniff-glue производным именем станет sniff_glue_SOURCES, а не sniff-glue_SOURCES, а исходные коды библиотеки libmumble++.a получат имя libmumble___a_SOURCES.

3.6. Пользовательские переменные

Некоторые переменные Makefile зарезервированы стандартами кодирования GNU для пользователя (сборщика пакета). Одним из примеров служит переменная CFLAGS.

Иногда разработчики пытаются установить такие переменные, как CFLAGS, для упрощения своей работы. Однако самому пакету не следует устанавливать пользовательские переменные, в частности для того, чтобы не влиять на переключатели, используемые при компиляции пакета. Поскольку эти переменные указаны как предназначенные для разработчика, тот вправе ожидать, что он может переопределить любую из них при сборке.

Для обхода этой проблемы Automake использует «теневую» переменную для каждой переменной пользовательских флагов. Теневые переменные не применяются для таких переменных, как CC, где они не имеют смысла. Теневые переменные используют префикс AM_ перед именем пользовательской переменной. Например, теневой переменной для YFLAGS будет AM_YFLAGS. Сопровождающий пакет человек - автор файлов Makefile.am и configure.ac - может настроить теневые переменные при необходимости.

3.7. Программы, которые могут быть нужны automake

Automake иногда нужны вспомогательные программы, чтобы созданные файлы Makefile могли работать корректно. Таких программ достаточно много и они перечислены ниже. Все эти файлы распространяются с Automake, но некоторые поддерживаются независимо. Automake копирует обновления перед каждым выпуском, но программы могут обновляться в интервалах между выпусками.

ar-lib
: Эта утилита делает архиватор Microsoft lib более похожим на POSIX.

compile
: Это оболочка для компиляторов, не принимающих одновременно опции -c и -o, используемая лишь при безысходности. Такие компиляторы редки и наиболее заметным является Microsoft C/C++ Compiler. Оболочка при необходимости также делает для компилятора доступными при преобразовании имён. файлов опции общего назначения -I, -L, -l, -Wl и -Xlinker.

config.guess
config.sub
: Эти две программы определяют канонические триплеты архитектуры build, host и target. Программы обновляются регулярно для поддержки новых архитектур и устранения проблем, возникающих при смене версии ядра. В каждый новый выпуск Automake включаются актуальные копии программ. При необходимости можно получить новые версии по ссылке.

depcomp
: Эта программа понимает, как работает компилятор и будет генерировать не только желаемый вывод, но и данные, используемые при автоматическом контроле зависимостей (8.19. Автоматическое отслеживание зависимостей).

install-sh
: Замена программы install для работы на платформах, где та не доступна или не работает.

mdate-sh
: Сценарий для генерации файла version.texi, создающий файл и выводящий информацию о его дате.

missing
: Оболочка для множества программ, нужных сопровождающему. Если нужной программы нет или она кажется устаревшей, missing будет выдавать предупреждение.

mkinstalldirs
: Этот сценарий служит оболочкой для команды mkdir -p, которая не является переносимой. Сейчас применяется команда install-sh -d, когда configure находит неработающую команду mkdir -p. Для совместимости со старыми версиями продолжается использование mkinstalldirs и программа распространяется, если automake находит её в пакете. Но она больше не устанавливается автоматически и может быть безопасно удалена.

py-compile
: Служит для байтовой компиляции сценариев Python.

test-driver
: Реализует используемый по умолчанию драйвер тестов.

texinfo.tex

Это не программа, а файл, нужный для команд make dvi, make ps и make pdf при работе с исходными файлами Texinfo. Свежая версия доступна по ссылке.

ylwrap

Программа обеспечивает для lex и yacc переименование выходных файлов, а также возможность параллельной работы нескольких экземпляров yacc в одном каталоге.

4. Примеры пакетов

В первом из рассматриваемых здесь примеров предполагается наличие проекта, использующего Autoconf с созданными вручную файлами Makefile, которые нужно преобразовать с помощью Automake. Здесь уместно обратиться к рассмотренному выше примеру (2.4. Сборка небольшого пакета).

Во втором примере показано, как можно собрать две программы из одного файла с использованием разных параметров компиляции.

4.1. Простой пример

Предположим, что завершена разработка программы zardoz, где применяется Autoconf для обеспечения переносимости, но файлы Makefile.in имеют особенности. Для обеспечения их переносимости применяется Automake.

Сначала нужно обновить файл configure.ac путём включения нужных automake команд. Для этого добавляется вызов AM_INIT_AUTOMAKE сразу после AC_INIT.

```
AC_INIT([zardoz], [1.0])
AM_INIT_AUTOMAKE
...
```

Поскольку с программой не связано каких-либо осложнений (например, использование gettext или нежелание создавать общую библиотеку), на этом можно закончить.

Затем нужно заново создать сценарий configure, для чего нужно сказать autoconf, как найти новый макрос, который был использован. Простейшим путём является использование программы aclocal для создания файла aclocal.m4. Этот файл уже может существовать, поскольку для программы были созданы макросы. Программа aclocal позволяет поместить ваши макросы в файл acinclude.m4, который можно просто переименовать и выполнить команды

```
mv aclocal.m4 acinclude.m4
aclocal
autoconf
```

Сейчас самое время создать файл Makefile.am для пакета zardoz. Поскольку zardoz является пользовательской программой, её уместно поместить в каталог bindir. Кроме того, zardoz имеет документацию в формате Texinfo. Сценарий configure.ac использует AC_REPLACE_FUNCS, поэтому нужна привязка к $(LIBOBJS).

```
bin_PROGRAMS = zardoz
zardoz_SOURCES = main.c head.c float.c vortex9.c gun.c
zardoz_LDADD = $(LIBOBJS)

info_TEXINFOS = zardoz.texi
```

После этого можно ввести команду automake --add-missing для создания файла Makefile.in и вспомогательных файлов.

4.2 Сборка двух программ из одного файла

Следующий пример несколько сложнее. Он показывает, как создать две программы (true и false) из одного исходного файла (true.c). Сложность состоит в том, что для каждой компиляции true.c применяются свои флаги cpp.

```
bin_PROGRAMS = true false
false_SOURCES =
false_LDADD = false.o

true.o: true.c
        $(COMPILE) -DEXIT_CODE=0 -c true.c

false.o: true.c
        $(COMPILE) -DEXIT_CODE=1 -o false.o -c true.c
```

Отметим, что здесь нет определения true_SOURCES, поскольку Automake неявно предполагает наличие исходного файла true.c (8.5. Принятые по умолчанию значения _SOURCES) и задаёт правила для компиляции true.o и компоновки true. Приведённое выше правило true.o: true.c из Makefile.am будет переопределено созданным Automake правилом для сборки true.o.

Переменная false_SOURCES определена пустой, т. е. неявное значение не подставляется. Поскольку источник false не указан, нужно сказать Automake, как компоновать программу, для чего служит строка false_LDADD. Переменная false_DEPENDENCIES, указывающая зависимости для цели false, будет создана Automake автоматически на основе содержимого false_LDADD.

Приведённые выше правила не будут работать, если компилятор не поддерживает опции -c и -o вместе. Простейшим способом обойти это является использование фиктивной зависимости (чтобы не было проблем при параллельной работе make).

```
true.o: true.c false.o
        $(COMPILE) -DEXIT_CODE=0 -c true.c

false.o: true.c
        $(COMPILE) -DEXIT_CODE=1 -c true.c && mv true.o false.o
```

Однако есть и более простой способ на основе флагов компиляции для каждой программы, как показано ниже.

```
bin_PROGRAMS = false true

false_SOURCES = true.c
false_CPPFLAGS = -DEXIT_CODE=1

true_SOURCES = true.c
true_CPPFLAGS = -DEXIT_CODE=0
```

В этом случае Automake приведёт к двухкратной компиляции true.c с разными флагами. В примере имена объектных файлов будут выбраны automake (false-true.o и true-true.o). Обычно имена таких файлов не имеют значения.

5. Создание Makefile.in

Для создания всех файлов Makefile.in в пакет следует использовать программу automake без аргументов из каталога верхнего уровня. Программа automake автоматически найдёт все подходящие файлы Makefile.am (просматривая configure.ac) и создаст соответствующие файлы Makefile.in. Отметим, что automake имеет достаточно упрощённое представление о содержимом пакета и предполагает наличие файла configure.ac лишь на верхнем уровне. Если в пакете имеется несколько файлов configure.ac, нужно запустить automake из каждого каталога, где есть такой файл. Другим вариантом является использование программы autoreconf, которая рекурсивно просматривает каталоги дерева пакета и при необходимости запускает automake.

Можно передать программе automake аргумент. К этому аргументу будет добавлен суффикс .am и полученное имя будет использовано в качестве имени входного файла. Это свойство обычно применяется для автоматического обновления устаревших файлов Makefile.in. Отметим, что automake всегда следует выполнять из каталога верхнего уровня в проекте (даже при регенерации Makefile.in в том или ином подкаталоге. Это требуется потому, что automake нужно сканировать файл configure.ac, а также по причине использования automake информации о размещении файлов Makefile.in в подкаталогах для смены поведения в некоторых случаях.

Automake использует программу autoconf для сканирования файла configure.ac и его зависимостей (aclocal.m4 и включённые файлы), поэтому путь к autoconf должен быть включён в переменную PATH. При наличии в среде переменной AUTOCONF, её значение будет использоваться вместо autoconf, что позволяет выбрать конкретную версию Autoconf. Следует отметить, что запуск из automake программы autoconf для сканирования файлов configure.ac не означает создания сценария configure и autoconf для этого придётся запустить явно.

Опции automake перечислены ниже.

-a
--add-missing
: Для Automake требуется в определённых ситуациях наличие некоторых файлов. Например, файл config.guess нужен при вызове из configure.ac макроса AC_CANONICAL_HOST. Automake распространяется с некоторыми из таких файлов (3.7. Программы, которые могут быть нужны automake) и эта опция приведёт к автоматическому добавлению таких программ в пакет, когда это возможно. В общем случае, если Automake говорит об отсутствии файла, следует задать эту опцию. По умолчанию Automake пытается создать символьную ссылку на свою копию отсутствующего файла, что можно изменить с помощью опции --copy.
Многие из потенциально отсутствующих файлов являются сценариями общего назначения, чьё расположение может быть задано макросом AC_CONFIG_AUX_DIR. Поэтому установка AC_CONFIG_AUX_DIR влияет на признание файла отсутствующим и его добавление (6.2. Дополнительные макросы). В некоторых режимах строгости устанавливаются дополнительные файлы (22. Опции --gnu и --gnits).

--libdir=DIR
: Задаёт поиск файлов данных Automake в каталоге DIR вместо каталога установки (обычно служит для отладки). Переменная среды AUTOMAKE_LIBDIR обеспечивает другой способ задания каталога с файлами данных Automake, однако приоритет --libdir выше.

--print-libdir
: Выводит путь к установочному каталогу, представленных Automake сценариев и файлов данных (например, texinfo.texi и install-sh).

-c
--copy
: При использовании с опцией --add-missing обеспечивает копирование установленных файлов вместо создания символьных ссылок.

-f
--force-missing
: При использовании с опцией --add-missing обеспечивает повторную установку стандартных файлов даже при их наличии в дереве кода.

--foreign
: Устанавливает глобально уровень строгости foreign (3.2. Строгость).

--gnits
: Устанавливает глобально уровень строгости gnits (22. Опции --gnu и --gnits).

--gnu
: Устанавливает глобально принятый по умолчанию уровень строгости (22. Опции --gnu и --gnits).

--help
: Выводит краткую справку по опциям командной строки.

-i
--ignore-deps
: Отключает отслеживание зависимостей в создаваемых Makefile (8.19. Автоматическое отслеживание зависимостей).

--include-deps
: Включает отслеживание зависимостей. Опция включена по умолчанию и сохранена по историческим причинам.

--no-force
: Исходно automake создаёт все файлы Makefile.in, упомянутые в configure.ac. Эта опция позволяет обновить лишь файлы, устаревшие в соответствии с какой-либо из их зависимостей.

-o DIR
--output-dir=DIR
: Задаёт размещение созданных Makefile.in в каталоге DIR. Исходно каждый Makefile.in создаётся в каталоге с соответствующим файлом Makefile.am. Опция устарела и будет удалена.

-v
--verbose
: Заставляет Automake выводить информацию о читаемых и создаваемых файлах.

--version
Выводит номер версии Automake и завершает работу.

-W CATEGORY
--warnings=CATEGORY
Задаёт вывод предупреждений указанной категории (CATEGORY) которая может быть одной из перечисленных.

gnu
предупреждения, связанные с GNU Coding Standards;
obsolete
устаревшие функции или конструкции;
override
пользовательские переопределения правил или переменных Automake;
portability
проблемы переносимости (например, использование не переносимых свойств make);
extra-portability
дополнительные вопросы переносимости, связанные с экзотическими инструментами (например, архиватор Microsoft lib);
syntax
странный синтаксис, неиспользуемые переменные, опечатки;
unsupported
неподдерживаемые или неполные функции;
all
все предупреждения;
none
не выводить никаких предупреждений;
error
считать предупреждения ошибками.

Категорию можно отключить с помощью префикса no-. Например, -Wno-syntax будет отключать предупреждения о неиспользуемых переменных.
По умолчанию выводятся сообщения категорий obsolete, syntax и unsupported. В дополнение к этому категории gnu и portability включаются на уровнях строгости gnu и gnits.
Выключение категории portability выключает опцию extra-portability, а включение extra-portability включает portability. Однако включение portability или выключение extra-portability не влияет на другую категорию.
Переменная среды WARNINGS может включать список разделённых запятыми категорий для вывода предупреждений. Это принимается во внимание перед разбором опций командной строки, а опция -Wnone отключит и категории из переменной WARNINGS. Переменная используется и другими инструментами, такими как autoconf (незнакомые категории игнорируются).
Если переменная среды AUTOMAKE_JOBS содержит положительное значение, оно определяет максимальное число потоков Perl, используемых automake при одновременной генерации файлов Makefile.in.

6. Сканирование configure.ac и использование aclocal

Automake сканирует файл configure.ac для поиска информации о пакете. Некоторые макросы autoconf являются обязательными и ряд переменных необходимо определить в configure.ac. Automake также использует данные из configure.ac для дополнительной настройки вывода.

Automake предоставляет некоторые макросы Autoconf для упрощения поддержки. Эти макросы могут автоматически включаться в файл aclocal.m4 с помощью программы aclocal.

6.1. Конфигурационные требования

Одним из реальных требований Automake является вызов из configure.ac макроса AM_INIT_AUTOMAKE, который выполняет ряд задач, требуемых для корректной работы Automake (6.4. Макросы Autoconf из пакета Automake).

К другим макросам, требуемым Automake, но не исполняемым AM_INIT_AUTOMAKE, относятся AC_CONFIG_FILES и AC_OUTPUT. Обычно два эти макроса вызываются в конце configure.ac.

```
...
AC_CONFIG_FILES([
  Makefile
  doc/Makefile
  src/Makefile
  src/lib/Makefile
  ...
])
AC_OUTPUT
```

Automake использует эти макросы для определения создаваемых файлов. Указанные файлы считаются создаваемыми Automake файлами Makefile.in, если имеется файл с таким же именем и расширением .am. Обычно AC_CONFIG_FILES([foo/Makefile]) будет заставлять Automake генерировать файл foo/Makefile.in при наличии foo/Makefile.am.

При использовании AC_CONFIG_FILES со множеством входных файлов (например, AC_CONFIG_FILES([Makefile:top.in:Makefile.in:bot.in]) automake будет сначала создавать файл .in, для которого имеется .am. Если таких файлов нет, вывод не считается созданным Automake.

Файлы, созданные AC_CONFIG_FILES (Makefile от Automake и прочие), удаляются командой make distclean. Их ввода распространяется автоматически, если он не не является выводом предшествующих команд AC_CONFIG_FILES. Правила пересборки создаются в Automake Makefile, имеющихся в подкаталоге выходного файла (если он есть) или Makefile верхнего уровня.

Описанный выше механизм (очистка, распространение, пересборка)хорошо работает, если AC_CONFIG_FILES содержит лишь литералы. При использовании переменных среды automake не сможет выполнить эту настройку и придётся поработать руками. Например, в случае

```
file=input
...
AC_CONFIG_FILES([output:$file],, [file=$file])
```

automake будет выводить правила для очистки вывода и повторной сборки. Однако правило повторной сборки не будет зависеть от ввода и файл не будет распространён (нужно добавить EXTRA_DIST = input в файл Makefile.am, если input содержит исходный код).

Аналогично

```
file=output
file2=out:in
...
AC_CONFIG_FILES([$file:input],, [file=$file])
AC_CONFIG_FILES([$file2],, [file2=$file2])
```

будет приводить лишь к распространению input. Файлы не будут очищаться автоматически (для этого следует добавить DISTCLEANFILES = output out) и не будет выводиться правило пересборки.

Обычно automake не может предсказать, какое значение будет содержать $file при последующем запуске configure и не может использовать переменную $file в Makefile. Однако при наличии ссылки на $file как ${file} (т. е. совместимым с make способом) и использовании AC_SUBST для обеспечения осмысленности ${file} в Makefile, automake сможет использовать ${file} для создания правил. Например, ниже показано, как сам пакет Automake создаёт сценарии для тестов с учётом версии.

```
AC_SUBST([APIVERSION], ...)
...
AC_CONFIG_FILES(
  [tests/aclocal-${APIVERSION}:tests/aclocal.in],
  [chmod +x tests/aclocal-${APIVERSION}],
  [APIVERSION=$APIVERSION])
AC_CONFIG_FILES(
  [tests/automake-${APIVERSION}:tests/automake.in],
  [chmod +x tests/automake-${APIVERSION}])
```

Очистка, распространение и повторная сборка здесь выполняются автоматически, поскольку значение ${APIVERSION} известно при запуске make.

Отметим, что не следует применять переменные оболочки для объявления файлов Makefile, для которых automake нужно создавать Makefile.in. AC_SUBST здесь не поможет, поскольку automake нужно знать имя файла во время работы, чтобы проверить наличие Makefile.am. В сложной ситуации, когда требуется такое использование переменных, нужно сказать Automake, какие файлы Makefile.in следует генерировать, из строки команды.

Можно разрешить automake выдавать дополнительные правила для AC_CONFIG_FILES с помощью AM_COND_IF (6.2. Дополнительные макросы).

В качестве заключения ниже приведены несколько рекомендаций.

- Используйте по возможности литералы для Makefile и других файлов.
- Используйте $file (или ${file} без AC_SUBST([file])) для файлов, которые automake следует игнорировать.
- Используйте ${file} и AC_SUBST([file]) для файлов, которые automake не следует игнорировать.

6.2. Дополнительные макросы

При каждом запуске Automake программа запускает Autoconf для трассировки файла configure.ac. Таким способом могут распознаваться некоторые макросы и соответствующим образом корректироваться создаваемые файлы Makefile.in. В настоящее время распознаются и работают макросы AC_CANONICAL_BUILD, AC_CANONICAL_HOST, AC_CANONICAL_TARGET. Automake обеспечивает наличие файлов config.guess и config.sub, а также переменных build_triplet, host_triplet и target_triplet в Makefile.

AC_CONFIG_AUX_DIR

Automake будет искать разные вспомогательные сценарии (такие как install-sh, ar-lib, config.guess, config.sub, depcomp, compile, install-sh, ltmain.sh, mdate-sh, missing, mkinstalldirs, py-compile, test-driver, texinfo.tex, ylwrap), в каталоге, заданном при вызове этого макроса. Поиск некоторых сценариев может выполняться лишь в тех случаях, когда они нужны в создаваемом Makefile.in.

Если макрос AC_CONFIG_AUX_DIR не задан, поиск сценариев ведётся в стандартных каталогах. Например, для mdate-sh, texinfo.tex и ylwrap поиск ведётся в каталоге исходного кода, соответствующем текущему файлу Makefile.am. Для остальных сценариев поиск происходит в каталогах ., .., ../.. относительно корневого каталога исходного кода.

Требуемые файлы из AC_CONFIG_AUX_DIR распространяются автоматически даже при отсутствии Makefile.am в данном каталоге.

AC_CONFIG_LIBOBJ_DIR

Automake будет требовать исходные файлы, объявленные с AC_LIBSOURCE в каталоге, заданном этим макросом.

AC_CONFIG_HEADERS

Automake будет генерировать правила для перестроения этих заголовков по соответствующим шаблонам (обычно шаблоном для foo.h служит foo.h.in). Старые версии Automake требовали использования макроса AM_CONFIG_HEADER, но сейчас он не применяется и будет удалён.

Как и в случае AC_CONFIG_FILES (6.1. Конфигурационные требования), части спецификации, использующие переменные оболочки, будут игнорироваться при очистке, распространении и перестройке.

AC_CONFIG_LINKS

Automake будет генерировать правила для удаления созданных configure ссылок по команде make distclean и распространять указанные исходные файлы по команде make dist.

Как и в случае AC_CONFIG_FILES (6.1. Конфигурационные требования), части спецификации, использующие переменные оболочки, будут игнорироваться при очистке и распространении (перестройки здесь нет).

AC_LIBOBJ
AC_LIBSOURCE
AC_LIBSOURCES
Automake будет автоматически распространять любой файл, указанный в AC_LIBSOURCE или AC_LIBSOURCES. Отметим, что макрос AC_LIBOBJ вызывает AC_LIBSOURCE, поэтому при наличии в макросе Autoconf вызова AC_LIBOBJ([file]), file.c будет автоматически распространяться Automake. Это включает множество макросов, подобных AC_FUNC_ALLOCA, AC_FUNC_MEMCMP, AC_REPLACE_FUNCS и др. В результате прямые назначения LIBOBJS больше не поддерживаются и следует использовать AC_LIBOBJ.

AC_PROG_RANLIB
Требуется при наличии в пакете собираемых библиотек.

AC_PROG_CXX
Требуется при включении любых исходных файлов C++.

AC_PROG_OBJC
Требуется при включении любых исходных файлов Objective C.

AC_PROG_OBJCXX
Требуется при включении любых исходных файлов Objective C++.

AC_PROG_F77
Требуется при включении любых исходных файлов Fortran 77.

AC_F77_LIBRARY_LDFLAGS
Требуется для программ и общих библиотек, написанных на нескольких языках с использованием Fortran 77 (Ошибка: источник перекрёстной ссылки не найден). Макросы Autoconf распространяются с пакетом Automake.

AC_FC_SRCEXT
Automake будет добавлять флаги, рассчитанные AC_FC_SRCEXT, при компиляции файлов с соответствующими расширениями имён.

AC_PROG_FC
Требуется при включении любых исходных файлов Fortran 90/95. Макрос включён в Autoconf версии 2.58 и выше.

AC_PROG_LIBTOOL
Automake будет включать обработку для libtool.

AC_PROG_YACC
При наличии исходных файлов Yacc нужно использовать этот макрос или определить переменную YACC в файле configure.ac (предпочтительно).

AC_PROG_LEX
При наличии исходных файлов Lex требуется использовать этот макрос.

AC_REQUIRE_AUX_FILE
Для каждого макроса AC_REQUIRE_AUX_FILE([FILE]) программа automake будет обеспечивать наличие FILE в каталоге aux или «жаловаться». Этот макрос следует применять в сторонних макросах Autoconf, которым нужны файлы поддержки в каталоге aux, заданном AC_CONFIG_AUX_DIR.

AC_SUBST
Первый аргумент автоматически определяется как переменная в каждом создаваемом Makefile.in, если для этой переменной не применяется также AM_SUBST_NOTMAKE.
Для каждой подставленной переменной VAR программа automake будет добавлять строку VAR = VALUE в каждый файл Makefile.in. Многие макросы Autoconf вызывают AC_SUBST для установки таким способом выходных переменных. Например, AC_PATH_XTRA определяет X_CFLAGS и X_LIBS. Таким образом, можно обращаться к этим переменным как $(X_CFLAGS) и $(X_LIBS) в любом файле Makefile.am, если вызывается AC_PATH_XTRA.

AM_CONDITIONAL
Включает условные конструкции в Automake (20. Конструкции с условием).

AM_COND_IF
Позволяет automake обнаруживать последующий доступ в файле configure.ac к условной конструкции, созданной макросом AM_CONDITIONAL, делая возможным условие AC_CONFIG_FILES (20.1. Использование условных конструкций).

AM_GNU_GETTEXT
Этот макрос требуется для пакетов, использующих GNU gettext (10.2. Gettext), и распространяется в пакете gettext. Увидев этот макрос, Automake обеспечивает соответствие пакета некоторым требованиям gettext.

AM_GNU_GETTEXT_INTL_SUBDIR
Этот макрос задаёт создание подкаталога intl/ даже при вызове AM_GNU_GETTEXT с первым аргументом external.

AM_MAINTAINER_MODE([DEFAULT-MODE])
Добавляет опцию --enable-maintainer-mode в команду configure и automake будет отключать по умолчанию правила maintainer-only в создаваемых файлах Makefile.in, если для DEFAULT-MODE не установлено enable. Макрос определяет условие MAINTAINER_MODE, которое можно использовать в файле Makefile.am (27.2. Сценарий missing и режим AM_MAINTAINER_MODE).

AM_SUBST_NOTMAKE(VAR)
Запрещает Automake определять переменную VAR, даже если для неё задана подстановка в config.status. Обычно Automake определяет переменную make для каждой подстановки configure, т. е. для каждого AC_SUBST([VAR]). Этот макрос запрещает такие определения. Если для этой переменной не было вызова AC_SUBST, макрос AM_SUBST_NOTMAKE не будет работать. Запрет определений переменных может быть полезен при подстановке многострочных значений, где VAR = @VALUE@ может давать непредсказуемые результаты.

m4_include
Файлы, включённые в configure.ac с использованием этого макроса, будут обнаруживаться Automake и автоматически распространяться, а также отображаться как зависимости в правилах Makefile.
Макрос m4_include редко применяется в файлах configure.ac, но может присутствовать в aclocal.m4, когда aclocal обнаруживает, что некоторые нужные макросы приходят из локальных файлов пакета, а не из общесистемных файлов.

6.3. Автоматическое создание aclocal.m4

Automake включает множество макросов Autoconf, которые могут применяться в пакете (6.4. Макросы Autoconf из пакета Automake). некоторые из этих макросов реально требуются Automake в определённых ситуациях. Они должны быть определены в файле aclocal.m4, иначе autoconf их не увидит.

Программа aclocal автоматически создаёт файлы aclocal.m4 на основе содержимого configure.ac. Это даёт удобный способ получить макросы Automake без их поиска. Механизм aclocal позволяет другим пакетам предоставлять свои макросы (6.3.3. Создание макросов aclocal). Это можно применять также для поддержки своего набора макросов (6.3.4. Обработка локальных макросов).

При запуске aclocal сканирует все файлы .m4, которые можно найти, просматривая определения макросов (6.3.2. Путь поиска макросов), затем сканирует файл configure.ac. Любое из упоминаний одного из этих макросов на первом этапе приводит к включению этого макроса (и нужных ему макросов) в файл aclocal.m4.

Включение содержащего определение макроса файла в aclocal.m4 обычно выполняется путём копирования всего текста файла, включая неиспользуемые определения, а также комментарии # и dnl. Если нужно полностью игнорировать комментарии программой aclocal, следует использовать ##.

Когда выбранный aclocal файл размещается в каталоге, указанном относительным путём поиска в аргументе опции aclocal -I, программа aclocal предполагает, что файл относится к пакету и использует m4_include вместо копирования файла в aclocal.m4. Это снижает размер пакета, упрощает контроль зависимостей и обеспечивает автоматическое распространение файла (6.3.4. Обработка локальных макросов). Любой макрос, найденный в общесистемном каталоге или по абсолютному пути, будет копироваться. Поэтому следует использовать -I pwd/reldir вместо -I reldir, когда тот или иной относительный каталог следует считать расположенным вне пакета.

Содержимое acinclude.m4 (если файл существует) также автоматически включается в aclocal.m4. Не рекомендуется применять acinclude.m4 в новых пакетах (6.3.4. Обработка локальных макросов).

При создании aclocal.m4 программа aclocal запускает autom4te для отслеживания реально используемых макросов и пропуска в aclocal.m4 всех макросов, которые упоминаются, но не преобразуются (это может происходить при условном вызове макроса). Предполагается, что файл autom4te доступен через переменную PATH как и autoconf. Местоположение файла можно переопределить в переменной окружения AUTOM4TE.

6.3.1. Опции aclocal

Ниже перечислены опции, принимаемые командой aclocal.

--automake-acdir=DIR
: Задаёт поиск макросов, предоставленных automake, в каталоге DIR вместо установочного каталога. Опция служит в основном для отладки. Аналогичного результата можно путём остановки переменной среды ACLOCAL_AUTOMAKE_DIR, но опция --automake-acdir имеет более высокий приоритет.

--system-acdir=DIR
: Задаёт поиск сторонних общесистемных файлов с макросами (и специального файла dirlist) в каталоге DIR вместо установочного каталога. Опция служит в основном для отладки.

--diff[=COMMAND]
: Выполнить команду COMMAND для файла M4, который будет установлен или перезаписан с помощью --install. По умолчанию в качестве COMMAND применяется diff -u. Опция предполагает наличие —install и --dry-run.

--dry-run
: Не перезаписывать и не создавать на деле aclocal.m4 и файлы M4 при наличии опции --install (имитация работы).

--help
: Выводит справочную информацию о программе.

-I DIR
: Добавляет DIR в список каталогов для поиска файлов .m4.

--install
: Задаёт установку сторонних общесистемных макросов в первый каталог, заданный -I DIR, вместо их копирования в выходной файл (DIR может быть и абсолютным путём). При использовании этой опции (и только в этом случае) aclocal будет учитывать строки #serial NUMBER в макросах. Файл M4 игнорируется, если в пути поиска есть другой файл M4 с таким же базовым именем и большим порядковым номером (6.3.5. Порядковые номера).

--force
: Всегда переписывать выходной файл, что по умолчанию выполняется лишь при необходимости (т. е. при изменении его содержимого или обновлении зависимостей). Эта опция принудительно обновляет файл aclocal.m4 (или файл, заданный опцией --output) и только его, не затрагивая файлов, которые может понадобиться установить с помощью --install.

--output=FILE
: Задаёт вывод в файл FILE вместо aclocal.m4.

--print-ac-dir
: Задаёт вывод каталоге, где aclocal ищет сторонние файлы .m4, подавляя обычную обработку. Эта опция применялась в прошлом сторонними приложениями для определения мест установки файлов .m4, но сейчас её использование не рекомендуется, поскольку она ведёт к неполному соблюдению $(prefix) а нарушение GNU Coding Standards. Похожая семантика может быть достигнута с помощью переменной среды ACLOCAL_PATH (6.3.3. Создание макросов aclocal).

--verbose
: Задаёт вывод имён. проверяемых файлов.

--version
: Выводит номер версии Automake и завершает работу.

-W CATEGORY
--warnings=CATEGORY
: Задаёт вывод предупреждений указанной категории CATEGORY:

  syntax
  сомнительные синтаксические конструкции, избыточные скобки в макросах, неиспользуемые макросы и т. п.;
  unsupported
  неизвестный макрос;
  all
  все предупреждения (принято по умолчанию);
  none
  отключить все предупреждения;

error
считать все предупреждения ошибками.

Переменная среды WARNINGS учитывается так же, как в случае automake (5. Создание Makefile.in).

6.3.2. Путь поиска макросов

По умолчанию aclocal ищет файлы .m4 в указанных ниже каталогах (в порядке их перечисления).

ACDIR-APIVERSION
Здесь хранятся макросы .m4 из пакета Automake. Значение APIVERSION зависит от выпуска Automake, например, для Automake 1.11.x значением APIVERSION будет 1.11.

ACDIR
Этот каталог предназначен для сторонних файлов .m4 и задаётся при сборке automake. Это @datadir@/aclocal/, что обычно преобразуется в ${prefix}/share/aclocal'. Для определения заданного в программе значения ACDIR служит опция --print-ac-dir (6.3.1. Опции aclocal).

В качестве примера предположим, что программа automake-1.11.2 была настроена с --prefix=/usr/local. Тогда путями поиска будут

```
1. /usr/local/share/aclocal-1.11.2/
2. /usr/local/share/aclocal/
```

Пути к каталогам ACDIR и ACDIR-APIVERSION могут быть изменены с помощью опций --system-acdir и --automake-acdir (6.3.1. Опции aclocal). Однако следует отметить, что эти опции предназначены для использования лишь в наборе тестов Automake или для отладки в необычных ситуациях, а не для применения конечными пользователями.

Как указано в параграфе 6.3.1. Опции aclocal, имеется несколько опций для смены или расширения путей поиска.

-I DIR

Любые дополнительные каталоги, указанные с опцией -I (6.3.1. Опции aclocal), добавляются в начало списка. Таким образом, использование опций aclocal -I /foo -I /bar создаст список поиска

```
1. /foo
2. /bar
3. ACDIR-APIVERSION
4. ACDIR
```

dirlist

При наличии файла dirlist в пути ACDIR предполагается, что этот файл содержит список шаблонов каталогов, которые добавляются в конец поискового списка. В записях dirlist обычно применяются шаблоны *, ?, [...]. Предположим, что файл ACDIR/dirlist имеет вид

```
/test1
/test2
/test3*
```

и команда aclocal была вызвана с опциями -I /foo -I /bar. Путь поиска в этом случае будет иметь вид

```
1. /foo
2. /bar
3. ACDIR-APIVERSION
4. ACDIR
5. /test1
6. /test2
7. все каталоги, начинающиеся с /test3
```

При использовании опции --system-acdir=DIR программа aclocal будет искать dirlist в каталоге DIR, но следует помнить о приведённом выше предупреждении по части использования --system-acdir.

Файл dirlist полезен в следующей ситуации. Предположим, что программа automake версии 1.11.2 установлена с опцией --prefix=/usr поставщиком операционной системы. Тогда по умолчанию будет применяться путь поиска

```
1. /usr/share/aclocal-1.11/
2. /usr/share/aclocal/
```

Однако при установке в системе множества пакетов вручную часто используется $prefix=/usr/local. В такой ситуации многие из дополнительных файлов .m4 будут размещено в каталоге /usr/local/share/aclocal. Единственным способом заставить программу /usr/bin/aclocal находить эти файлы .m4 будет использование команды aclocal -I /usr/local/share/aclocal, что неудобно. Файл /usr/share/aclocal/dirlist с приведённой ниже строкой решает эту проблему.

```
/usr/local/share/aclocal
```

В этом случае по умолчанию будет использоваться путь поиска

```
1. /usr/share/aclocal-1.11/
2. /usr/share/aclocal/
3. /usr/local/share/aclocal/
```

без применения опций -I, которые можно применить для специфических потребностей проекта (my-source-dir/m4/). Файл dirlist может быть полезен при установке Automake в своей учётной записи (не общесистемной), если нужно видеть макросы, установленные в других местах системы.

ACLOCAL_PATH

Этот механизм является простейшим способом настройки путей поиска макро-файлов. Любой каталог, указанный в разделённом двоеточиями списке переменной среды ACLOCAL_PATH, добавляется в путь поиска и имеет преимущество перед системными каталогами (включая указанные в dirlist), за исключением ACDIR-APIVERSION и каталогов, добавленных с помощью опции -I.

Отметим, что при использовании опции --install любой файл .m4, содержащий нужный макрос и найденный в каталоге из ACLOCAL_PATH, будет установлен локально с учётом порядковых номеров в .m4 (6.3.5. Порядковые номера). В отличие от dirlist, переменная ACLOCAL_PATH полезна при использовании глобальной копии Automake и желании видеть макросы в своём домашнем каталоге.

Предполагаемые несовместимости в будущем

Порядок каталогов в пути поиска весьма запутан и может создавать путаницу и неоптимальные решения, поэтому вероятно его изменение в будущих выпусках Automake. В частности, каталоги из ACLOCAL_PATH и ACDIR могут оказаться предпочтительней каталогов ACDIR-APIVERSION, а каталоги из ACDIR/dirlist могут оказаться предпочтительней ACDIR. Это может создавать проблемы совместимости.

6.3.3. Создание макросов aclocal

Программа aclocal сама по себе не знает о каких-либо макросах и может быть легко расширена для использования любых макросов. Это может применяться библиотеками для добавления макросов Autoconf, доступных другим программам. Например, библиотека gettext включает макрос AM_GNU_GETTEXT. Который следует применять всем пакетам, использующим gettext. При установке библиотеки устанавливается этот макрос и aclocal будет видеть его.

Для макро-файлов следует использовать расширение .m4, а файлы следует помещать в каталог $(datadir)/aclocal.

```
aclocaldir = $(datadir)/aclocal
aclocal_DATA = mymacro.m4 myothermacro.m4
```

Следует использовать каталог $(datadir)/aclocal, а не что-то иное на основе aclocal --print-ac-dir (27.10. Установка в жёстко заданные места). Может оказаться полезным предложить пользователю добавить каталог $(datadir)/aclocal в переменную ACLOCAL_PATH, чтобы программа aclocal находила файлы .m4, автоматически установленные пакетом.

Файлу макросов следует быть последовательностью макросов AC_DEFUN с корректным использованием скобок. Программа aclocal также понимает макросы AC_REQUIRE, что позволяет помещать каждый макрос в отдельный файл. В каждом файле следует задавать определения макросов, не включая дополнительных действий. В частности, любые вызовы AC_PREREQ следует делать внутри макроса, а не в начале файла.

Начиная с Automake версии 1.8, программа aclocal выдаёт предупреждения при некорректных вызовах AC_DEFUN. Это может вызывать раздражение, поскольку раньше такой строгости не было и во многих сторонних макросах возможны такие вызовы. Причина более строгого контроля заключается в том, что будущие выпуски aclocal (6.3.6. Будущее aclocal) должны будут временно включать эти сторонние файлы .m4 (возможно несколько раз) даже при отсутствии реальной потребности в них. Пересмотр имеющихся макросов не требует больших усилий. Например,

```
# дурной стиль
AC_PREREQ(2.68)
AC_DEFUN(AX_FOOBAR,
[AC_REQUIRE([AX_SOMETHING])dnl
AX_FOO
AX_BAR
])
```

следует переписать в

```
AC_DEFUN([AX_FOOBAR],
[AC_PREREQ([2.68])dnl
AC_REQUIRE([AX_SOMETHING])dnl
AX_FOO
AX_BAR
])
```

Размещение вызова AC_PREREQ внутри макроса гарантирует, что Autoconf 2.68 не потребуется, если AX_FOOBAR на деле не применяется. Важно, что указание первого аргумента AC_DEFUN в скобках позволяет переопределить или дважды включить этот макрос (в ином случае этот первый аргумент будет преобразовываться во втором определении). Для согласованности в скобки помещаются даже аргументы, которым это не требуется, например 2.68.

Если предупреждение относится к стороннему макросу, следует обратиться к сопровождающему пакет. Использование свежей версии макросов поможет избавиться от избыточных предупреждений.

Другим случаем широкого использования aclocal является управление локальными макросами пакета, описанное ниже.

6.3.4. Обработка локальных макросов

Тесты, включённые в Autoconf не охватывают все потребности, поэтому для расширения часто применяют свои или сторонние макросы. Есть два способа организации в пакете дополнительных макросов.

Первый (исторический) способ заключается в указании всех макросов в файле acinclude.m4. Этот файл включается в aclocal.m4 при запуске aclocal и макросы становятся видны программе autoconf. Однако при большом числе включаемых макросов поддержка файла осложняется и становится почти невозможным совместное использование.

Второй вариант (рекомендуемый) заключается в записи каждого макроса в отдельный файл и размещение всех таких файлов в одном каталоге (обычно его называют m4/). В этом случае достаточно добавить в файл configure.ac корректный вызов AC_CONFIG_MACRO_DIRS

```
AC_CONFIG_MACRO_DIRS([m4])
```

Программа aclocal будет автоматически добавлять каталог m4/ в путь поиска файлов m4.

При работе aclocal программа будет собирать файл aclocal.m4, где m4_include используется для каждого файла из каталога m4/, определяющего нужный макрос. Не найденные локально макросы будут отыскиваться в общесистемных каталогах (6.3.2. Путь поиска макросов).

Пользовательские макросы следует распространять на тех же основаниях, что и configure.ac, чтобы другие люди имели все исходные файлы вашего пакета, если хотят работать с ним. В реальности такое распространение происходит автоматически, поскольку распространяются все файлы из m4_include.

Однако нет согласия в части распространения сторонних макросов, которые может использовать пакет. Многие библиотеки устанавливают свои макросы в общесистемный каталог aclocal (6.3.3. Создание макросов aclocal). Например, пакет Guile включает файл guile.m4 с макросом GUILE_FLAGS. Который может применяться для установки флагов компилятора и компоновщика при использовании Guile. Применение GUILE_FLAGS в configure.ac будет заставлять aclocal копировать файл guile.m4 в aclocal.m4, но guile.m4 не является частью проекта, поэтому не будет распространяться. Технически это означает, что пользователь, которому нужно перестроить aclocal.m4, должен будет установить сначала Guile. Это нормально, если Guile требуется для сборки пакета, но если пакет может работать в

архитектуре где установка Guile невозможна, это создаст препятствие. Простым решением будет копирование таких сторонних макросов в локальный каталог m4/ для их распространения. Поскольку в Automake 1.10 программа aclocal поддерживает опцию --install для копирования общесистемных сторонних макросов в локальный каталог, это тоже помогает решить проблему. В такой ситуации общесистемные макросы будут копироваться в каталог m4/ при первом запуске aclocal. После этого установленные локально макросы будут иметь предпочтение перед общесистемными при каждом запуске aclocal.

Одной из причин держать --install во флагах даже при первом запуске является то, что при последующем редактировании configure.ac и зависимости от нового макроса этот макрос будет установлен в m4/ автоматически. Другая причина заключается в том, что для обновления макросов могут применяться порядковые номера, когда в системе устанавливаются новые версии общих макросов. Порядковые номера указываются строкой вида

```
#serial NNN
```

где NNN содержит лишь цифры и точки. Эту строку следует включать в файл M4 до определений макросов. Хорошим тоном является поддержка порядкового номера для каждого распространяемого макроса, даже если не используется опция --install — это позволит другим использовать ваши макросы.

6.3.5. Порядковые номера

Поскольку сторонние макросы из файлов *.m4 естественно обобщаются множеством проектов, некоторым людям нравится указывать их версии. Таким способом проще сказать, какой из файлов M4 является более новым. С 1996 г. стало нормой использование для этого порядковых номеров в форме

```
# serial VERSION
```

где значение VERSION указывает номер версии (только цифры и точки). Обычно в качестве номера используют целые числа и увеличивают номер при каждом изменении макроса. Строка с номером должна предшествовать определениям. Первым символом строки должен быть диез (#), а после номера версии можно указать дополнения

```
#serial VERSION GARBAGE
```

Обычно эти номера игнорируются программами aclocal и autoconf, как и комментарии. Однако при использовании aclocal --install эти порядковые номера будут менять способ выбора программой aclocal порядковых номеров для установки в пакет. При наличии нескольких файлов с одинаковыми номерами и наличии хотя бы в одном из них строки #serial, программ aclocal будет игнорировать файлы с более старым номером (или без него).

Отметим, что порядковый номер относится ко всему файлу M4, а не только к содержащимся в нем макросам. Файл может включать много макросов, но порядковый номер должен быть единственным.

Ниже представлен пример использования опции --install и взаимодействия с порядковыми номерами. Предположим, что поддерживается пакет MyPackage, для которого в configure.ac указан сторонний макрос AX_THIRD_PARTY, определённый в файле /usr/share/aclocal/thirdparty.m4, как показано ниже

```
# serial 1
AC_DEFUN([AX_THIRD_PARTY], [...])
```

MyPackage использует каталог m4/ для хранения локальных макросов, как описано в параграфе 6.3.4. Обработка локальных макросов, и включает строку

```
AC_CONFIG_MACRO_DIRS([m4])
```

в файле configure.ac.

Изначально каталог m4/ пуст. При первом запуске будет отмечено, что

- configure.ac использует AX_THIRD_PARTY;
- локальные макросы не определяют AX_THIRD_PARTY;
- файл /usr/share/aclocal/thirdparty.m4’ определяет AX_THIRD_PARTY с порядковым номером 1.

Поскольку макрос /usr/share/aclocal/thirdparty.m4 является общесистемным и программа aclocal была запущена с опцией --install, она будет копировать этот файл в m4/thirdparty.m4 и вывод aclocal.m4 будет включать m4_include([m4/thirdparty.m4]).

При следующем запуске aclocal --install обнаруживаются некоторые изменение и aclocal скажет, что

- configure.ac использует AX_THIRD_PARTY;
- m4/thirdparty.m4 определяет AX_THIRD_PARTY с порядковым номером 1;
- /usr/share/aclocal/thirdparty.m4 определяет AX_THIRD_PARTY с порядковым номером 1.

Поскольку в обоих файлах указан один порядковый номер, aclocal использует найденный первым файл (6.3.2. Путь поиска макросов). В результате aclocal игнорирует файл /usr/share/aclocal/thirdparty.m4 и выводит aclocal.m4 с m4_include([m4/thirdparty.m4]).

Локальные каталоги, указанные в опции -I, всегда просматриваются до общесистемных каталогов, поэтому локальные файлы всегда оказываются предпочтительней общесистемных при совпадении порядковых номеров.

Предположим, что внешний общесистемный макрос был изменён (например, при обновлении установившего макрос пакета). Пусть новый макрос имеет порядковый номер 2. При следующем вызове aclocal --install будет

- configure.ac использует AX_THIRD_PARTY;
- m4/thirdparty.m4 определяет AX_THIRD_PARTY с порядковым номером 1;
- /usr/share/aclocal/thirdparty.m4 определяет AX_THIRD_PARTY с порядковым номером 2.

Когда aclocal видит больший порядковый номер, программа немедленно забывает все, что известно из файлов с таким же именем и меньшим порядковым номером. Поэтому при обнаружении usr/share/aclocal/thirdparty.m4 с номером 2 aclocal будет работать как будто ей ничего не известно о m4/thirdparty.m4. Это возвращает к ситуации, похожей на

начало примера, где нет локального файла с определением макроса. Программа aclocal будет устанавливать новую версию макроса m4/thirdparty.m4, переопределяя прежнюю. Пакет MyPackage в результате обновляет свой макрос как побочный эффект запуска aclocal.

Если не нужно, чтобы программа aclocal обновляла локальные макросы, можно использовать aclocal —diff для просмотра изменений, которые команда aclocal --install будет вносить в эти макросы.

Опция --force для команды aclocal не оказывает влияния на файлы, устанавливаемые опцией --install. Например, при изменении локальных макросов не предполагается замена их на общесистемные по команде с опциями --install --force. Если такая замена нужна, следует просто удалить локальные макросы и ввести команду aclocal --install.

6.3.6. Будущее aclocal

Предполагается, что использование программы aclocal будет прекращено, поскольку эту функциональность не следует предоставлять в Automake. Пакету Automake следует сосредоточиться на создании файлов Makefile, а работа с M4 относится к Autoconf. Тот факт, что некоторые люди устанавливают Automake лишь для использования aclocal, но не пользуются самой программой automake указывает, что эта функция не востребована. Новые реализации могут отличаться, например, может обеспечиваться схема в стиле m4/, рассмотренная в параграфе 6.3.4. Обработка локальных макросов. Пока не понятно, как это произойдёт. Вопрос этот неоднократно обсуждался, но задача ещё не решена.

С точки зрения пользователя удаление aclocal может оказаться болезненным. Есть простая мера предосторожности, позволяющая смягчить этот переход - никогда не использовать программу aclocal саму по себе. Следует использовать эту программу лишь под контролем autoreconf и правил перестройки Automake. Если же используется непосредственный вызов aclocal, придётся столкнуться с проблемами при исключении программы.

Многие пакеты распространяются со сценариями bootstrap или autogen.sh, которые просто вызывают aclocal, libtoolize, gettextize или autopoint, autoconf, autoheader и automake в нужном порядке. На деле точно то, что может сделать autoreconf. Если ваш пакет включает сценарий bootstrap или autogen.sh, следует рассмотреть использование autoreconf. Это должно упростить логику и в результате сценарии станут ненужными как и вызовы aclocal напрямую.

В настоящее время сторонние пакеты продолжают устанавливать публичные макросы в каталог /usr/share/aclocal/. При замене aclocal другим инструментом каталог может быть переименован, но поддержка /usr/share/aclocal/ для совместимости со старыми версиями должна быть простой, если все макросы написаны верно (6.3.3. Создание макросов aclocal).

6.4. Макросы Autoconf из пакета Automake

Automake распространяется с несколькими макросами Autoconf, которые можно вызывать из файла configure.ac. Программа aclocal будет включать используемые макросы в файл aclocal.m4.

6.4.1. Макросы общего пользования

AM_INIT_AUTOMAKE([OPTIONS])

Запускает множество макросов, требуемых для корректной работы создаваемых файлов Makefile.

AM_INIT_AUTOMAKE вызывается с одним аргументом в виде списка разделённых пробелами опций Automake, которые следует применить к каждому файлу Makefile.am в дереве кода. Влияние каждой опции описано ниже (17. Смена поведения Automake).

Этот макрос может также вызываться в устаревшей форме AM_INIT_AUTOMAKE(PACKAGE, VERSION, [NO-DEFINE]) с двумя аргументами - пакет и номер версии, которые извлекаются с помощью макроса Autoconf AC_INIT. Однако отличие состоит в том, что для вызова AC_INIT макрос AM_INIT_AUTOMAKE поддерживает преобразование переменных среды в аргументах PACKAGE и VERSION (в ином случае по умолчанию используются PACKAGE_TARNAME и PACKAGE_VERSION, определённые через вызов AC_INIT). В некоторых случаях такой вариант может оказаться полезным. Есть надежда, что в будущих версиях Autoconf будет улучшена поддержка автоматического определения версий. Если это произойдёт., поддержка вызова AM_INIT_AUTOMAKE с двумя аргументами будет исключена из Automake.

Если в вашем файле configure.ac имеются строки вида

```
AC_INIT([src/foo.c])
AM_INIT_AUTOMAKE([mumble], [1.5])
```

их следует заменить, как показано ниже

```
AC_INIT([mumble], [1.5])
AC_CONFIG_SRCDIR([src/foo.c])
AM_INIT_AUTOMAKE
```

Отметим, что при обновлении configure.ac с прежних версий Automake не всегда корректен показанный выше перенос аргументов package и version из AM_INIT_AUTOMAKE в макрос AC_INIT. Первым аргументом AC_INIT должно быть имя пакета (например, GNU Automake), а не имя архива (например, automake), используемое в AM_INIT_AUTOMAKE. Autoconf пытается вывести имя архива из имени пакета и это работает в большинстве случаев. Если же это не проходит, можно использовать вызов AC_INIT с 4 аргументами для явного указания архива.

По умолчанию используются аргументы AC_DEFINE PACKAGE и VERSION, но этого можно избежать путём передачи опции no-define (17.2. Список опций Automake)

```
AM_INIT_AUTOMAKE([no-define ...])
```

AM_PATH_LISPDIR

Ищет программу emacs и (при её обнаружении) устанавливает в переменной lispdir полный путь к каталогу Emacs site-lisp.

Этот тест предполагает, что найденная программа emacs поддерживает Emacs Lisp (например, GNU Emacs или XEmacs). При обнаружении другого варианта emacs этот тест «зависает» (например, старая версия MicroEmacs входит в интерактивный режим, для выхода из которого нужно нажать клавиши C-x C-c[1], что неочевидно для непривычных к emacs пользователей). Однако в большинстве случаев тест можно прервать клавишами C-c. Для предотвращения проблем можно установить в переменной окружения EMACS значение no или использовать

[1] *Control+x, Control+c - одновременное нажатие клавиш.*

опцию --with-lispdir при вызове сценария configure с явным указанием корректного пути (если имеется emacs с поддержкой Emacs Lisp).

AM_PROG_AR([ACT-IF-FAIL])
Этот макрос требуется вызывать при использовании в проекте необычного архиватора (такого как Microsoft lib). Необязательный аргумент указывает программу, выполняемую, если интерфейс архиватора не распознан. По умолчанию прерывает выполнение сценария configure с возвратом сообщения об ошибке.

AM_PROG_AS
Этот макрос применяется при наличии в проекте ассемблерного кода, указывая применяемый ассемблер (по умолчанию компилятор C) и устанавливая CCAS, а при необходимости и CCASFLAGS.

AM_PROG_CC_C_O
Этот устаревший макрос проверяет поддержку компилятором C опций -c и -o одновременно. Начиная с Automake 1.14 макрос AC_PROG_CC самостоятельно выполняет такую проверку, поэтому использовать AM_PROG_CC_C_O больше не требуется.

AM_PROG_LEX
Похож на AC_PROG_LEX, но использует сценарий missing в системах без lex (например, HP-UX 10).

AM_PROG_GCJ
Находит программу gcj или возвращает ошибку. Макрос устанавливает переменные GCJ и GCJFLAGS. Программа gcj обеспечивает Java-интерфейс для компилятора GCC.

AM_PROG_UPC([COMPILER-SEARCH-LIST])
Находит компилятор для Unified Parallel C и устанавливает переменную UPC. По умолчанию COMPILER-SEARCH-LIST имеет значение upcc upc. Макрос прерывает сценарий configure, если компилятор Unified Parallel C не найден.

AM_MISSING_PROG(NAME, PROGRAM)
Находит инструмент поддержки PROGRAM и задаёт в переменной среды NAME его местоположение. Если найти PROGRAM не удалось, NAME будет задавать вызов сценария missing для получения информации об отсутствии инструмента (27.2. Сценарий missing и режим AM_MAINTAINER_MODE).

AM_SILENT_RULES
Задаёт менее подробный вывод в процессе сборки (21.3. Как Automake может «заглушить» make).

AM_WITH_DMALLOC
Добавляет поддержку пакета Dmalloc. Если пользователь задаёт configure --with-dmalloc, макрос определяет WITH_DMALLOC и добавляет -ldmalloc в LIBS.

6.4.2. Устаревший макрос

Описанный ниже макрос мог требоваться в прошлых выпусках, но его не следует применять в новом коде. Макрос будет удалён. из Automake и для его исключения следует воспользоваться командой autoupdate configure.ac.

AM_PROG_MKDIR_P
В Automake версий 1.8 - 1.9.6 этот макрос устанавливал для выходной переменной mkdir_p значение mkdir -p, install-sh -d или mkinstalldirs.
Сейчас Autoconf обеспечивает похожую функциональность с помощью AC_PROG_MKDIR_P, однако при этом определяется переменная MKDIR_P. Если макрос AM_PROG_MKDIR_P по прежнему используется в вашем configure.ac или переменная $(mkdir_p) присутствует в Makefile.am, рекомендуется как можно быстрей отказаться от этого, поскольку макрос и переменная будут удалены в новых выпусках Automake.

6.4.3. Приватные макросы

Перечисленные ниже макросы не следует вызывать напрямую, поскольку они предназначены для вызова из других макросов и могут быть изменены в будущем. А лучше совсем не применять их!

_AM_DEPENDENCIES
AM_SET_DEPDIR
AM_DEP_TRACKAM_OUTPUT_DEPENDENCY_COMMANDS
Служат для реализации в Automake схемы автоматического отслеживания зависимостей. Они могут автоматически вызываться Automake при необходимости.

AM_MAKE_INCLUDE
Этот макрос служит для определения способа обработки операторов include в пользовательском make. Макрос автоматически вызывается при необходимости.

AM_PROG_INSTALL_STRIP
Служит для определения версии install, которая может применяться для «вырезания» программы при установке. Макрос автоматически вызывается при необходимости.

AM_SANITY_CHECK
Проверка того, что файл в каталоге сборки новее файла в каталоге источников. Может приводить к отказу в системах с некорректной установкой часов. Макрос вызывается автоматически из AM_INIT_AUTOMAKE.

7. Каталоги

Для простых проектов, распространяющих все файлы в одном каталоге, достаточно иметь один файл Makefile.am. Более крупные проекты организуют файлы в разные каталоги дерева. Например, могут быть каталоги для исходных кодов программы, набора тестов, документации. А в очень крупных проектах могут создаваться каталоги для отдельных программ, библиотек, модулей и т. п.
традиционно каталоги организуются рекурсивно и каждый каталог содержит свой Makefile. При запуске make из каталога верхнего уровня по очереди выполняется вход в нужные каталоги и вызывается новый экземпляр make для сборки содержимого каталога.

Поскольку такой подход получил широкое распространение, Automake имеет для него встроенную поддержку. Однако рекурсивная модель имеет свои плюсы и минусы. Вполне возможно создавать пакеты с структурой каталогов, где рекурсия не применяется или используется редко, например, GNU Bison и GNU Automake ().

7.1. Рекурсия каталогов

В пакетах, использующих make recursion, файл Makefile.am на верхнем уровне должен указать Automake каталоги для сборки. Это делается с помощью переменной SUBDIRS, которая включает список каталогов, где должна выполняться та или иная сборка.

Правила для множества целей (например, all) в созданном Makefile будут выполнять команду в текущем каталоге и указанных подкаталогах. Отметим, что в каталогах SUBDIRS не требуется наличия файлов Makefile.am, достаточно Makefile (после настройки конфигурации). Это позволяет включать библиотеки из пакетов, не использующих Automake, таких как gettext (23.2. Сторонние файлы Makefile).

В пакетах с подкаталогами файл Makefile.am верхнего уровня часто бывает очень коротким. Например, Makefile.am из GNU Hello имеет вид

```
EXTRA_DIST = BUGS ChangeLog.O README-alpha
SUBDIRS = doc intl po src tests
```

Когда Automake вызывает make в подкаталоге, используется та же переменная MAKE. Программа передаёт значение переменной AM_MAKEFLAGS при вызове make, устанавливаемую в файле Makefile., если нужно передать флаги команде make.

Каталоги, указанные в SUBDIRS, обычно являются прямыми потомками текущего каталога и каждый подкаталог содержит свой файл Makefile.am с переменной SUBDIRS, указывающей каталоги следующего уровня. Automake можно использовать для создания пакетов с произвольной глубиной вложенности.

По умолчанию Automake создаёт файлы Makefile, в которых обработка подкаталогов выполняется раньше обработки текущего каталога. Однако этот порядок можно поменять, помещая текущий каталог (.) в переменную SUBDIRS. Например, SUBDIRS = lib src . test приведёт к обработке каталогов в порядке lib/, src/, текущий каталог и test/. Обычно каталоги с тестами обрабатываются в последнюю очередь, поскольку они предназначены для проверки собранного.

Помимо встроенной рекурсии целей Automake (all, check и т. п.), разработчик может определить свои рекурсивные цели. Это делается путём передачи имён. таких целей в качестве аргументов макросу m4 AM_EXTRA_RECURSIVE_TARGETS в файле configure.ac. Automake создаёт правила для обработки рекурсии в таких целях и разработчик может задать реальные действия для них, определяя цели -local, как показано ниже.

```
% cat configure.ac
AC_INIT([pkg-name], [1.0]
AM_INIT_AUTOMAKE
AM_EXTRA_RECURSIVE_TARGETS([foo])
AC_CONFIG_FILES([Makefile sub/Makefile sub/src/Makefile])
AC_OUTPUT
% cat Makefile.am
SUBDIRS = sub
foo-local:
        @echo This will be run by "make foo".
% cat sub/Makefile.am
SUBDIRS = src
% cat sub/src/Makefile.am
foo-local:
        @echo This too will be run by a "make foo" issued either in
        @echo the 'sub/src/' directory, the 'sub/' directory, or the
        @echo top-level directory.
```

7.2. Условные подкаталоги

Можно задать переменную SUBDIRS с условиями, если нужно собрать лишь часть пакета. Предположим, что в пакете имеются подкаталоги src/ и opt/, причём src/ следует собирать всегда, а программе configure можно указать, следует ли собирать opt/ (в примере предполагается, что для сборки opt/ нужно установить в переменной $want_opt значение yes). В результате команда make всегда должна выполняться в каталоге src/ и может выполняться также в opt/. Однако команду make dist всегда следует выполнять в обоих каталогах src/ и opt/, поскольку распространять opt/ следует в любом случае. Это означает, что создание opt/Makefile следует выполнять безусловно.

Есть два способа решения этой задачи. Можно воспользоваться условными конструкциями Automake (20. Конструкции с условием) или использовать переменные AC_SUBST. Первый способ является предпочтительным. Перед описание обоих вариантов рассмотрим переменную DIST_SUBDIRS.

7.2.1. Переменные SUBDIRS и DIST_SUBDIRS

Automake рассматривает 2 набора каталогов, заданные в переменных SUBDIRS и DIST_SUBDIRS. SUBDIRS указывает подкаталоги текущего каталога, которые нужно собрать (7.1. Рекурсия каталогов), и должна задаваться вручную. Ниже будет показано, как исключить тот или иной каталог из сборки по заданным условиям. DIST_SUBDIRS применяется в правилах, которые нужны для рекурсии во все каталоги, даже если они не используются при сборке. В приведённом выше примере каталог opt/ не используется при сборке, но включается в дистрибутив. В результате opt может не присутствовать в SUBDIRS, но должен быть указан в DIST_SUBDIRS.

Переменная DIST_SUBDIRS используется целями maintainer-clean, distclean и dist, а в остальных правилах применяется SUBDIRS. Если переменная SUBDIRS задана с использованием условной конструкции Automake, программа Automake будет автоматически определять DIST_SUBDIRS на основе возможных значений SUBDIRS и всех условий. Если SUBDIRS включает переменные AC_SUBST, DIST_SUBDIRS не удастся задать корректно, поскольку Automake не будет знать возможных значений переменных. В таких случаях нужно задавать DIST_SUBDIRS вручную.

7.2.2. Подкаталоги с AM_CONDITIONAL

Сценарию configure следует создавать Makefile для каждого каталога и определять условие сборки каталога opt/.

```
...
AM_CONDITIONAL([COND_OPT], [test "$want_opt" = yes])
AC_CONFIG_FILES([Makefile src/Makefile opt/Makefile])
...
```

Переменную SUBDIRS можно определить в Makefile.am верхнего уровня, как показано ниже.

```
if COND_OPT
  MAYBE_OPT = opt
endif
SUBDIRS = src $(MAYBE_OPT)
```

При запуске make будет выполняться рекурсия в каталог src/ и возможно в opt/. По команде make dist будет выполняться рекурсия в src/ и opt/, поскольку make dist в отличие от make all не использует переменную SUBDIRS.

В этом случае Automake будет задавать DIST_SUBDIRS = src opt автоматически, поскольку известно, что MAYBE_OPT может при выполнении условий включать каталог opt.

7.2.3. Подкаталоги с AC_SUBST

Другим вариантом является определение MAYBE_OPT из ./configure с использованием AC_SUBST.

```
...
if test "$want_opt" = yes; then
  MAYBE_OPT=opt
else
  MAYBE_OPT=
fi
AC_SUBST([MAYBE_OPT])
AC_CONFIG_FILES([Makefile src/Makefile opt/Makefile])
...
```

В этом случае файл Makefile.am на верхнем уровне будет иметь вид

```
SUBDIRS = src $(MAYBE_OPT)
DIST_SUBDIRS = src opt
```

Недостаток этого варианта заключается в том, что Automake не знает всех возможных значений MAYBE_OPT, которые нужны для указания DIST_SUBDIRS.

7.2.4. Не заданные в конфигурации каталоги

Семантику DIST_SUBDIRS часто толкуют некорректно и пытаются задать условия для каталогов настройки и сборки. Здесь настройкой считается создание Makefile (это может включать запуск вложенного сценария configure).

В приведённых выше примерах предполагается создание каждого файла Makefile даже в каталогах, где сборка не выполняется. Причина этого заключается в том, что команда make dist должна распространять все каталоги (например, зависимый от платформы код), поэтому команда make dist должна выполняться рекурсивно во всех каталогах и они должны быть настроены и указаны в переменной DIST_SUBDIRS.

Сборка пакетов с использованием лишь части каталогов является сложным делом и не рекомендуется новичкам, поскольку легко ведёт к созданию неполных архивов. Этот вопрос не рассматривается подробно, а для любителей приключений ниже приведено несколько правил.

- Переменная SUBDIRS всегда должна быть подмножеством DIST_SUBDIRS.
- Любой каталог, указанный в DIST_SUBDIRS и SUBDIRS, должен быть настроен, т. е. включать файл Makefile, поскольку иначе рекурсивные правила make не смогут обработать каталог.
- Все настроенные в конфигурации каталоги должны быть указаны в переменной DIST_SUBDIRS. Это нужно для того, чтобы правила очистки удаляли созданные Makefile. Разумно считать DIST_SUBDIRS переменной, в которой указаны все настроенные каталоги.

Для предотвращения рекурсии в ненастроенные каталоги нужно убедиться, что таких каталогов нет в DIST_SUBDIRS (и SUBDIRS). Например, при условном определении SUBDIRS с использованием AC_SUBST без явного задания DIST_SUBDIRS по умолчанию эта переменная будет совпадать с $(SUBDIRS). Другим вариантом является явное задание DIST_SUBDIRS = $(SUBDIRS).

Каталоги, не включённые в DIST_SUBDIRS, не будут распространяться, если для этого не приняты иные меры (например, выполнение make dist в конфигурации, где все каталоги заведомо включены в DIST_SUBDIRS, или создание цели dist-hook для распространения пропущенных каталогов).

В некоторых пакетах ненастроенные каталоги в действительности не нужно распространять. Хотя такие пакеты не требуют выполнения приведённых выше рекомендаций, есть ещё одна ловушка. Если имя каталога указано в SUBDIRS или DIST_SUBDIRS, программа automake будет уверена в наличии каталога. Поэтому automake нельзя запустить в дистрибутиве с пропущенным каталогом. Одним из способов обхода такой проверки является использование метода AC_SUBST для объявления условных каталогов. Поскольку automake не знает значений переменных AC_SUBST, гарантировать их наличие программа не сможет.

7.3. Другая модель организации каталогов

Если вы читали отличную статью Peter Miller «Recursive Make Considered Harmful», предыдущие рассуждения о рекурсивном использовании make могут показаться ненужными. Для тех, кто не читал статью, отметим, что её основной тезис заключается в том, что рекурсивные вызовы make медленны и ведут к ошибкам.

Automake обеспечивает достаточную поддержку сборки в нескольких каталогах[1] с одним файлом Makefile.am для сложной структуры каталогов. По умолчанию для устанавливаемого файла из подкаталога имя каталога вырезается перед установкой. Например, приведённая ниже строка приведёт к установке файла заголовков в $(includedir)/stdio.h.

```
include_HEADERS = inc/stdio.h
```

Однако можно использовать префикс nobase_ для предотвращения этого вырезания. Приведённая ниже строка обеспечит установку заголовочного файла в $(includedir)/sys/types.h.

```
nobase_include_HEADERS = sys/types.h
```

Префикс nobase_ следует указывать до dist_ или nodist_ при их совместном использовании (14.2. Тонкая настройка распространения). Например,

[1] *Эта работа является новой и может содержать ошибки.*

```
nobase_dist_pkgdata_DATA = images/vortex.pgm sounds/whirl.ogg
```

В заключение отметим, что переменную с префиксом часто можно заменить несколькими переменными, по одной для каждого целевого каталога (3.3. Схема именования). Например, приведённую строку можно заменить комбинацией

```
imagesdir = $(pkgdatadir)/images
soundsdir = $(pkgdatadir)/sounds
dist_images_DATA = images/vortex.pgm
dist_sounds_DATA = sounds/whirl.ogg
```

Этот вариант позволяет изменить один установочный каталог без изменения схемы дерева кодов. В настоящее время единственным исключением из этого правила является nobase_*_LTLIBRARIES, где нет гарантии определённого порядка установки для набора переменных без префикса nobase_.

7.4. Вложенные пакеты

В системе сборки GNU возможна произвольная глубина вложенности пакетов. Это означает, что пакет может содержать в себе другие пакеты со своим сценарием configure, файлами Makefile и т. п. Эти вложенные пакеты следует просто размещать в подкаталогах родительского пакета. Каталоги можно указать в SUBDIRS подобно другим каталогам. Однако для создания файлов Makefile в субпакетах должны использоваться свои сценарии configure, а не родительский. Это достигается с помощью макроса Autoconf AC_CONFIG_SUBDIRS.

Ниже приведён пример пакета для программы arm, которая связана с библиотекой hand в форме вложенного пакета в каталоге hand/. Файл configure.ac для arm показан ниже.

```
AC_INIT([arm], [1.0])
AC_CONFIG_AUX_DIR([.])
AM_INIT_AUTOMAKE
AC_PROG_CC
AC_CONFIG_FILES([Makefile])
# Call hand's ./configure script recursively.
AC_CONFIG_SUBDIRS([hand])
AC_OUTPUT
```

Файл Makefile.am для arm имеет вид

```
# Build the library in the hand subdirectory first.
SUBDIRS = hand

# Include hand's header when compiling this directory.
AM_CPPFLAGS = -I$(srcdir)/hand

bin_PROGRAMS = arm
arm_SOURCES = arm.c
# link with the hand library.
arm_LDADD = hand/libhand.a
```

Ниже приведён файл hand/configure.ac для библиотеки hand.

```
AC_INIT([hand], [1.2])
AC_CONFIG_AUX_DIR([.])
AM_INIT_AUTOMAKE
AC_PROG_CC
AM_PROG_AR
AC_PROG_RANLIB
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Файл hand/Makefile.am имеет вид

```
lib_LIBRARIES = libhand.a
libhand_a_SOURCES = hand.c
```

При запуске make dist из каталога верхнего уровня будет создаваться архив arm-1.0.tar.gz с кодом arm и подкаталогом hand. Этот пакет можно собрать и установить как обычные пакеты командами ./configure && make && make install (пакет hand будет собран и установлен в этом процессе).

При запуске make dist из каталога hand будет создаваться архив hand-1.2.tar.gz, содержащий лишь код вложенного пакета, который можно использовать независимо.

Целью инструкции AC_CONFIG_AUX_DIR([.]) является обеспечение поиска программами Automake и Autoconf дополнительных сценариев в текущем каталоге. Например, здесь будет два файла install-sh - в каталоге верхнего уровня пакета arm и в подкаталоге hand/ пакета hand.

Исторически по умолчанию поиск этих файлов выполняется в родительском каталоге и его родителе. Поэтому при удалении строки AC_CONFIG_AUX_DIR([.]) из файла hand/configure.ac этот субпакет будет использовать дополнительный сценарий пакета arm. Это может показаться увеличением размера (несколько килобайт), но на деле означает утрату модульности, поскольку пакет hand перестаёт быть самодостаточным (make dist в подкаталоге не будет работать).

Пакетам, не использующим Automake, требуются дополнительные действия для такой интеграции (23.2. Сторонние файлы Makefile).

8. Сборка программ и библиотек

Значительная часть функциональности Automake связана с упрощением сборки программ и библиотек.

8.1. Сборка программы

Для сборки программы нужно сказать Automake какие исходные коды относятся к программе и с какими библиотеками её следует компоновать.

В этом разделе также рассматривается условная компиляция программ. Большая часть приведённых здесь описаний относится и к сборке библиотек (8.2. Сборка библиотек) и библиотекам libtool (8.3. Сборка общих библиотек).

8.1.1. Указание источников программы

В каталоге, содержащем исходный код, из которого собирается программа (в отличие от библиотеки или сценария), используется первичная переменная PROGRAMS. Программы могут устанавливаться в bindir, sbindir, libexecdir, pkglibexecdir или не устанавливаться совсем (noinst_). Они могут также собираться лишь для цели make (префикс (check_). Например,

```
bin_PROGRAMS = hello
```

В этом простом случае результирующий файл Makefile.in будет содержать код для сборки программы hello.

С каждой программой связано несколько вспомогательных переменных, указываемых после программы. Эти переменные не обязательны и имеют разумные значения, принятые по умолчанию. Каждая переменная, её применение и значение по умолчанию описаны ниже. Во всех случаях используется пример hello.

Переменная hello_SOURCES служит для указания исходных файлов, служащих для сборки программы

```
hello_SOURCES = hello.c version.c getopt.c getopt1.c getopt.h system.h
```

Это ведёт к компиляции каждого заданного файла .c в файл .o, затем они компонуются в hello. Если переменная hello_SOURCES не задана, используется один файл hello.c (8.5. Принятые по умолчанию значения _SOURCES).

В одном каталоге может собираться множество программ. Возможно также собрать много программ из одного файла с исходным кодом, указанного в каждом определении _SOURCES.

Файлы заголовков в определении _SOURCES будут включаться в дистрибутив, а больше нигде не нужны. Если это не очевидно, в переменную не следует включать заголовочный файл, созданный сценарием configure, поскольку его не нужно распространять. Файлы Lex (.l) и Yacc (.y) тоже могут включаться в _SOURCES (8.8. Поддержка Yacc и Lex).

8.1.2. Компоновка программ

Если нужна компоновка с библиотекой, которая не найдена сценарием configure, можно применить для этого LDADD. Эта переменная служит для указания дополнительных объектов и библиотек с целью компоновки. Для установки флагов компоновщика переменная не подходит и для этого нужно использовать AM_LDFLAGS.

Иногда множество программ собирается в одном каталоге, но программы различаются по требованиям при компоновке. В этом случае можно использовать переменную PROG_LDADD (где PROG - имя программы, как оно указано в переменной _PROGRAMS, обычно записанное строчными буквами) для переопределения LDADD. Если эта переменная существует для данной программы, программа не будет компоноваться с использованием LDADD.

Например, в пакете GNU cpio программы pax, cpio и mt компонуются с библиотекой libcpio.a. однако rmt собирается в том же каталоге и не имеет требований к компоновке. Кроме того, mt и rmt собираются лишь в некоторых архитектурах. Ниже приведён файл src/Makefile.am для решения этой задачи (сокращённый).

```
bin_PROGRAMS = cpio pax $(MT)
libexec_PROGRAMS = $(RMT)
EXTRA_PROGRAMS = mt rmt

LDADD = ../lib/libcpio.a $(INTLLIBS)
rmt_LDADD =

cpio_SOURCES = ...
pax_SOURCES = ...
mt_SOURCES = ...
rmt_SOURCES = ...
```

PROG_LDADD не подходит для передачи специфичных для программы флагов компоновщика (за исключением -l, -L, -dlopen и -dlpreopen), поэтому для передачи флагов применяется переменная PROG_LDFLAGS.

Иногда полезно, чтобы программа зависела от некой иной цели, которая не является частью программы. Это можно организовать с помощью переменной PROG_DEPENDENCIES или EXTRA_PROG_DEPENDENCIES. Обе переменные содержат список зависимостей, но интерпретации различаются. Поскольку эти зависимости связаны с правилом компоновки, используемым при создании программы, в переменных следует обычно перечислять файлы, используемые командой компоновки (*.$(OBJEXT), *.a, *.la). В редких случаях может потребоваться добавление иных типов файлов, таких как сценарии компоновщиков, но указание исходных файлов в _DEPENDENCIES является ошибкой. Если некоторые исходные файлы нужно собрать до того, как будут собраны все компоненты программы, следует применять переменную BUILT_SOURCES.

Если переменная PROG_DEPENDENCIES не задана, её определяет Automake, присваивая значение переменной PROG_LDADD, из которого удалено большинство подстановок -l, -L, -dlopen и -dlpreopen. Оставшиеся конфигурационные подстановки включают лишь $(LIBOBJS) и $(ALLOCA), поскольку они заведомо не приводят к созданию недопустимых значений для PROG_DEPENDENCIES. Использование переменных _DEPENDENCIES проиллюстрировано в параграфе 8.1.3. Условная компиляция исходного кода.

Переменная EXTRA_PROG_DEPENDENCIES может быть полезна в случаях, когда нужно просто расширить созданную automake переменную PROG_DEPENDENCIES, а не заменить её.

Рекомендуется избегать опций -l в LDADD и PROG_LDADD при ссылках на библиотеки, собираемые пакетом. Вместо этого следует явно указать имя файла библиотеки, как показано выше в примере cpio. Опцию -l следует использовать лишь для перечисления сторонних библиотек. Если следовать этому правилу, подразумеваемое значение PROG_DEPENDENCIES будет включать список локальных библиотек, не включая остальных.

8.1.3. Условная компиляция исходного кода

Можно поместить подстановку (например, @FOO@ или $(FOO), где FOO определена через AC_SUBST) в переменную _SOURCES. Это сложно объяснить, но такая подстановка не работает и Automake будет давать ошибку. К счастью есть два других способа получить нужный результат.

Подстановки _LDADD

Automake нужно знать все исходные файлы, которые могут применяться при сборке программы, даже если некоторые из них используются не всегда. Файлы, используемые лишь при определённых условиях, следует указывать в подходящих переменных EXTRA_. Например, если hello-linux.c или hello-generic.c условно включаются в hello, Makefile.am может иметь вид

```
bin_PROGRAMS = hello
hello_SOURCES = hello-common.c
EXTRA_hello_SOURCES = hello-linux.c hello-generic.c
hello_LDADD = $(HELLO_SYSTEM)
hello_DEPENDENCIES = $(HELLO_SYSTEM)
```

Затем можно организовать подстановку $(HELLO_SYSTEM) в configure.ac

```
...
case $host in
  *linux*) HELLO_SYSTEM='hello-linux.$(OBJEXT)' ;;
  *)       HELLO_SYSTEM='hello-generic.$(OBJEXT)' ;;
esac
AC_SUBST([HELLO_SYSTEM])
...
```

В этом случае переменная HELLO_SYSTEM будет заменена hello-linux.o или hello-generic.o и добавлена в обе переменные hello_DEPENDENCIES и hello_LDADD для использования при сборке и компоновке.

Условные конструкции Automake

Например, можно использовать приведённую ниже конструкцию Automake в файле Makefile.am при сборке hello.

```
bin_PROGRAMS = hello
if LINUX
hello_SOURCES = hello-linux.c hello-common.c
else
hello_SOURCES = hello-generic.c hello-common.c
endif
```

В этом случае в файле configure.ac следует задать условие LINUX с использованием AM_CONDITIONAL (20. Конструкции с условием). При использовании таких условных конструкций не требуется использовать переменную EXTRA_, поскольку Automake будет проверять содержимое каждой переменной для создания полного списка исходных файлов. Если программа использует много файлов, может оказаться удобным оператор +=.

```
bin_PROGRAMS = hello
hello_SOURCES = hello-common.c
if LINUX
hello_SOURCES += hello-linux.c
else
hello_SOURCES += hello-generic.c
endif
```

8.1.4. Условная компиляция программ

Иногда полезно указать собираемые программы во время настройки конфигурации. Например, в GNU cpio компоненты mt и rmt собираются только в особых случаях. Здесь также применяются подстановки и конструкции с условиями.

Подстановки configure

В этом случае нужно уведомить Automake обо всех программах, которые могут собираться, но в то же время включать в создаваемый файл Makefile.in программы, заданные сценарием configure. Это делается путём подстановки сценарием configure значение для каждого определения _PROGRAMS с указанием необязательных программ в EXTRA_PROGRAMS.

```
bin_PROGRAMS = cpio pax $(MT)
libexec_PROGRAMS = $(RMT)
EXTRA_PROGRAMS = mt rmt
```

Как отмечено в параграфе 8.20. Поддержка расширений исполняемых файлов, Automake будет переопределять bin_PROGRAMS, libexec_PROGRAMS и EXTRA_PROGRAMS, добавляя суффикс $(EXEEXT) к каждому двоичному файлу. Очевидно, что невозможно переопределить значения полученные путём подстановок при выполнении configure, поэтому нужно самостоятельно добавить суффиксы $(EXEEXT), как в AC_SUBST([MT], ['mt${EXEEXT}']).

Условные конструкции Automake

Можно также использовать условные конструкции Automake (20. Конструкции с условием) для выбора собираемых программ. В этом случае не нужно заботиться о переменных $(EXEEXT) и EXTRA_PROGRAMS.

```
bin_PROGRAMS = cpio pax
if WANT_MT
  bin_PROGRAMS += mt
endif
if WANT_RMT
  libexec_PROGRAMS = rmt
endif
```

8.2. Сборка библиотек

Сборка библиотек похожа на сборку программ. В этом случае применяется первичная переменная LIBRARIES. Библиотеки могут устанавливаться в libdir или pkglibdir. В параграфе 8.3. Сборка общих библиотек рассмотрено создание библиотек общего пользования (shared) с помощью libtool и LTLIBRARIES.

Каждая переменная _LIBRARIES содержит список собираемых библиотек. Например, для создания библиотеки libcpio.a без её установки можно задать

```
noinst_LIBRARIES = libcpio.a
libcpio_a_SOURCES = ...
```

Включаемые в библиотеку источники задаются так же, как для программ, через переменные _SOURCES. Отметим, что имена библиотек преобразуются (3.5. Именование производных переменных), поэтому переменная _SOURCES для libcpio.a будет иметь вид libcpio_a_SOURCES, а не libcpio.a_SOURCES.

В библиотеку можно добавить объекты с помощью переменной LIBRARY_LIBADD, которую следует применять для объектов, заданных сценарием configure. Например, для cpio можно указать

```
libcpio_a_LIBADD = $(LIBOBJS) $(ALLOCA)
```

источники для дополнительных объектов, которых не было во время работы configure, должны добавляться в переменную BUILT_SOURCES (9.4. Исходные файлы для сборки).

Сборка статической библиотеки выполняется путём компиляции всех объектных файлов и последующего вызова $(AR) $(ARFLAGS) с именем библиотеки и списком объектов. Завершается процесс вызовом $(RANLIB) для этой библиотеки. Следует вызвать макрос AC_PROG_RANLIB в файле configure.ac для указания RANLIB (иначе Automake будет «жаловаться»). Следует также вызвать AM_PROG_AR для задания AR, чтобы обеспечить поддержку необычных архивов, таких как Microsoft lib. По умолчанию ARFLAGS имеет значение cru, которое можно переопределить путём установки в файле Makefile.am или вызова AC_SUBST из файла configure.ac. Можно переопределить переменную AR путём задания на уровне библиотеки переменной maude_AR (8.4. Переменные для программ и библиотек).

Следует соблюдать осторожность при выборе условных компонент библиотеки. Сборка пустой библиотеки не переносима, поэтому следует убедиться в наличии в библиотеке хотя бы одного объекта.

Для использования статической библиотеки при сборке программы в эту программу добавляется переменная LDADD. В примере показана статическая компоновка cpio с библиотекой libcpio.a.

```
noinst_LIBRARIES = libcpio.a
libcpio_a_SOURCES = ...

bin_PROGRAMS = cpio
cpio_SOURCES = cpio.c ...
cpio_LDADD = libcpio.a
```

8.3. Сборка общих библиотек

Создание совместно используемых (shared) библиотек является более сложной задачей. Поэтому был создан инструмент GNU Libtool, помогающий собирать такие библиотеки независимо от платформы.

8.3.1. Концепции Libtool

Libtool абстрагирует разделяемые и статические библиотеки в единую концепцию библиотек libtool. Библиотеками libtool являются файлы с расширением .la, которые могут быть статическими разделяемыми или теми и другими сразу. Точную природу библиотеки невозможно определить до запуска сценария ./configure. Не все платформы поддерживают каждый тип библиотек и пользователь может явно задать тип собираемой библиотеки, однако сопровождающие пакет разработчики могут настроить принятую по умолчанию сборку.

Поскольку объектные файлы для разделяемых и статических библиотек должны компилироваться по-разному, libtool применяется и при компиляции. Объектные файлы, собираемые libtool, называют объектами libtool и для них используется расширение .lo. Из этих объектов собираются библиотеки libtool.

Не следует принимать каких-либо допущений о структуре файлов .la и .lo, а также способах их создания программой libtool - это забота libtool. Однако наличие этих файлов имеет значение, поскольку они служат целями и зависимостями в правилах Makefile при сборке библиотек libtool. В некоторых случаях могут потребоваться ссылки на такие файлы, например, при условном задании зависимостей для сборки исходных файлов по условию (8.3.4. Библиотеки Libtool с условными источниками).

При разработке подключаемых модулей (plug-in) с динамической загрузкой следует обратить внимание на libltdl - библиотеку расширения libtool. Она обеспечивает переносимую функцию dlopen для динамической загрузки библиотек libtool и поддерживает статическую компоновку, когда она необходима.

Описание использования libtool вместе с Automake достаточно подробно рассмотрено в руководстве libtool.

8.3.2. Сборка библиотек Libtool

Automake использует libtool для сборки библиотек, объявленных в LTLIBRARIES. Каждая переменная _LTLIBRARIES содержит список собираемых библиотек libtool. Например, для создания библиотеки libgettext.la и её установки в libdir

```
lib_LTLIBRARIES = libgettext.la
libgettext_la_SOURCES = gettext.c gettext.h ...
```

Automake заранее определяет переменную pkglibdir, что позволяет использовать pkglib_LTLIBRARIES для установки библиотек в $(libdir)/@PACKAGE@/.

Если gettext.h является общедоступным заголовочным файлом, который нужно установить для использования библиотеки, его следует объявить в переменной _HEADERS, а не в libgettext_la_SOURCES. В последней переменной следует указывать файлы заголовков, которые являются внутренними и не относятся к общедоступному интерфейсу.

```
lib_LTLIBRARIES = libgettext.la
libgettext_la_SOURCES = gettext.c ...
include_HEADERS = gettext.h ...
```

Пакет может собрать и установить такую библиотеку вместе с использующими её программами. Эту зависимость следует указывать в LDADD. Ниже приведён пример сборки программы hello, компонуемой с libgettext.la.

```
lib_LTLIBRARIES = libgettext.la
libgettext_la_SOURCES = gettext.c ...

bin_PROGRAMS = hello
hello_SOURCES = hello.c ...
hello_LDADD = libgettext.la
```

Выбор статической или динамической компоновки hello с libgettext.la пока не определён и будет зависеть от конфигурации libtool и возможностей хоста.

8.3.3. Условная сборка библиотек Libtool

Подобно условной сборке программ (8.1.4. Условная компиляция программ), есть два варианта сборки библиотек по условию - подстановки Autoconf AC_SUBST и условные конструкции Automake.

Для реализации важно то, что место установки библиотеки имеет значение для libtool и должно быть указано к моменту компоновки с помощью опции -rpath. Для библиотек, место установки которых известно во время работы Automake, программа Automake будет автоматически передавать libtool подходящую опцию -rpath. Это выполняется для библиотек, явно указанных в переменных _LTLIBRARIES, таких как lib_LTLIBRARIES.

Однако для библиотек, определяемых во время настройки конфигурации (упоминаемых в EXTRA_LTLIBRARIES), Automake не знает окончательный каталог установки. В таких случаях нужно вручную добавлять опцию -rpath в подходящую переменную _LDFLAGS.

В приведённых ниже примерах показаны различия между этими методами. В первом примере для WANTEDLIBS используется подстановка AC_SUBST, выполняемая сценарием ./configure, где устанавливается libfoo.la, libbar.la, обе или ни одной библиотеки. Хотя $(WANTEDLIBS) присутствует в lib_LTLIBRARIES, Automake не может знать о libfoo.la или libbar.la при создании правила компоновки библиотек. Поэтому аргумент -rpath должен быть задан явно.

```
EXTRA_LTLIBRARIES = libfoo.la libbar.la
lib_LTLIBRARIES = $(WANTEDLIBS)
libfoo_la_SOURCES = foo.c ...
libfoo_la_LDFLAGS = -rpath '$(libdir)'
libbar_la_SOURCES = bar.c ...
libbar_la_LDFLAGS = -rpath '$(libdir)'
```

ниже показан файл Makefile.am при использовании условных конструкций Automake (WANT_LIBFOO и WANT_LIBBAR). Здесь Automake может определить -rpath, поскольку обе библиотеки явно попадут в $(libdir) при установке.

```
lib_LTLIBRARIES =
if WANT_LIBFOO
lib_LTLIBRARIES += libfoo.la
endif
if WANT_LIBBAR
lib_LTLIBRARIES += libbar.la
endif
libfoo_la_SOURCES = foo.c ...
libbar_la_SOURCES = bar.c …
```

8.3.4. Библиотеки Libtool с условными источниками

Условную компиляцию исходных кодов библиотек можно реализовать так же, как это делается для программ (8.1.3. Условная компиляция исходного кода). Единственным отличием является использование переменной _LIBADD вместо _LDADD и указание объектов libtool (файлы .lo).

Возвращаясь к примеру hello из параграфа 8.1.3. Условная компиляция исходного кода, можно собрать библиотеку libhello.la, используя hello-linux.c или hello-generic.c, указанных в файле Makefile.am.

```
lib_LTLIBRARIES = libhello.la
libhello_la_SOURCES = hello-common.c
EXTRA_libhello_la_SOURCES = hello-linux.c hello-generic.c
libhello_la_LIBADD = $(HELLO_SYSTEM)
libhello_la_DEPENDENCIES = $(HELLO_SYSTEM)
```

Нужно убедиться, что configure задаёт в HELLO_SYSTEM значение hello-linux.lo или hello-generic.lo.

Можно также просто воспользоваться условной конструкцией Automake, показанной ниже.

```
lib_LTLIBRARIES = libhello.la
libhello_la_SOURCES = hello-common.c
if LINUX
libhello_la_SOURCES += hello-linux.c
else
libhello_la_SOURCES += hello-generic.c
endif
```

8.3.5. Вспомогательные библиотеки Libtool

Иногда нужно собрать библиотеки libtool, которые не следует устанавливать. Их называют вспомогательными (libtool convenience libraries) и эти библиотеки обычно применяются для инкапсуляции множества суббиблиотек, которые позднее собираются в одну большую устанавливаемую библиотеку.

Вспомогательные библиотеки Libtool объявляются в переменных без каталогов, таких как noinst_LTLIBRARIES, check_LTLIBRARIES или даже EXTRA_LTLIBRARIES. В отличие от устанавливаемых библиотек libtool, им не нужна опция -rpath в момент компоновки (это единственное отличие).

Вспомогательные библиотеки из noinst_LTLIBRARIES собираются всегда, а библиотеки из check_LTLIBRARIES - лишь по команде make check. Библиотеки из переменной EXTRA_LTLIBRARIES никогда не создаются явно, Automake выводит правила для их сборки, но при отсутствии библиотеки в зависимостях Makefile она не будет собрана (именно поэтому EXTRA_LTLIBRARIES применяется для условной компиляции).

Ниже приведён пример, объединяющий вспомогательные библиотеки из подкаталогов в одну библиотеку libtop.la.

```
# -- Top-level Makefile.am --
SUBDIRS = sub1 sub2 ...
lib_LTLIBRARIES = libtop.la
libtop_la_SOURCES =
libtop_la_LIBADD = \
  sub1/libsub1.la \
  sub2/libsub2.la \
  ...

# -- sub1/Makefile.am --
noinst_LTLIBRARIES = libsub1.la
libsub1_la_SOURCES = ...

# -- sub2/Makefile.am --
# showing nested convenience libraries
SUBDIRS = sub2.1 sub2.2 ...
noinst_LTLIBRARIES = libsub2.la
libsub2_la_SOURCES =
libsub2_la_LIBADD = \
  sub21/libsub21.la \
  sub22/libsub22.la \
  ...
```

При таком подходе следует помнить, что automake будет предполагать, что libtop.la будет связываться с помощью компоновщика C. Это обусловлено тем, что значение libtop_la_SOURCES пусто, поэтому automake подразумевает язык C. При непустом значении libtop_la_SOURCES программа automake выберет компоновщик в соответствии с параграфом 8.14.3.1. Выбор компоновщика.

Если какая-то из суббиблиотек содержит источники, отличные от C, важно выбрать правильный компоновщик. Одним из способов является «фиктивное» указание присутствия отличного от C исходного кода, заставляющее automake выбрать подходящий компоновщик. Ниже приведён файл Makefile верхнего уровня для выбора компоновки C++.

```
SUBDIRS = sub1 sub2 ...
lib_LTLIBRARIES = libtop.la
libtop_la_SOURCES =
# Dummy C++ source to cause C++ linking.
nodist_EXTRA_libtop_la_SOURCES = dummy.cxx
libtop_la_LIBADD = \
  sub1/libsub1.la \
  sub2/libsub2.la \
  ...
```

Переменные EXTRA_*_SOURCES служат для отслеживания исходных файлов, которые могут компилироваться (полезно при условной компиляции с использованием AC_SUBST, 8.1.3. Условная компиляция исходного кода), а префикс nodist_ указывает источники, которые не нужно распространять (8.4. Переменные для программ и библиотек). Файл dummy.cxx не обязан присутствовать в дереве исходного кода. Если же есть какой-то реальный файл, отличный от C, в libtop_la_SOURCES, нет смысла указывать фиктивный в nodist_EXTRA_libtop_la_SOURCES.

8.3.6. Модули Libtool

Библиотеки libtool для динамической загрузки (dlopen), указываемые libtool параметром -module при компоновке.

```
pkglib_LTLIBRARIES = mymodule.la
mymodule_la_SOURCES = doit.c
mymodule_la_LDFLAGS = -module
```

Обычно Automake требует, чтобы имена библиотек начинались с lib, однако при сборке динамически загружаемого модуля может возникнуть желание воспользоваться «нестандартным» именем. Automake не будет «жаловаться» на такие имена, если известно, что собираемая библиотека является модулем libtool, т. е. -module явно присутствует в переменной библиотеки _LDFLAGS (или в общей переменной AM_LDFLAGS, если для библиотеки не задаётся _LDFLAGS).

Как обычно, переменные AC_SUBST неведомы Automake, поскольку их значение не известны во время работы automake. Поэтому при установке -module через такие переменные Automake не заметит их и будет работать как с обычной библиотекой libtool, строго относясь к именам.

Если переменная mymodule_la_SOURCES не задана, подразумевается один файл mymodule.c (8.5. Принятые по умолчанию значения _SOURCES).

8.3.7. _LIBADD, _LDFLAGS, _LIBTOOLFLAGS

Как отмечено в предыдущем параграфе, следует применять переменную LIBRARY_LIBADD для указания списка дополнительных объектов (.lo) или библиотек libtool (.la), добавляемых в LIBRARY. В переменной LIBRARY_LDFLAGS указываются дополнительные флаги компоновки libtool, такие как -version-info, -static и т. п.

Команда libtool поддерживает два типа опций - базовые и зависящие от режима. Специфические для режима опции, такие как отмеченные выше флаги компоновки, следует объединять с другими флагами, передаваемыми инструментам при вызове из libtool (отсюда использование LIBRARY_LDFLAGS для флагов компоновки libtool). Базовые опции включают --tag=TAG и --silent и их следует указывать до выбора режиме в строке команды, а в Makefile.am их следует перечислять в переменной LIBRARY_LIBTOOLFLAGS. Если переменная LIBRARY_LIBTOOLFLAGS не задана, взамен применяется AM_LIBTOOLFLAGS.

Эти флаги передаются libtool после опции --tag=TAG, найденной Automake (если она есть), поэтому для переопределения или дополнения опции --tag=TAG следует использовать LIBRARY_LIBTOOLFLAGS (или AM_LIBTOOLFLAGS).

Правила libtool используют также переменную LIBTOOLFLAGS, которую не следует устанавливать Makefile.am, она является пользовательской (27.6. Порядок переменных флагов). Это позволяет пользователю применять, например, команды make LIBTOOLFLAGS=--silent. Отметим, что на уровень информативности вывода libtool может также влиять поддержка в Automake «тихих» правил (21.3. Как Automake может «заглушить» make).

8.3.8. LTLIBOBJS и LTALLOCA

Там, где обычные библиотеки могут включать $(LIBOBJS) или $(ALLOCA) (8.6. Особая обработка LIBOBJS и ALLOCA), библиотека libtool должна использовать $(LTLIBOBJS) или $(LTALLOCA). Это обусловлено тем, что объектные файлы, с которыми работает libtool не обязательно используют расширение .o.

В настоящее время определение LTLIBOBJS из LIBOBJS выполняется автоматически программой Autoconf.

8.3.9. Проблемы, связанные с использованием Libtool

8.3.9.1. Error: ‘required file `./ltmain.sh' not found’

Libtool распространяется с инструментом libtoolize, который устанавливает файлы поддержки libtool для пакета. При запуске этой команды создаётся и устанавливается сценарий ltmain.sh, который следует выполнить до запуска aclocal и automake.

При обновлении старых пакетов эта проблема может возникать в результате того, что, начиная с Automake 1.6, запуск libtoolize был исключён из действий Automake и функциональность перенесена в команду autoreconf. Замена прежних сценариев bootstrap и autogen.sh вызовом autoreconf должна устранить в будущем эту несовместимость.

8.3.9.2. Объекты ‘created with both libtool and without’

Иной раз один и тот же файл с исходным кодом используется для сборки библиотеки libtool и другой цели (программа или библиотека). Рассмотрим файл Makefile.am

```
bin_PROGRAMS = prog
prog_SOURCES = prog.c foo.c ...

lib_LTLIBRARIES = libfoo.la
libfoo_la_SOURCES = foo.c ...
```

В описанном тривиальном случае проблемы можно избежать, компонуя libfoo.la с prog вместо указания foo.c в prog_SOURCES. Но в реальности может потребоваться разделение prog и libfoo.la. Технически это означает, что нужно собрать foo.$(OBJEXT) для prog и foo.lo для libfoo.la. Проблема заключается в том, что в процессе создания foo.lo libtool может удалить (или заменить foo.$(OBJEXT), а этого следует избегать. Поэтому Automake при обнаружении такой ситуации будет выдавать сообщение вида

```
object 'foo.$(OBJEXT)' created both with libtool and without
```

Обход этой проблемы заключается в обеспечении разных базовых имён. объектов. Как отмечено в параграфе 27.7. Переименование объектных файлов, это выполняется автоматически при установке флагов на уровне каждой цели.

```
bin_PROGRAMS = prog
prog_SOURCES = prog.c foo.c ...
prog_CFLAGS = $(AM_CFLAGS)

lib_LTLIBRARIES = libfoo.la
libfoo_la_SOURCES = foo.c ...
```

Добавление prog_CFLAGS = $(AM_CFLAGS) почти ничего не делает (no-op), поскольку при наличии переменной prog_CFLAGS она используется вместо AM_CFLAGS. Однако побочным эффектом этого является компиляция prog.c и foo.c как prog-prog.$(OBJEXT) и prog-foo.$(OBJEXT) для решения упомянутой проблемы.

8.4. Переменные для программ и библиотек

С каждой программой связан набор переменных, которые можно использовать для управления сборкой программы. Подобный список имеется и для каждой библиотеки. В качестве базы для именования таких переменных служит каноническое имя программы или библиотеки, которое в приведённом ниже списке обозначено maude. В своих файлах Makefile.am вы должны будете указать реальные имена. В приведённом списке maude считается программой, но практически те же правила применимы для статических и динамических библиотек, а различия отмечены в тексте.

maude_SOURCES

При наличии этой переменной она содержит список исходных файлов, компилируемых для сборки программы. Эти файлы по умолчанию добавляются в дистрибутив программы. При сборке программы Automake будет обеспечивать компиляцию каждого источника в один файл .o (.lo при использовании libtool). Обычно эти объектные файлы именуются по источнику, но это может быть изменено. Если файл в переменной _SOURCES имеет нераспознанное расширение, Automake может выбрать для него один из двух вариантов. При наличии правила для суффиксов для преобразования файлов с нераспознанным расширением в файлы .o, automake будет использовать файл как другие источники (8.18. Поддержка других языков). В противном случае файл игнорируется, как будто это заголовочный файл.

Префиксы dist_ и nodist_ могут служить для контроля распространения файлов из _SOURCES. Префикс dist_ является избыточным, поскольку источники распространяются по умолчанию, но он может применяться для чёткости понимания. Можно иметь варианты dist_ и nodist_ для данной переменной _SOURCES, что позволяет распространять одни файлы, не распространяя другие. Например,

```
nodist_maude_SOURCES = nodist.c
dist_maude_SOURCES = dist-me.c
```

По умолчанию выходной (в Unix-системах .o) будет помещаться в текущий каталог сборки. Однако при действии в текущем каталоге опции subdir-objects файл .o будет помещаться в подкаталог, названный по исходному файлу. Например, при включённой опции subdir-objects файл sub/dir/file.c будет компилироваться в sub/dir/file.o. некоторые люди предпочитают такой режим работы. Опцию subdir-objects можно задать в файле AUTOMAKE_OPTIONS.

EXTRA_maude_SOURCES

Automake нужно знать список предназначенных для компиляции файлов статически. Во-первых, это единственный способ указать Automake языки, поддержка которых требуется в данной файле Makefile.in[1]. Это означает, например, что можно использовать подстановку вида @my_sources@ для переменной _SOURCES’. Если нужна компиляция источников по условию и configure применяется для подстановки имён. соответствующих объектов (например, _LDADD), нужно указать соответствующие исходные файлы в переменной EXTRA_. Переменная также поддерживает префиксы dist_ и nodist_. Например, nodist_EXTRA_maude_SOURCES будет перечислять дополнительные исходные файлы, которые нужно собрать, но не распространять.

maude_AR

Статическая библиотека по умолчанию создаётся вызовом $(AR) $(ARFLAGS), за которым следует имя библиотеки и помещаемые в неё объекты. Это можно переопределить установкой переменной _AR. Обычно это применяется для C++, поскольку некоторые компиляторы C++ требуют специального вызова для создания экземпляров всех шаблонов, которые следует включить в библиотеку. Например, для компилятора SGI C++ нужно установить переменную вида

```
libmaude_a_AR = $(CXX) -ar -o
```

maude_LIBADD

Дополнительные объекты можно добавлять в библиотеку с помощью переменной _LIBADD. Например, это следует использовать для объектов, заданных configure (8.2. Сборка библиотек). Для библиотек libtool переменная maude_LIBADD может указывать также другие объекты libtool.

maude_LDADD

Дополнительные объекты (*.$(OBJEXT)) и библиотеки (*.a, *.la) можно добавить в в программу, указав их в переменной _LDADD. Например, это можно использовать для объектов, заданных configure (8.1.2. Компоновка программ).

Переменные _LDADD и _LIBADD не подходят для передачи зависящих от программы флагов компоновщика (за исключением -l, -L, -dlopen и -dlpreopen). Для этого служит переменная _LDFLAGS. Например, если файл configure.ac использует AC_PATH_XTRA, можно скомпоновать программу с библиотекой X

[1] *Есть и другие причины для этого ограничения.*

```
maude_LDADD = $(X_PRE_LIBS) $(X_LIBS) $(X_EXTRA_LIBS)
```

Рекомендуется применять опции -l и -L лишь при ссылках на сторонние библиотеки и указывать явно имена файлов для всех библиотек, создаваемых пакетом. Это гарантирует корректную установку переменной maude_DEPENDENCIES по умолчанию.

maude_LDFLAGS

Эта переменная служит для передачи дополнительных флагов этапу компоновки программы или общей библиотеки, переопределяя переменную AM_LDFLAGS.

maude_LIBTOOLFLAGS

Эта переменная служит для передачи дополнительных опций программе libtool, переопределяя переменную AM_LIBTOOLFLAGS. Опции выводятся перед опцией libtool --mode=MODE, поэтому им не следует зависеть от режима (они относятся к флагам компилятора или компоновщика) (9.4. Исходные файлы для сборки).

maude_DEPENDENCIES

EXTRA_maude_DEPENDENCIES

Иногда нужна зависимость цели (программа или библиотека) он файла, который не является её частью. Это можно задать с помощью переменной _DEPENDENCIES, от которой зависит каждая цель без дальнейшей интерпретации. Поскольку эти зависимости связаны с правилом компоновки, используемым для создания цели, переменная обычно должна указывать файлы, используемые командой компоновки. Это файлы *.$(OBJEXT), *.a или *.la для программ, *.lo и *.la для библиотек Libtool и *.$(OBJEXT) для статических библиотек. В редких случаях может потребоваться добавление других файлов, таких как сценарии компоновки, но включение исходных файлов в _DEPENDENCIES является ошибкой. Если какой-то исходный файл нужно скомпилировать до остальных компонент собираемой программы, следует использовать переменную BUILT_SOURCES (9.4. Исходные файлы для сборки).

Если переменная _DEPENDENCIES не задана, Automake создаёт её. Автоматически заданное значение содержит переменную _LDADD или _LIBADD, откуда удалено большинство подстановок configure (-l, -L, -dlopen, -dlpreopen). Остаются лишь подстановки $(LIBOBJS) и $(ALLOCA), поскольку они заведомо не ведут к созданию непригодных значений _DEPENDENCIES.

_DEPENDENCIES чаще применяется для условной компиляции с переменной AC_SUBST, содержащей список объектов (8.1.3. Условная компиляция исходного кода). Переменная EXTRA_*_DEPENDENCIES может быть полезна, когда нужно просто дополнить созданную автоматически переменную _DEPENDENCIES без её замены.

maude_LINK

Можно переопределить компоновщик на уровне программы. По умолчанию компоновщик выбирается на основе языка, используемого программой. Например, для программы с исходным кодом C++ будет использоваться в качестве компоновщика компилятор C++. Переменная _LINK содержит имя программы, которой могут быть переданы в качестве аргументов имена всех файлов .o и библиотек для сборки. Отметим, что имя базовой программы не передаётся в _LINK и обычно используется $@, как показано ниже

```
maude_LINK = $(CCLD) -magic -o $@
```

Если переменная _LINK не задана, она может быть создана и использована Automake, благодаря установленным на уровне цели флагам, таким как _CFLAGS, _LDFLAGS и _LIBTOOLFLAGS, если они применяются.

maude_CCASFLAGS
maude_CFLAGS
maude_CPPFLAGS
maude_CXXFLAGS
maude_FFLAGS
maude_GCJFLAGS
maude_LFLAGS
maude_OBJCFLAGS
maude_OBJCXXFLAGS
maude_RFLAGS
maude_UPCFLAGS
maude_YFLAGS

Automake позволяет установить флаги компиляции на уровне программы или библиотеки. Один файл исходного кода можно включить в несколько программ и он может компилироваться для каждой со своими флагами. Это работает с любым языком, напрямую поддерживаемым Automake. Такими флагами являются _CCASFLAGS, _CFLAGS, _CPPFLAGS, _CXXFLAGS, _FFLAGS, _GCJFLAGS, _LFLAGS, _OBJCFLAGS, _OBJCXXFLAGS, _RFLAGS, _UPCFLAGS и _YFLAGS. При использовании таких флагов Automake будет выбирать разные имена для промежуточных объектных файлов. Обычно файл, например, sample.c будет компилироваться в sample.o, однако при установке для программы переменной _CFLAGS, объект может быть назван, например, maude-sample.o (27.7. Переименование объектных файлов).

При компиляции с флагами на уровне программы обычная форма переменной флагов AM_ не включаются в компиляцию автоматически (однако пользовательская форма включается). Например, если нужно при компиляции maude использовать значение AM_CFLAGS, следует задать

```
maude_CFLAGS = ... ваши флаги ... $(AM_CFLAGS)
```

В параграфе 27.6. Порядок переменных флагов рассмотрено взаимодействие пользовательских переменных, теневых переменных AM_ и переменных на уровне цели.

maude_SHORTNAME

На некоторых платформах размер имён. файлов сильно ограничен и для поддержки в таких системах флагов компиляции на уровне цели Automake позволяет задать короткое имя, которое будет влиять на именование промежуточных объектных файлов. Например, в

```
bin_PROGRAMS = maude
maude_CPPFLAGS = -DSOMEFLAG
maude_SHORTNAME = m
maude_SOURCES = sample.c ...
```

объектный файл будет назван m-sample.o вместо maude-sample.o. Это свойство редко применяется на практике и рекомендуется не применять его без необходимости.

8.5. Принятые по умолчанию значения _SOURCES

Переменные _SOURCES служат для задания исходных файлов программ (8.1. Сборка программы), библиотек (8.2. Сборка библиотек) и библиотек Libtool (8.3. Сборка общих библиотек). Если такая переменная не задана для цели,

Automake будет определять её. По умолчанию компилируется один файл C, чьё базовое имя является именем цели с заменой расширения значением переменной AM_DEFAULT_SOURCE_EXT (по умолчанию .c). Например, при наличии в файле Makefile.am приведённой ниже строки без соответствующей переменной libfoo_a_SOURCES

```
lib_LIBRARIES = libfoo.a sub/libc++.a
```

будет собрана библиотека libfoo.a с использованием подразумеваемого имени файла libfoo.c и библиотека sub/libc++.a из sub/libc++.c. В старой версии библиотека sub/libc++.a была бы собрана из файла sub_libc___a.c, т. е. по умолчанию имя исходного файла канонизировалось по имени цели, а затем добавлялось расширение .c. Новое поведение представляется более осмысленным, но для совместимости с прежними версиями automake будет использовать старое имя, если есть файл или правило с таким именем, а переменная AM_DEFAULT_SOURCE_EXT не задана.)

Принятые по умолчанию источники полезны в основном для тестовых наборов при сборке множества тестов из одного источника, например, переменные

```
check_PROGRAMS = test1 test2 test3
AM_DEFAULT_SOURCE_EXT = .cpp
```

задают сборку программ test1, test2, test3 из файлов test1.cpp, test2.cpp и test3.cpp. Без последней строки для сборки применялись бы файлы test1.c, test2.c, test3.c.

Другим вариантом является сборка множества модулей Libtool (moduleN.la), каждый из которых определён в своём файле (moduleN.c).

```
AM_LDFLAGS = -module
lib_LTLIBRARIES = module1.la module2.la module3.la
```

Есть ситуации, в которых следует избегать компиляции подразумеваемых источников, - когда цель не следует собирать из источников. Выше был рассмотрен пример (4.2 Сборка двух программ из одного файла), где все цели уже скомпилированы и нужно просто собрать их с помощью переменной _LDADD. В этом случае нужно указать пустую переменную _SOURCES, чтобы программа automake не задавала компиляцию принятых по умолчанию источников.

```
bin_PROGRAMS = target
target_SOURCES =
target_LDADD = libmain.a libmisc.a
```

8.6. Особая обработка LIBOBJS и ALLOCA

В переменных $(LIBOBJS) и $(ALLOCA) перечисляются объектные файлы, которые следует скомпилировать в проект для реализации функций, которые отсутствуют или не работают в системе. Переменные задаёт сценарий configure.

Эти переменные определяются макросами Autoconf, такими как AC_LIBOBJ, AC_REPLACE_FUNCS или AC_FUNC_ALLOCA. Многие макросы Autoconf вызывают AC_LIBOBJ или AC_REPLACE_FUNCS для заполнения списка $(LIBOBJS).

Работа с этими переменными очень похожа на условную компиляцию с использованием переменных AC_SUBST (8.1.3. Условная компиляция исходного кода). Т. е. при сборке программы переменные $(LIBOBJS) и $(ALLOCA) следует добавить в связанную переменную *_LDADD или в переменную *_LIBADD при сборке библиотеки. Однако нет необходимости перечислять соответствующие источники в EXTRA_*_SOURCES или задавать *_DEPENDENCIES. Automake автоматически добавляет $(LIBOBJS) и $(ALLOCA) в зависимости и автоматически находит соответствующие исходные файлы (путём отслеживания вызовов макроса Autoconf AC_LIBSOURCE). Если переменные *_DEPENDENCIES уже определены явно по какой-либо причине, нужно добавить их вручную или использовать EXTRA_*_DEPENDENCIES вместо *_DEPENDENCIES.

Эти переменные обычно применяются для сборки переносимых библиотек, которые компонуются со всеми программами проекта. Ниже приведён простой пример, где файл configure.ac включает некоторые проверки, влияющие на LIBOBJS или ALLOCA.

```
# configure.ac
...
AC_CONFIG_LIBOBJ_DIR([lib])
...
AC_FUNC_MALLOC              dnl May add malloc.$(OBJEXT) to LIBOBJS
AC_FUNC_MEMCMP              dnl May add memcmp.$(OBJEXT) to LIBOBJS
AC_REPLACE_FUNCS([strdup]) dnl May add strdup.$(OBJEXT) to LIBOBJS
AC_FUNC_ALLOCA              dnl May add alloca.$(OBJEXT) to ALLOCA
...
AC_CONFIG_FILES([
  lib/Makefile
  src/Makefile
])
AC_OUTPUT
```

AC_CONFIG_LIBOBJ_DIR сообщает Autoconf, что исходные файлы объектов найдены в каталоге lib/. Automake может использовать эту информацию, а в ином случае ожидает, что эти файлы размещены в каталоге, где применяются переменные $(LIBOBJS) и $(ALLOCA). В каталоге lib/ должны присутствовать файлы malloc.c, memcmp.c, strdup.c, alloca.c. Ниже приведён файл Makefile.am из этого каталога

```
# lib/Makefile.am

noinst_LIBRARIES = libcompat.a
libcompat_a_SOURCES =
libcompat_a_LIBADD = $(LIBOBJS) $(ALLOCA)
```

Библиотека может иметь любое имя и она в любом случае не будет устанавливаться. Она лишь содержит замещающие версии отсутствующих или повреждённых функций, чтобы их можно было включить в компоновку. Многие проекты также включают свои дополнительные функции с такую библиотеку, они просто добавляются в _SOURCES.

Здесь есть небольшая ловушка - переменные $(LIBOBJS) и $(ALLOCA) могут быть пустыми, а пустые библиотеки не переносимы. Следует убедиться, что в libcompat.a всегда что-то включено. Большинство проектов добавляет некоторые утилиты в этот каталог и указывают их в libcompat_a_SOURCES, поэтому libcompat.a не будет пустой.

Ниже показано, как можно использовать библиотеку из каталога src/.

```
# src/Makefile.am

# Link all programs in this directory with libcompat.a
```

```
LDADD = ../lib/libcompat.a

bin_PROGRAMS = tool1 tool2 ...
tool1_SOURCES = ...
tool2_SOURCES = ...
```

Когда опция subdir-objects не используется, как в приведённом выше примере, переменную $(LIBOBJS) или $(ALLOCA) можно использовать лишь в каталоге, где размещены источники. Например, здесь будет ошибкой использование переменной $(LIBOBJS) или $(ALLOCA) в src/Makefile.am. Однако при использовании subdir-objects и AC_CONFIG_LIBOBJ_DIR не будет ошибки при использовании этих переменных в других каталогах. Например, src/Makefile.am можно изменить, как показано ниже.

```
# src/Makefile.am

AUTOMAKE_OPTIONS = subdir-objects
LDADD = $(LIBOBJS) $(ALLOCA)

bin_PROGRAMS = tool1 tool2 ...
tool1_SOURCES = ...
tool2_SOURCES = ...
```

Поскольку $(LIBOBJS) и $(ALLOCA) содержат имена объектных файлов, заканчивающиеся на .$(OBJEXT), они не подходят для библиотек Libtool (где ожидается расширение .lo). Взамен следует применять LTLIBOBJS и LTALLOCA. Переменная LTLIBOBJS задаётся автоматически программой Autoconf и её не следует задавать вручную, однако пока ещё для создания LTALLOCA нужно задавать ALLOCA вручную.

8.7. Переменные, используемые при сборке программ

Иногда полезно знать, какие из переменных Makefile использует программа Automake и в каком порядке (27.6. Порядок переменных флагов). Некоторые переменные наследуются от Autoconf (CC, CFLAGS, CPPFLAGS, DEFS, LDFLAGS, LIBS). Ниже перечислены переменные, которые Automake задаёт самостоятельно.

AM_CPPFLAGS

Значение этой переменной передаётся каждой компиляции, вызывающей препроцессор C, Переменная содержит список аргументов препроцессора. В ней следует указывать, например, опции -I и -D. Automake уже автоматически представляет некоторые опции -I в отдельной переменной, которая представляется каждой компиляции с вызовом препроцессора C. В частности, создаются опции -I., -I$(srcdir) и -I с указанием каталога, содержащего файл config.h (если применяется AC_CONFIG_HEADERS). Это можно отключить с помощью опции nostdinc.

Когда включаемый файл создаётся при сборке и не является частью дистрибутива, его расположение задаёт переменная $(builddir), а не $(srcdir). Это особенно важно для пакетов, использующих файлы заголовков, размещённые в подкаталогах, и разрешающих сборки за пределами дерева исходных кодов (2.2.6. Параллельная сборка (VPATH)). В таких случаях рекомендуется использовать пару опций -I, таких, например, как -Isome/subdir -I$(srcdir)/some/subdir или -I$(top_builddir)/some/subdir -I$(top_srcdir)/some/subdir. Отметим, что ссылка на дерево сборки должна предшествовать ссылке на дерево кода, поэтому случайно созданные в дереве кода файлы игнорируются.

AM_CPPFLAGS игнорируется при наличии переменной _CPPFLAGS на уровне программы или библиотеки.

INCLUDES

Выполняет ту же работу, что и переменная AM_CPPFLAGS' (или любая используемая на уровне цели переменная _CPPFLAGS). Это просто старое имя, переменную не следует применять, используя взамен AM_CPPFLAGS или _CPPFLAGS на уровне цели.

AM_CFLAGS

Эту переменную автор Makefile.am может использовать для передачи дополнительных аргументов компилятору C. В некоторых случаях вместо неё применяется переменная _CFLAGS на уровне программы или библиотеки.

COMPILE

Команда, применяемая для компиляции файлов C. Имя файла добавляется в конце для завершения команды.

AM_LDFLAGS

Эту переменную автор Makefile.am может использовать для передачи дополнительных флагов компоновщику. В некоторых случаях вместо неё применяется переменная _LDFLAGS на уровне программы или библиотеки.

LINK

Команда, используемая для компоновки программы C. Она уже включает -o $@ и обычные ссылки на переменные (например, CFLAGS). Переменная содержит также имена объектных файлов и библиотек для компоновки. Переменная не используется, если компоновщик переопределен переменной _LINK на уровне цели или флаги на уровне цели позволяют Automake задать такую переменную _LINK.

8.8. Поддержка Yacc и Lex

В Automake поддержка Yacc и Lex имеет ряд странностей.

Automake предполагает, что файлы .c, созданные yacc (или lex) должны именоваться на основе базового имени входного файла. Т. е. для исходного файла yacc с именем foo.y программа Automake будет создавать промежуточный файл foo.c (в отличие от более традиционного y.tab.c).

Расширение исходного файла yacc служит для определения полученных в результате файлов C, C++ или заголовков. Отметим, что файлы заголовков создаются лишь при использовании опции Yacc -d (см. ниже). Файлы с расширением .y будут преобразованы в файлы .c и заголовки .h, .yy преобразуются в .cc и .hh, .y++ - в c++ и h++, .yxx - в .cxx и .hxx, а .ypp - в .cpp и .hpp.

Точно так же исходные файлы lex могут служить для создания файлов C или C++, при этом распознаются расширения .l, .ll, .l++, .lxx и .lpp.

Не следует явно указывать промежуточные файлы (C или C++) в переменных SOURCES, указываются лишь источники. Промежуточные файлы, созданные yacc или lex, будут включаться в любой создаваемый дистрибутив и пользователю не потребуется yacc и lex.

Если найден файл yacc, в configure.ac должна быть определена переменная YACC. Это проще всего решить вызовом макроса AC_PROG_YACC. При вызове yacc программе передаются переменные AM_YFLAGS и YFLAGS. Последняя переменная является пользовательской, а первая предназначена для создателя файла Makefile.am.

AM_YFLAGS обычно служит для передачи опции -d программе yacc. Automake понимает это и автоматически настраивает правила обновления и распространения заголовочных файлов, созданных yacc -d[1]. Однако Automake не может знать, где будет применяться заголовок и нужно обеспечить его сборку до попытки использования. Обычно это требуется, чтобы работало отслеживание зависимостей, когда заголовок включён в другой файл. Решением является включение заголовочного файла в BUILT_SOURCES (9.4. Исходные файлы для сборки), как показано ниже.

```
BUILT_SOURCES = parser.h
AM_YFLAGS = -d
bin_PROGRAMS = foo
foo_SOURCES = ... parser.y ...
```

Если найден файл lex, в configure.ac должна быть определена переменная LEX, для чего можно использовать макрос AC_PROG_LEX, но рекомендуется применять AM_PROG_LEX (6.4. Макросы Autoconf из пакета Automake). При вызове lex программе передаются переменные AM_LFLAGS и LFLAGS. Первая предназначена для создателей Makefile.am, вторая - для пользователей.

При использовании режима AM_MAINTAINER_MODE (27.2. Сценарий missing и режим AM_MAINTAINER_MODE) правило повторной сборки для распространяемых источников Yacc и Lex применяется лишь при включении maintainer-mode или после удаления этих файлов.

При использовании источников lex или yacc команда automake -a автоматически устанавливает в пакет программу ylwrap (3.7. Программы, которые могут быть нужны automake). Эта программа применяется правилами сборки для переименования выходных файлов этих инструментов и позволяет включать множество источников yacc или lex в один каталог (выходное имя yacc зафиксировано, а при параллельной работе make может вызваться несколько экземпляров yacc одновременно). Для yacc простой блокировки недостаточно, поскольку в выводе yacc всегда применяются одни и те же символы, поэтому невозможно связать два анализатора yacc в одном исполняемом файле. Рекомендуется применять приведённые ниже правила переименования, заимствованные из gdb.

```
#define yymaxdepth c_maxdepth
#define yyparse c_parse
#define yylex   c_lex
#define yyerror c_error
#define yylval  c_lval
#define yychar  c_char
#define yydebug c_debug
#define yypact  c_pact
#define yyr1    c_r1
#define yyr2    c_r2
#define yydef   c_def
#define yychk   c_chk
#define yypgo   c_pgo
#define yyact   c_act
#define yyexca  c_exca
#define yyerrflag c_errflag
#define yynerrs c_nerrs
#define yyps    c_ps
#define yypv    c_pv
#define yys     c_s
#define yy_yys  c_yys
#define yystate c_state
#define yytmp   c_tmp
#define yyv     c_v
#define yy_yyv  c_yyv
#define yyval   c_val
#define yylloc  c_lloc
#define yyreds  c_reds
#define yytoks  c_toks
#define yylhs   c_yylhs
#define yylen   c_yylen
#define yydefred c_yydefred
#define yydgoto  c_yydgoto
#define yysindex c_yysindex
#define yyrindex c_yyrindex
#define yygindex c_yygindex
#define yytable  c_yytable
#define yycheck  c_yycheck
#define yyname   c_yyname
#define yyrule   c_yyrule
```

В каждом определении следует заменить префикс c_ удобным для вас префиксом. Эти определения работают с bison, byacc и традиционной программой yacc.

8.9. Поддержка C++

Automake полностью поддерживает C++. Любой пакет с кодом C++ должен задавать выходную переменную CXX в файле configure.ac. Простейшим вариантом является использование макроса AC_PROG_CXX. При наличии исходных файлов C++ определяется ряд дополнительных переменных.

CXX
Имя компилятора C++.

CXXFLAGS
Все флаги, передаваемые компилятору C++.

AM_CXXFLAGS
Вариант CXXFLAGS для сопровождающих пакет.

CXXCOMPILE
Команда, используемая для реальной компиляции исходного файла C++. Имя файла добавляется в конце.

CXXLINK
Команда, используемая для реальной компоновки исходного файла C++.

[1] *Следует отметить, что automake распознает -d в AM_YFLAGS лишь при отсутствии объединения (кластера) с другими опциями. Например, опция не будет распознана если AM_YFLAGS имеет значение -dt, но будет видна при значении -d -t или -t -d.*

8.10. Поддержка Objective C

Automake включает некоторую поддержку Objective C. Любой пакет с кодом Objective C должен задавать выходную переменную OBJC в файле configure.ac. Простейшим вариантом является использование макроса AC_PROG_OBJC. При наличии исходных файлов Objective C определяется ряд дополнительных переменных.

OBJC
Имя компилятора Objective C.
OBJCFLAGS
Все флаги, передаваемые компилятору Objective C.
AM_OBJCFLAGS
Вариант OBJCFLAGS для сопровождающих пакет.
OBJCCOMPILE
Команда, используемая для реальной компиляции исходного файла Objective C. Имя файла добавляется в конце.
OBJCLINK
Команда, используемая для реальной компоновки исходного файла Objective C.

8.11. Поддержка Objective C++

Automake включает некоторую поддержку Objective C++. Любой пакет с кодом Objective C++ должен задавать выходную переменную OBJCXX в файле configure.ac. Простейшим вариантом является использование макроса AC_PROG_OBJCXX. При наличии исходных файлов Objective C++ определяется ряд дополнительных переменных.

OBJCXX
Имя компилятора Objective C++.
OBJCXXFLAGS
Все флаги, передаваемые компилятору Objective C++.
AM_OBJCXXFLAGS
Вариант OBJCXXFLAGS для сопровождающих пакет.
OBJCXXCOMPILE
Команда, используемая для реальной компиляции исходного файла Objective C++. Имя файла добавляется в конце.
OBJCXXLINK
Команда, используемая для реальной компоновки исходного файла Objective C++.

8.12. Поддержка Unified Parallel C

Automake включает некоторую поддержку Unified Parallel C. Любой пакет с кодом Unified Parallel C должен задавать выходную переменную UPC в файле configure.ac. Простейшим вариантом является использование макроса AM_PROG_UPC. При наличии исходных файлов Unified Parallel C определяется ряд дополнительных переменных.

UPC
Имя компилятора Unified Parallel C.
UPCFLAGS
Все флаги, передаваемые компилятору Unified Parallel C.
AM_UPCFLAGS
Вариант UPCFLAGS для сопровождающих пакет.
UPCCOMPILE
Команда, используемая для реальной компиляции исходного файла Unified Parallel C. Имя файла добавляется в конце.
UPCLINK
Команда, используемая для реальной компоновки исходного файла Unified Parallel C.

8.13. Поддержка ассемблера

Automake включает некоторую поддержку ассемблерного кода для двух форм файлов - обычные (*.s) и после препроцессора CPP (*.S или *.sx).

В переменной CCAS указывается имя компилятора для ассемблерного кода. Этот компилятор должен работать подобно компилятору C и, в частности, должен воспринимать опции -c и -o. Значения переменных CCASFLAGS и AM_CCASFLAGS (или определение флагов на уровне цели) передаются компилятору. Для файлов после препроцессора используются ещё переменные DEFS, DEFAULT_INCLUDES, INCLUDES, CPPFLAGS, AM_CPPFLAGS.

Макрос autoconf AM_PROG_AS определяет переменные CCAS и CCASFLAGS (если они не заданы, в CCAS указывается компилятор C, а в CCASFLAGS - флаги компилятора C), но можно задать их самостоятельно.

Программа automake считает ассемблерным кодом лишь файлы .s, .S и .sx.

8.14. Поддержка Fortran 77

Automake полностью поддерживает Fortran 77. При наличии в пакете кода Fortran 77 должна включаться переменная F77 в файл configure.ac. Проще всего это сделать с помощью макроса AC_PROG_F77. При наличии исходных файлов Fortran 77 определяется ряд дополнительных переменных.

F77
Имя компилятора Fortran 77.
FFLAGS
Все флаги, передаваемые компилятору Fortran 77.
AM_FFLAGS
Вариант FFLAGS для сопровождающих пакет.
RFLAGS
Все флаги, передаваемые компилятору Ratfor.
AM_RFLAGS
Вариант RFLAGS для сопровождающих пакет.

F77COMPILE
Команда, используемая для реальной компиляции исходного файла Fortran 77. Имя файла добавляется в конце.
FLINK
Команда, используемая для реальной компоновки программы или библиотеки Fortran 77.
Automake может обрабатывать файлы после препроцессора Fortran 77 и Ratfor для их компиляции[1]. Имеется также некоторая поддержка создания программ и общих библиотек, где используется код Fortran 77 вместе с другими языками (8.14.3. Совмещение Fortran 77 с C и C++).

8.14.1. Предварительная обработка Fortran 77
N.f выполняется автоматически из N.F или N.r. Это правило лишь запускает препроцессор для преобразования исходного файла Fortran 77 или Ratfor в строгий код Fortran 77. Использование команды показано ниже.

.F
$(F77) -F $(DEFS) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS) $(AM_FFLAGS) $(FFLAGS)
.r
$(F77) -F $(AM_FFLAGS) $(FFLAGS) $(AM_RFLAGS) $(RFLAGS)

8.14.2. Компиляция файлов Fortran 77
N.o выполняется автоматически из N.f, N.F или N.r путём запуска компилятора Fortran 77. Использование команды показано ниже.

.f
$(F77) -c $(AM_FFLAGS) $(FFLAGS)
.F
$(F77) -c $(DEFS) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS) $(AM_FFLAGS) $(FFLAGS)
.r
$(F77) -c $(AM_FFLAGS) $(FFLAGS) $(AM_RFLAGS) $(RFLAGS)

8.14.3. Совмещение Fortran 77 с C и C++
Automake обеспечивает ограниченную поддержку создания программ и общих библиотек, в которых Fortran 77 применяется вместе с C и/или C++. Однако существует много вопросов, связанных со смешанным использованием Fortran 77 и других языков, которые не решаются в Automake, но обрабатываются другими пакетами[2].

Automake может помочь в решении двух задач.

1. Автоматический выбор компоновщика в зависимости от комбинации исходных кодов.
2. Автоматический выбор подходящих флагов компоновщика (например, -L и -l) для передачи выбранному автоматически компоновщику для привязки нужных библиотек Fortran 77.

   Эти дополнительные флаги компоновщика Fortran 77 передаются в выходной переменной FLIBS макросом Autoconf AC_F77_LIBRARY_LDFLAGS.

Если Automake видит, что программа или общая библиотека (из _PROGRAMS или _LTLIBRARIES) содержит исходный код Fortran 77 в комбинации с C и/или C++, требуется вызвать макрос AC_F77_LIBRARY_LDFLAGS из файла configure.ac, после чего появится $(FLIBS) в подходящей переменной _LDADD (для программ) или _LIBADD (для общих библиотек). Создатель файла Makefile.am должен убедиться, что $(FLIBS) присутствует в _LDADD или _LIBADD. Рассмотрим, например, приведённый ниже файл Makefile.am.

```
bin_PROGRAMS = foo
foo_SOURCES  = main.cc foo.f
foo_LDADD    = libfoo.la $(FLIBS)

pkglib_LTLIBRARIES = libfoo.la
libfoo_la_SOURCES  = bar.f baz.c zardoz.cc
libfoo_la_LIBADD   = $(FLIBS)
```

Здесь Automake будет настаивать на включении AC_F77_LIBRARY_LDFLAGS в configure.ac, а также выдавать предупреждение, если $(FLIBS) не указана в foo_LDADD и libfoo_la_LIBADD.

8.14.3.1. Выбор компоновщика
Для программы или библиотеки, включающей несколько языков, Automake выбирает компоновщик в соответствии с приведёнными ниже уровнями приоритета (в скобках указаны переменные, содержащие команду компоновщика).

1. Native Java (‘GCJLINK’)
2. Objective C++ (‘OBJCXXLINK’)
3. C++ (‘CXXLINK’)
4. Fortran 77 (‘F77LINK’)
5. Fortran (‘FCLINK’)
6. Objective C (‘OBJCLINK’)
7. Unified Parallel C (‘UPCLINK’)
8. C (‘LINK’)

Например, при компиляции кода Fortran 77, C и C++ в программу будет использован компоновщик C++. В таких случаях, если компоновщику C или Fortran 77 нужны какие-либо специальные библиотеки, не включённые в компоновщик C++, они должны быть добавлены в _LDADD и/или _LIBADD при создании файла Makefile.am.

[1]*Почти вся информация в последующих параграфах относится к предварительной обработке Fortran 77 и почти дословно скопирована из Catalogue of Rules (make) .*
[2]*Например, cfortran решает вопросы совместного использования языков и работает почти со всеми компиляторами Fortran 77, C и C++ практически на любой платформе. Однако cfortran пока не распространяется свободно, хотя это планируется в будущем.*

Automake при выборе компоновщика просматривает лишь имена файлов, указанные в переменных _SOURCES, а по умолчанию применяется компоновщик C. Иногда это неудобно, поскольку компоновка выполняется с библиотекой, написанной на другом языке, и нужно точнее выбрать компоновщик. В параграфе 8.3.5. Вспомогательные библиотеки Libtool приведены рекомендации по использованию nodist_EXTRA_..._SOURCES.

Переменная _LINK на уровне цели переопределяет описанный выше выбор, а заданные для цели флаги заставляют Automake создать переменную _LINK для цели в соответствии с выбранным языком, как показано выше.

8.15. Поддержка Fortran 9x

Automake включает поддержку Fortran 9x. Любой пакет с кодом Fortran 9x должен определять выходную переменную FC в файле configure.ac. Сделать это проще всего с помощью макроса AC_PROG_FC. При наличии источников Fortran 9x определяется ряд переменных, перечисленных ниже.

FC
Имя компилятора Fortran 9x.
FCFLAGS
Все флаги, передаваемые компилятору Fortran 9x.
AM_FCFLAGS
Вариант FCFLAGS для сопровождающих пакет.
FCCOMPILE
Команда, используемая для реальной компиляции файла Fortran 9x. Имя файла добавляется в конце.
FCLINK
Команда для реальной компоновки программы или общей библиотеки на Fortran 9x.

8.16. Компиляция файлов Java с помощью gcj

Automake поддерживает естественно скомпилированный код Java (с использованием gcj - внешнего интерфейса Java для GCC. Имеется также слабая поддержка компиляции Java в байт-код с использованием компилятора javac, но она будет прекращена (10.4. Компиляция байт-кода Java (устарела)).

Любой пакет, включающий код Java для компиляции, должен задавать переменную GCJ в файле configure.ac, а иногда нужно определить ещё переменную GCJFLAGS (в configure.ac или Makefile.am). Проще всего это сделать с помощью макроса AM_PROG_GCJ. По умолчанию программы с исходным кодом Java компонуются с помощью gcj.

Как обычно, содержимое AM_GCJFLAGS передаётся всякой компиляции, вызывающей gcj (в роли опережающего коммутатора при вызове для создания файлов .class используется AM_JAVACFLAGS). Если нужно передать gcj опции из Makefile.am, следует применять эту переменную, а не пользовательскую переменную GCJFLAGS.

Компилятор gcj можно использовать для файлов .java, .class, .zip или .jar.

При компоновке gcj требует, чтобы основной класс был задан с помощью опции --main=. Проще всего это сделать, используя для программы переменную _LDFLAGS.

8.17. Поддержка Vala

Automake обеспечивает начальную поддержку для Vala. Требуется valac версии 0.7.0 или выше, а пользователь должен применять GNU make.

```
foo_SOURCES = foo.vala bar.vala zardoc.c
```

Любой файл .vala, указанный в переменной _SOURCES будет компилироваться в код C компилятором Vala. Созданные файлы .c включаются в дистрибутив, поэтому конечному пользователю не требуется иметь компилятор Vala.

Automake распространяется с макросом Autoconf AM_PROG_VALAC, который находит компилятор Vala и может проверять номер версии.

AM_PROG_VALAC ([MINIMUM-VERSION], [ACTION-IF-FOUND], [ACTION-IF-NOT-FOUND])
Поиск компилятора Vala в путях PATH. При обнаружении компилятора переменная VALAC будет указывать на него (см. ниже). Макрос имеет 3 необязательных аргумента. Первый (при наличии) указывает минимальную версию компилятора Vala, требуемую для этого пакета. Если компилятор найден и удовлетворяет MINIMUM-VERSION, выполняется ACTION-IF-FOUND (по умолчанию ничего не делает), иначе выполняется ACTION-IF-NOT-FOUND (по умолчанию выводит предупреждение, если компилятор не найден или слишком старый.

При компиляции файлов Vala используется несколько дополнительных переменных, приведённых ниже.

VALAC
Абсолютный путь к компилятору Vala или просто valac, если при работе configure не найден подходящий компилятор Vala.
VALAFLAGS
Дополнительные аргументы для компилятора Vala.
AM_VALAFLAGS
Вариант 'VALAFLAGS для сопровождающих пакет.

```
lib_LTLIBRARIES = libfoo.la
libfoo_la_SOURCES = foo.vala
```

Отметим, что в настоящее время нет возможности задать *_VALAFLAGS на уровне цели (27.7. Переименование объектных файлов) для создания разных файлов C из одного файла Vala.

8.18. Поддержка других языков

Automake в настоящее время полностью поддерживает лишь языки C, C++ (8.9. Поддержка C++), Objective C (8.10. Поддержка Objective C), Objective C++ (8.11. Поддержка Objective C++), Fortran 77 (8.14. Поддержка Fortran 77), Fortran 9x (8.15. Поддержка Fortran 9x) и Java (8.16. Компиляция файлов Java с помощью gcj). для других языков имеется лишь начальная поддержка. Ограниченная возможность добавить свою поддержку языков обеспечивается за счёт правил обработки суффиксов (18.2. Обработка новых расширений имён. файлов).

8.19. Автоматическое отслеживание зависимостей

Разработчикам зачастую сложно обновлять Makefile.am при каждом изменении зависимостей включаемых файлов и Automake поддерживает автоматическое отслеживание зависимостей (2.2.12. Автоматическая проверка зависимостей).

Automake всегда использует для компиляции полные зависимости, включая системные заголовки. В модели Automake определение зависимостей должно быть побочным эффектом сборки. Поэтому зависимости определяются запуском всех компиляций через специальную оболочку depcomp, которая умеет заставить разные компиляторы C и C++ генерировать сведения о зависимостях в нужно формате. Команда automake -a устанавливает depcomp в каталог исходного кода. Если depcomp не понимает, как корректно вызвать компилятор, отслеживание зависимостей для сборки будет просто отключено.

Опыт работы с ранними версиями Automake показал, что генерировать зависимости только в системе сопровождающего ненадежно, поскольку конфигурации могут сильно различаться. Поэтому сейчас Automake определяет зависимости в процессе сборки. Автоматическое определение зависимостей можно отключить, поместив значение no-dependencies в переменную AUTOMAKE_OPTIONS или передав его в качестве аргумента AM_INIT_AUTOMAKE (предпочтительно). Можно также запустить automake с опцией -i. По умолчанию отслеживание зависимостей включено.

Можно также отключить отслеживание зависимостей опцией —disable-dependency-tracking в команде ./configure.

8.20. Поддержка расширений исполняемых файлов

На некоторых платформах (таких как Windows) для исполняемых файлов предполагается определённое расширение (.exe). Поэтому на таких платформах некоторые компиляторы (включая GCC) будут автоматически создавать foo.exe при запросе генерации foo. Automake обеспечивает поддержку такого подхода, но, к сожалению, не полную.

Следует понимать, что Automake переопределяет переменные вида

```
    bin_PROGRAMS = liver
в

    bin_PROGRAMS = liver$(EXEEXT)
```

Цели, создаваемые Automake получают расширение $(EXEEXT).

Переменные TESTS и XFAIL_TESTS (15.2. Простые тесты) также переопределяются, если они содержат имена файлов, объявленные как программы в том же Makefile (это полезно при указании некоторых программ из check_PROGRAMS в переменной TESTS).

Однако Automake не может применить эти переопределения к подстановкам configure. Это означает, что при условной сборке программ с такими подстановками файл configure.ac должен позаботиться о добавлении $(EXEEXT) при создании выходной переменной.

Иногда сопровождающие пакет создают явные правила компоновки для своих программ. Без поддержки расширений исполняемых файлов это просто - достаточно создать правило, целью которого является имя программы. Однако при поддержки расширений имён. исполняемых файлов вместо этого требуется добавлять суффикс $(EXEEXT).

Это может быть неудобно для сопровождающих, которые знают, что их пакет никогда не будет работать на платформе с расширением имени исполняемых файлов. Для них имеется опция no-exeext (17. Смена поведения Automake), которая отключает это свойство. Работает это достаточно коряво - при наличии опции no-exeext правило для цели foo в Makefile.am переопределит созданное automake правило для foo$(EXEEXT).

9. Другие производные объекты

Automake может работать с производными объектами, не являющимися программами C. Иногда поддержка сборки таких объектов должна задаваться явно, но Automake автоматически обрабатывает их установку и распространение.

9.1. Исполняемые сценарии

Можно задать и установить программы, которые являются сценариями. Такие программы указываются с использованием первичной переменной SCRIPTS. Распространение финальной, устанавливаемой формы сценария обычно выглядит в Makefile как

```
    # Install my_script in $(bindir) and distribute it.
    dist_bin_SCRIPTS = my_script
```

По умолчанию сценарии не распространяются и, как показано выше, для включения в дистрибутив можно использовать префикс dist_. Сценарии могут устанавливаться в bindir, sbindir, libexecdir, pkglibexecdir или pkgdatadir. Сценарии, которые не нужно устанавливать, могут быть указаны в переменной noinst_SCRIPTS, а нужные лишь для make check следует указывать в check_SCRIPTS.

Для сборки сценария в Makefile.am нужно включить подходящие правила. Например, сама программа automake является сценарием Perl, который создаётся из automake.in. Обработка показана ниже.

```
    bin_SCRIPTS = automake
    CLEANFILES = $(bin_SCRIPTS)
    EXTRA_DIST = automake.in

    do_subst = sed -e 's,[@]datadir[@],$(datadir),g' \
               -e 's,[@]PERL[@],$(PERL),g' \
               -e 's,[@]PACKAGE[@],$(PACKAGE),g' \
               -e 's,[@]VERSION[@],$(VERSION),g' \
               ...

    automake: automake.in Makefile
            $(do_subst) < $(srcdir)/automake.in > automake
            chmod +x automake
```

Сценарии, для которых было представлено правило сборки, нужно удалить явно с помощью CLEANFILES (13. Очистка), а их исходные файлы следует распространять обычно с помощью EXTRA_DIST (14.1. Основы дистрибутивов).

Другим способом сборки сценариев является из обработка из configure с макросом AC_CONFIG_FILES. В этом случае Automake знает, какие файлы следует очищать и распространять и как должны выглядеть правила пересборки. Например, если configure.ac включает строку

```
AC_CONFIG_FILES([src/my_script], [chmod +x src/my_script])
```

для сборки src/my_script из src/my_script.in, то src/Makefile.am для установки этого сценария в $(bindir) может иметь вид

```
bin_SCRIPTS = my_script
CLEANFILES = $(bin_SCRIPTS)
```

Здесь не нужна переменная EXTRA_DIST или правило для сборки, Automake выводит их из AC_CONFIG_FILES (6.1. Конфигурационные требования). Переменная CLEANFILES остаётся полезной, поскольку Automake по умолчанию очищает цели AC_CONFIG_FILES в distclean, а не в clean.

Хотя это выглядит проще, сборка сценария таким способом имеет один недостаток - переменные каталогов (такие как $(datadir)) не полностью преобразуются и могут указывать на другие переменные каталогов.

9.2. Заголовочные файлы

Заголовочные файлы, которые нужно установить, задаются переменными HEADERS. Заголовки могут устанавливаться в includedir, oldincludedir, pkgincludedir или любой другой заданный каталог (3.3. Схема именования). Например,

```
include_HEADERS = foo.h bar/bar.h
```

будет устанавливать два файла $(includedir)/foo.h и $(includedir)/bar.h. Поддерживаются также префиксы nobase_ (7.3. Другая модель организации каталогов).

```
nobase_include_HEADERS = foo.h bar/bar.h
```

Обычно требуется устанавливать лишь файлы, сопровождающие устанавливаемые библиотеки. Заголовки, используемые программами или дополнительными библиотеками, не устанавливаются. Для таких заголовков можно использовать переменную noinst_HEADERS. Однако, если заголовок относится к одной дополнительной библиотеке или программе, рекомендуется указывать его в переменной _SOURCES этой библиотеки или программы (8.1.1. Указание источников программы) вместо noinst_HEADERS. Это будет более понятно при рассмотрении файла Makefile.am. Переменная noinst_HEADERS подходит для использования в каталоге, содержащем лишь заголовки без связанной с ними программы или библиотеки.

Все файлы заголовков должны быть указаны в переменной _SOURCES или _HEADERS, а пропущенные файлы не будут включены в дистрибутив.

Для заголовочных файлов, которые собраны и не должны распространяться, используется префикс nodist_ в форме nodist_include_HEADERS или nodist_prog_SOURCES. Если создаваемые заголовки нужны при сборке, они должны быть подготовлены заранее (9.4. Исходные файлы для сборки).

9.3. Независимые от архитектуры файлы

Automake поддерживает установку файлов данных с помощью переменных DATA. Данные могут помещаться в datadir, sysconfdir, sharedstatedir, localstatedir или pkgdatadir. По умолчанию файлы данных не включаются в дистрибутив. Можно использовать префикс dist_ для настройки этого на уровне переменных. Ниже показано, как Automake объявляет дополнительные файлы данных.

```
dist_pkgdata_DATA = clean-kr.am clean.am ...
```

9.4. Исходные файлы для сборки

Поскольку автоматическое отслеживание зависимостей в Automake является побочным результатом компиляции (8.19. Автоматическое отслеживание зависимостей), возникает вопрос начальной настройки (bootstrap) - цель не следует собирать до выполнения зависимостей, но эти зависимости неизвестны, пока цель не скомпилирована.

Обычно проблемы не возникает, поскольку зависимости являются распространяемыми источниками и не нуждаются в сборке. Предположим, что файл foo.c включает foo.h. При первой компиляции foo.o, программа make знает лишь, о зависимости foo.o от foo.c. В качестве побочного результатом этой компиляции depcomp записывает зависимость от foo.h, чтобы учесть её при последующем вызове make. Очевидно, что в этом случае проблемы не возникает - или файла не существует и его нужно собрать (безотносительно к зависимостям), или имеются точные зависимости и ими можно воспользоваться для решения вопроса о повторной сборке foo.o.

Другая ситуация возникает при отсутствии foo.h на момент первого запуска make. Например, может быть правило для сборки foo.h. На этот раз собрать file.o не получится, поскольку компилятор не найдёт foo.h. Программе make не удастся выполнить правило сборки foo.h в первую очередь из-за отсутствия данных о зависимостях. Обойти эту проблему позволяет переменная BUILT_SOURCES. Исходный файл из BUILT_SOURCES создаётся по команде make all или make check (или даже make install) до обработки других целей. Однако такой исходный файл не будет скомпилирован, пока это не будет запрошено явно в той или иной переменной _SOURCES.

Для завершения примера можно было бы использовать BUILT_SOURCES = foo.h, чтобы обеспечить сборку foo.h до всех других целей (включая foo.o) по команде make all или make check. Имя переменной BUILT_SOURCES не совсем точно, поскольку любой файл, который должен быть создан с начале процесса сборки, может быть указан в этой переменной. Более того, все источники сборки не обязательно перечисляются в BUILT_SOURCES. Например, созданный файл .c не присутствует в BUILT_SOURCES (пока не включён другим источником), поскольку он не известен как зависимость связанного объекта.

Подчеркнём, что BUILT_SOURCES учитывается лишь командами make all, make check и make install. Это означает, что нельзя собрать конкретную цель (например, make foo) в чистом дереве, если она зависит от сборки источников. Однако предшествующий запуск make all решает эту задачу, поскольку зависимости становятся доступными.

9.4.1. Пример сборки источников

Предположим, что файл foo.c включает заголовок bindir.h, который зависит от установки и не распространяется (нужно собирать). Этот файл bindir.h определяет макрос препроцессора bindir для назначения переменной make bindir

(наследуется от configure). Ниже рассмотрено несколько вариантов сборки. Этот список не учитывает всех вариантов работы со сборкой источников, но представляет некоторые идеи по решению задачи.

Первая сборка

Здесь рассматривается вопрос начальной настройки (bootstrap), упомянутая выше (9.4. Исходные файлы для сборки).

Ниже приведён предварительный файл Makefile.am.

```
# This won't work.
bin_PROGRAMS = foo
foo_SOURCES = foo.c
nodist_foo_SOURCES = bindir.h
CLEANFILES = bindir.h
bindir.h: Makefile
        echo '#define bindir "$(bindir)"' >$@
```

Это не будет работать, поскольку Automake не знает, что foo.c включает bindir.h. Напомним, что автоматическое отслеживание зависимостей является побочным результатом компиляции, поэтому зависимости foo.o будут известны лишь после компиляции foo.o (8.19. Автоматическое отслеживание зависимостей). Симптомы показаны ниже.

```
% make
source='foo.c' object='foo.o' libtool=no \
depfile='.deps/foo.Po' tmpdepfile='.deps/foo.TPo' \
depmode=gcc /bin/sh ./depcomp \
gcc -I. -I. -g -O2 -c `test -f 'foo.c' || echo './'`foo.c
foo.c:2: bindir.h: No such file or directory
make: *** [foo.o] Error 1
```

В этом примере bindir.h не распространяется, не устанавливается и даже вовремя не создаётся. Можно задаться вопросом осмысленности строки nodist_foo_SOURCES = bindir.h. Она просто указывает, что bindir.h является источником для foo, поэтому, например, его следует проверять при генерации тегов (18.1. Взаимодействие с etags).

Использование BUILT_SOURCES

Решение состоит в создании файла bindir.h заранее с использованием предназначенной для этого переменной BUILT_SOURCES (9.4. Исходные файлы для сборки).

```
bin_PROGRAMS = foo
foo_SOURCES = foo.c
nodist_foo_SOURCES = bindir.h
BUILT_SOURCES = bindir.h
CLEANFILES = bindir.h
bindir.h: Makefile
        echo '#define bindir "$(bindir)"' >$@
```

Посмотрим сначала, как создаётся bindir.h

```
% make
echo '#define bindir "/usr/local/bin"' >bindir.h
make  all-am
make[1]: Entering directory `/home/adl/tmp'
source='foo.c' object='foo.o' libtool=no \
depfile='.deps/foo.Po' tmpdepfile='.deps/foo.TPo' \
depmode=gcc /bin/sh ./depcomp \
gcc -I. -I. -g -O2 -c `test -f 'foo.c' || echo './'`foo.c
gcc  -g -O2   -o foo  foo.o
make[1]: Leaving directory `/home/adl/tmp'
```

Однако, как было отмечено выше, BUILT_SOURCES применяется только к целям all, check и installи по-прежнему не будет работать для цели make foo

```
% make clean
test -z "bindir.h" || rm -f bindir.h
test -z "foo" || rm -f foo
rm -f *.o
% : > .deps/foo.Po # Suppress previously recorded dependencies
% make foo
source='foo.c' object='foo.o' libtool=no \
depfile='.deps/foo.Po' tmpdepfile='.deps/foo.TPo' \
depmode=gcc /bin/sh ./depcomp \
gcc -I. -I. -g -O2 -c `test -f 'foo.c' || echo './'`foo.c
foo.c:2: bindir.h: No such file or directory
make: *** [foo.o] Error 1
```

Запись зависимостей вручную

Обычно BUILT_SOURCES устраивает, поскольку команды вида make foo просто не используются до make all, как в предыдущем примере. Однако при необходимости можно отказаться от использования BUILT_SOURCES и записать зависимости явно в Makefile.am.

```
bin_PROGRAMS = foo
foo_SOURCES = foo.c
nodist_foo_SOURCES = bindir.h
foo.$(OBJEXT): bindir.h
CLEANFILES = bindir.h
bindir.h: Makefile
        echo '#define bindir "$(bindir)"' >$@
```

Нет нужды явно указывать все зависимости foo.o, достаточно тех, которые требуется выполнить. Если зависимость уже выполнена, это не помешает первой компиляции и будет записано обычным кодом отслеживания зависимостей. Отметим, что после первой компиляции код отслеживания запишет также зависимость между foo.o и bindir.h, поэтому заданная явно зависимость полезна лишь при первой сборке.

Добавление таких явных зависимостей может создавать опасность, связанную с тем, что Automake не пытается переопределять правила. Правило foo.$(OBJEXT): bindir.h заменяет любое правило, которое Automake может пожелать вывести для сборки foo.$(OBJEXT). В данном случае это работает, поскольку Automake не имеет вывода для какой-либо цели foo.$(OBJEXT):, полагаясь на правило суффиксов (т. е., .c.$(OBJEXT):). При явном задании зависимостей всегда следует проверять файл Makefile.in.

Сборка bindir.h из configure

Можно задать макрос препроцессора из сценария configure с помощью файла config.h или путём обработки файла bindir.h.in с помощью макроса AC_CONFIG_FILES. На этом этапе должно быть ясно, что сборка bindir.h из configure работает для данного примера. Файл bindir.h будет создан до сборки какой-либо цели, поэтому проблемы с зависимостями не возникнет. Сжатая форма Makefile представлена ниже и в ней даже не упоминается bindir.h.

```
bin_PROGRAMS = foo
foo_SOURCES = foo.c
```

Однако не всегда возможно собрать источник из configure, особенно при генерации такого источника инструментом, который также нужно собрать.

Сборка bindir.c без bindir.h

Другая идея заключается в определении bindir как переменной или функции, экспортируемой из bindir.o, и сборка bindir.c вместо bindir.h.

```
noinst_PROGRAMS = foo
foo_SOURCES = foo.c bindir.h
nodist_foo_SOURCES = bindir.c
CLEANFILES = bindir.c
bindir.c: Makefile
        echo 'const char bindir[] = "$(bindir)";' >$@
```

Файл bindir.h содержит просто определения переменных и его сборка не требуется, поэтому ошибок возникать не должно. Файл bindir.o всегда зависит от bindir.c, поэтому bindir.c будет собран первым.

Что лучше?

Здесь нет панацеи и каждый из вариантов имеет свои плюсы и минусы.

Переменную BUILT_SOURCES нельзя использовать для сборки make foo в «чистом» дереве.

Явное добавление зависимостей может приводить к ошибочным переопределениям правил Automake.

Сборка файлов из ./configure возможна не всегда, равно как и преобразование файлов .h в .c.

10. Другие инструменты GNU

Поскольку основной задачей Automake является создание файлов Makefile.in для использования программами GNU, этот инструмент использует другие средства GNU.

10.1. Emacs Lisp

Automake обеспечивает некоторую поддержку Emacs Lisp. Основным применением переменной LISP является хранение списка файлов .el. Возможными префиксами для основного имени являются lisp_ и noinst_. При задании lisp_LISP требуется вызвать макрос AM_PATH_LISPDIR (6.4. Макросы Autoconf из пакета Automake) из configure.ac.

Источники Lisp по умолчанию не распространяются и для контроля распространения можно использовать переменную LISP с префиксом dist_ (dist_lisp_LISP или dist_noinst_LISP).

Automake будет выполнять байтовую компиляцию для всех исходных файлов Emacs Lisp с использованием Emacs, найденного макросом AM_PATH_LISPDIR 9если программа найдена). При компиляции флаги, заданные разработчиком в AM_ELCFLAGS и пользователем в ELCFLAGS, передаются Emacs. Результаты байтовой компиляции файлов Emacs Lisp не переносимы между разными версиями Emacs, поэтому не следует её использовать при наличии в системе нескольких версий Emacs. Кроме того, для многих пакетов байтовая компиляция не даёт преимуществ. Тем не менее, её использование рекомендуется для источников Emacs Lisp.

Есть для способа избежать байтовой компиляции и рекомендуется приведённая ниже конструкция.

```
lisp_LISP = file1.el file2.el
ELCFILES =
```

ELCFILES является внутренней переменной Automake, в которой обычно указаны все файлы .elc, для которых нужна байтовая компиляция. Automake задаёт ELCFILES автоматически из lisp_LISP. Сброс этой переменной автоматически отключает байтовую компиляцию.

Начиная с Automake 1.8 рекомендуется использовать lisp_DATA, как показано ниже.

```
lisp_DATA = file1.el file2.el
```

Отметим, что эти конструкции не эквивалентны. _LISP не будет устанавливать файлы при отсутствии Emacs, а _DATA устанавливает эти файлы всегда.

10.2. Gettext

При наличии макроса AM_GNU_GETTEXT в файле configure.ac Automake включает поддержку GNU gettext для работы с разными естественными языками. Для поддержки gettext в Automake требуется добавление в пакет одного или двух подкаталогов - po и возможно intl. Последний нужен, если AM_GNU_GETTEXT не вызывается с аргументом external или применяется AM_GNU_GETTEXT_INTL_SUBDIR. Automake обеспечивает наличие этих каталогов и их включение в SUBDIRS.

10.3. Libtool

Automake поддерживает GNU Libtool с помощью переменной LTLIBRARIES (8.3. Сборка общих библиотек).

10.4. Компиляция байт-кода Java (устарела)

Automake обеспечивает минимальную поддержку компиляции байт-кода Java с использованием переменной JAVA (в дополнение к поддержке компиляции Java в машинный код, 8.16. Компиляция файлов Java с помощью gcj). Большинство описанных в этом параграфе возможностей и сам интерфейс устарели. В новых выпусках Automake будут предприняты попытки обеспечить более эффективный интерфейс, но вероятность его совместимости с

прежними версиями весьма мала и описанные здесь возможности могут быть удалены через некоторое время после внедрения нового интерфейса (если он будет). В любом случае возможности переменной JAVA больше не развиваются даже для исправления ошибок.

Любой файл .java, указанный в переменной _JAVA, будет компилироваться с помощью JAVAC в процессе сборки. По умолчанию файлы .java не включаются в дистрибутив и для распространения их следует использовать префикс dist_.

Ниже приведён пример распространения файлов .java и установки файлов .class, полученных при компиляции.

```
javadir = $(datadir)/java
dist_java_JAVA = a.java b.java ...
```

В настоящее время Automake применяет ограничение, в соответствии с которым можно использовать лишь одну переменную _JAVA в данном файле Makefile.am. Это обусловлено тем, что обычно невозможно узнать, какой файл .class из какого .java, поэтому нет возможности узнать, какие файлы куда устанавливать. Например, файл .java может определять множество классов и имена файлов .class невозможно определить без анализа файла .java.

При компиляции источников Java используется несколько дополнительных переменных.

JAVAC
Имя компилятора Java (по умолчанию javac).

JAVACFLAGS
Флаги для передачи компилятору (3.6. Пользовательские переменные).

AM_JAVACFLAGS
Дополнительные флаги, передаваемые компилятору Java. Эту переменную (а не JAVACFLAGS) следует использовать при необходимости включения флагов компилятора Java в Makefile.am.

JAVAROOT
Значение этой переменной передаётся в опции -d компилятору javac (по умолчанию $(top_builddir)).

CLASSPATH_ENV
Выражение оболочки (shell), используемое для установки переменной среды CLASSPATH в командной строке javac. В будущем обработка пути к классам может измениться.

10.5. Python

Automake поддерживает компиляцию Python с помощью переменной PYTHON, которая обычно устанавливается вызовом макроса AM_PATH_PYTHON в configure.ac, и используется, как в приведённом ниже фрагменте Makefile.am.

```
python_PYTHON = tree.py leave.py
```

Для всех файлов, указанных в переменной _PYTHON будет выполнена байт-компиляция с помощью py-compile в процессе установки. Программа py-compile на деле создаёт стандартные (.pyc) и оптимизированные (.pyo) варианты байт-кода исходных файлов. Отметим, что в результате компиляции в процессе установки файлы, перечисленные в noinst_PYTHON, компилироваться не будут. Исходные файлы Python по умолчанию включаются в дистрибутив и для их исключения следует применять префикс nodist_ (как nodist_python_PYTHON).

Automake распространяется с макросом Autoconf AM_PATH_PYTHON, который определяет некоторые связанные с Python переменные каталогов (см. ниже). При вызове AM_PATH_PYTHON из файла configure.ac можно использовать переменные python_PYTHON или pkgpython_PYTHON для перечисления исходных файлов Python в Makefile.am в зависимости от того, следует ли их устанавливать (см. pythondir и pkgpythondir ниже).

Макрос AM_PATH_PYTHON ([VERSION], [ACTION-IF-FOUND], [ACTION-IF-NOT-FOUND])
Отыскивает в системе интерпретатор Python. Макрос принимает 3 необязательных параметра. При наличии первого аргумента он указывает минимальную версию Python, требуемую для пакета и AM_PATH_PYTHON будет пропускать интерпретаторы Python с версией ниже VERSION. Если интерпретатор найден и соответствует VERSION, выполняется ACTION-IF-FOUND. В остальных случаях применяется ACTION-IF-NOT-FOUND. Если действие ACTION-IF-NOT-FOUND не задано, как в приведённом ниже примере, по умолчанию работа сценария configure прерывается.

```
AM_PATH_PYTHON([2.2])
```

Это нормально, когда Python является абсолютным требованием для пакета. Если же Python ≥ 2.5 является лишь опцией для пакета, AM_PATH_PYTHON можно вызвать в форме

```
AM_PATH_PYTHON([2.5],, [:])
```

Если переменная PYTHON установлена при вызове AM_PATH_PYTHON, проверяться будет лишь заданный интерпретатор Python.

Макрос AM_PATH_PYTHON создаёт приведённые ниже переменные на основе установки Python, найденной при настройке.

PYTHON
Имя исполняемого файла Python или :, если подходящий интерпретатор не найден. В предположении использования ACTION-IF-NOT-FOUND (иначе ./configure прервёт работу при отсутствии Python), значение PYTHON можно использовать для задания условий отключения сборки соответствующих частей, как показано ниже.

```
AM_PATH_PYTHON(,, [:])
AM_CONDITIONAL([HAVE_PYTHON], [test "$PYTHON" != :])
```

PYTHON_VERSION
Номер версии интерпретатора Python в форме MAJOR.MINOR (например, 2.5) - текущее значение sys.version[:3].

PYTHON_PREFIX
Строка ${prefix}, которая может использоваться в будущей работе, где потребуется содержимое Python sys.prefix, но общепринято всегда использовать значение из configure.

PYTHON_EXEC_PREFIX
Строка '${exec_prefix}, которая может использоваться в будущей работе, где потребуется содержимое Python sys.exec_prefix, но общепринято всегда использовать значение из configure.

PYTHON_PLATFORM
Каноническое имя, используемое Python для описания операционной системы, как указано в sys.platform. Это значение иногда требуется при сборке расширений Python.

pythondir
Имя каталога для размещения site-packages в стандартном дереве установки Python.

pkgpythondir
Каталог в pythondir, который указывается после пакета, т. е. $(pythondir)/$(PACKAGE).

pyexecdir
Каталог, в который следует устанавливать расширения Python (общие библиотеки). Модули расширения, написанные на C, можно объявлять Automake как показано ниже.

```
pyexec_LTLIBRARIES = quaternion.la
quaternion_la_SOURCES = quaternion.c support.c support.h
quaternion_la_LDFLAGS = -avoid-version -module
```

pkgpyexecdir
Вспомогательная переменная, задаваемая в форме $(pyexecdir)/$(PACKAGE).

Все указанные переменные каталогов, которые начинаются с префикса ${prefix} или ${exec_prefix}, не преобразуются. Это хорошо работает в Makefiles, но осложняет использование переменных в configure. Такой подход диктуют стандарты кодирования GNU и пользователь может запустить make prefix=/foo install. В руководстве Autoconf этот вопрос рассмотрен более подробно. См. также параграф 27.10. Установка в жёстко заданные места.

11. Сборка документации

В настоящее время Automake поддерживает Texinfo и страницы man.

11.1. Texinfo

Если в текущем каталоге есть исходные файлы Texinfo, нужно объявить их в переменной TEXINFOS. Обычно файлы Texinfo преобразуются в info, поэтому здесь как правило должна применяться переменная info_TEXINFOS. Источники Texinfo должны иметь расширение .texi. Automake воспринимает также расширения .txi и .texinfo, но их использование не рекомендуется и будет вызывать предупреждения.

Automake генерирует файлы для сборки .info, .dvi, .ps, .pdf и .html из источников Texinfo. В соответствии со стандартами кодирования GNU лишь файлы .info собираются по команде make all и устанавливаются по команде make install (если не задано no-installinfo). Файлы .info распространяются автоматически, поэтому наличие Texinfo не требуется для установки пакета. Следуем отметить, что созданные файлы .info в отличие от других по умолчанию помещаются в srcdir, а не в builddir. Это можно изменить с помощью опции info-in-builddir.

Документация в других форматах может быть создана командами make dvi, make ps, make pdf, make html и явно установлена командами make install-dvi, make install-ps, make install-pdf и make install-html. Команда make uninstall удаляет все - установленную по умолчанию документацию и результаты приведённых выше команд. Все эти цели могут быть преобразованы с помощью правил -local (23.1. Расширение правил Automake).

Если файл .texi включает (@include) version.texi, этот файл будет создан автоматически. Файл version.texi определяет 4 флага Texinfo, которые можно использовать - @value{EDITION}, @value{VERSION}, @value{UPDATED} и @value{UPDATED-MONTH}.

EDITION
VERSION
Обе эти переменные содержат номер версии программы.

UPDATED
Дата изменения основного файла .texi.

UPDATED-MONTH
Название месяца, когда был изменён основной файл .texi.

Для поддержки version.texi нужен сценарий mdate-sh, который предоставляется Automake и включается автоматически при вызове automake с опцией --add-missing.

Когда имеется множество файлов Texinfo и нужно использовать свойство version.texi, требуется отдельный файл версии для каждого файла Texinfo. Automake будет обрабатывать любое включение в файл Texinfo, соответствующее vers*.texi, как автоматически созданный файл версии. Иногда файл info реально зависит от нескольких файлов .texi. Например, в GNU Hello файл hello.texi включает fdl.texi. Можно указать Automake такую зависимость с помощью переменной TEXI_TEXINFOS. Например, для GNU это имеет вид

```
info_TEXINFOS = hello.texi
hello_TEXINFOS = fdl.texi
```

По умолчанию Automake требует наличия файла texinfo.tex в одном каталоге с Makefile.am, где указаны файлы .texi. При использовании AC_CONFIG_AUX_DIR в configure.ac, файл texinfo.tex отыскивается. В обоих случаях automake затем представляет texinfo.tex, если задана опция --add-missing и пытается распространить этот файл. Однако при установке переменной TEXINFO_TEX (см. ниже) она переопределяет местоположение файла и отключает его установку в источники, а также распространение.

Опция no-texinfo.tex позволяет обойти требование наличия texinfo.tex. Использование переменной TEXINFO_TEX предпочтительно, однако данные опция позволяет работать с целями dvi, ps и pdf, поэтому сохранена.

Automake создаёт правило install-info и некоторые люди его используют. По умолчанию страницы info устанавливаются командой make install, поэтому команда make install-info смысла не имеет. Установку можно отключить опцией no-installinfo и файлы .info не будут устанавливаться по умолчанию и пользователю придётся явно задать make install-info.

По умолчанию команды make install-info и make uninstall-info пытаются запустить install-info (если программа доступна) для обновления (или создания/удаления) индекса ${infodir}/dir. Если это не нужно, можно экспортировать переменную AM_UPDATE_INFO_DIR со значением no.

Ниже приведены переменные, используемые правилами сборки Texinfo.

MAKEINFO
Имя программы, вызываемой для сборки файлов .info, которая задаётся Automake. Если программа makeinfo найдена в системе, она будет использоваться по умолчанию. В противном случае применяется missing.

MAKEINFOHTML
Команда для сборки файлов .html, которую Automake задаёт как $(MAKEINFO) --html.

MAKEINFOFLAGS
Пользовательские флаги, передаваемые при каждом вызове $(MAKEINFO) и $(MAKEINFOHTML). Эта пользовательская переменная (3.6. Пользовательские переменные) не предполагается в файлах Makefile и может применяться пользователем для указания дополнительных флагов.

AM_MAKEINFOFLAGS
AM_MAKEINFOHTMLFLAGS
Флаги сопровождающего, передаваемые при каждом вызове makeinfo. В отличие от MAKEINFOFLAGS, эти переменные задаются сопровождающим в файле Makefile.am. $(AM_MAKEINFOFLAGS) передаётся makeinfo при сборке файлов .info, $(AM_MAKEINFOHTMLFLAGS) применяется при сборке .html. Например, приведённая ниже строка задаёт создание одного файла .html для руководства без разделителей тем (node).

```
AM_MAKEINFOHTMLFLAGS = --no-headers --no-split
```

AM_MAKEINFOHTMLFLAGS по умолчанию имеет значение $(AM_MAKEINFOFLAGS), это означает, что задание AM_MAKEINFOFLAGS без указания AM_MAKEINFOHTMLFLAGS будет влиять на сборку .info и .html.

TEXI2DVI
Имя команды преобразования .texi в .dvi. По умолчанию это texi2dvi (сценарий из пакета Texinfo).

TEXI2PDF
Имя команды преобразования .texi в .pdf' file. По умолчанию это $(TEXI2DVI) --pdf --batch.

DVIPS
Имя команды для сборки файла .ps из .dvi (по умолчанию dvips).

TEXINFO_TEX
Если пакет содержит файлы Texinfo в разных каталогах, эта переменная говорит Automake, где находится канонический файл texinfo.tex для пакета. В переменной должен указываться относительный путь от текущего файла Makefile.am к texinfo.tex, например,

```
TEXINFO_TEX = ../doc/texinfo.tex
```

11.2. Страницы Man

Пакет может также включать страницы man, которые объявляются в переменной MANS (обычно используется man_MANS). Man-страницы автоматически устанавливаются в корректный каталог mandir на основе расширения. Расширения, подобные .1c, обрабатываются путём поиска и использования найденного расширения для выбора нужного каталога mandir. Имена разделов обозначаются цифрами от 0 до 9, а также буквами l и n.

Иногда разработчики предпочитают именовать страницы руководства в форме foo.man внутри дерева кода, а затем переименовывать их с подходящим суффиксом (например, foo.1) при установке. Automake поддерживает такой режим. Для корректного раздела SECTION имеется соответствующий каталог manSECTIONdir и переменная _MANS. Указанные в такой переменной файлы устанавливаются в соответствующий раздел. Если файл уже имеет корректный суффикс, он устанавливается как есть, в остальных случаях суффикс устанавливается в соответствии с разделом. Например, в случае

```
man1_MANS = rename.man thesame.1 alsothesame.1c
```

файл rename.man будет переименован при установке в rename.1, а имена остальных сохранятся.

По умолчанию man-страницы устанавливаются командой make install. Однако проект GNU не требует страниц man и многие сопровождающие не тратят сил на поддержку их актуальности. В таких случаях опция no-installman будет отключать установку страниц man по умолчанию. Пользователь может явно установить их командой make install-man.

Для быстрой установки с большим числом файлов предпочтительно использовать manSECTION_MANS, а не man_MANS и файлы, которые не нужно переименовывать.

Man-страницы в настоящее время не считаются источниками, поскольку автоматическая их генерация не является общепринятой. В результате они не включаются в дистрибутив автоматически . Однако это можно изменить с помощью префикса dist_. Ниже показано, как распространить и установить две страницы man пакета GNU cpio, который включает документацию Texinfo и страницы man.

```
dist_man_MANS = cpio.1 mt.1
```

Префикс nobase_ для страниц man не имеет смысл, поэтому он не разрешён.

Исполняемые файлы и страницы man можно переименовать после установки (2.2.9. Переименование программ при установке). Для страниц man это можно предотвратить с помощью префикса notrans_. Например, для исполняемого файла foo, разрешающего доступ к библиотеке foo из командной строки можно избежать переименования foo.3 в виде

```
man_MANS = foo.1
notrans_man_MANS = foo.3
```

Префикс notrans_ должен быть задан впереди при использовании с dist_ или nodist_ (14.2. Тонкая настройка распространения). Например,

```
notrans_dist_man3_MANS = bar.3
```

12. Установка

Automake обрабатывает детали реальной установки программы после её сборки. Все файлы, указанные первичными переменными, устанавливаются в нужные места по команде make install.

12.1. Основы инсталляции

Файл, названный в первичной переменной, устанавливается путём копирования результата сборки в подходящий каталог с использованием базового имени

```
bin_PROGRAMS = hello subdir/goodbye
```

В этом примере hello и goodbye устанавливаются в $(bindir).

Иногда полезно избегать этапа базового имени при установке. Например, может присутствовать множество заголовочных файлов в подкаталогах дерева источников, расположенных точно так, как их нужно установить. В таких случаях можно использовать префикс nobase_ для исключения этапа базового имени. Например,

```
nobase_include_HEADERS = stdio.h sys/types.h
```

будет устанавливать stdio.h в $(includedir) а types.h в $(includedir)/sys.

Для большинства типов файлов Automake будет устанавливать множество файлов в один приём., избегая в то же время слишком длинных команд (3.4. Ограничение размера команд). Поскольку некоторые программы install не устанавливают один и тот же файл дважды за вызов, может потребоваться проверка уникальности имён. файлов, указанных в одной переменной (например, nobase_include_HEADERS).

Не следует полагаться на порядок перечисления файлов в переменной. Аналогично при параллельной работе make не следует полагаться на порядок установки даже для разнотипных файлов (за исключением зависимости библиотек).

12.2. Две части установки

Automake создаёт отдельные правила install-data и install-exec, если установщик инсталлирует программу на разные машины в общей структурой каталогов. Это позволяет установить независимые от машины компоненты в один приём. Цель install-exec служит для установки зависимых от платформы файлов, а install-data - для независимых. Цель install зависит от обеих отмеченных целей. Хотя Automake пытается автоматически разделить объекты по категориям, автор Makefile.am в конечном итоге отвечает за корректность этого деления.

Переменные со стандартными префиксами data, info, man, include, oldinclude, pkgdata и pkginclude помещаются в цель install-data. Переменные со стандартными префиксами bin, sbin, libexec, sysconf, localstate, lib, pkglib устанавливаются целью install-exec. Например, файлы data_DATA устанавливаются целью install-data, а bin_PROGRAMS - install-exec. Любая переменная с пользовательским префиксом, включающим exec (например, myexecbin_PROGRAMS) будет отнесена к install-exec, а все прочие - к install-data.

12.3. Расширение установки

Можно расширить механизм, задав правило install-exec-local или install-data-local, которое будет выполняться по команде make install. Эти правила могут делать почти все и нужна осторожность.

Automake поддерживает также две «ловушки» (hook) - install-exec-hook и install-data-hook, которые работают после всех остальных правил установки соответствующего типа (exec или data). Например, можно внести изменения после установки с помощью таких ловушек (23.1. Расширение правил Automake).

12.4. Двухэтапная установка

Automake поддерживает переменную DESTDIR для всех правил установки, которая применяется при выполнении команды make install для перемещения объектов в промежуточную область (staging). Для каждого объекта и пути используется префикс, заданный значением DESTDIR, перед копированием в место установки. Например,

```
mkdir /tmp/staging &&
make DESTDIR=/tmp/staging install
```

Команда mkdir позволяет избежать проблем с безопасностью, когда злоумышленник создаёт символьную ссылку из /tmp/staging на область жертвы. Программа make затем устанавливает объекты в каталог /tmp/staging. При установке /gnu/bin/foo и /gnu/share/aclocal/foo.m4 показанная выше команда будет создавать /tmp/staging/gnu/bin/foo и /tmp/staging/gnu/share/aclocal/foo.m4. Это свойство обычно применяется для сборки установочных образов и пакетов (2.2.10. Сборка двоичных пакетов с использованием DESTDIR).

Поддержка DESTDIR реализована путём прямого включения в правила установки. Если Makefile.am использует правило локальной установки (например, install-exec-local) или установочную ловушку, нужно включать в них DESTDIR.

12.5. Правила установки для пользователя

Automake также создаёт правила для целей uninstall, installdirs и install-strip, а также поддерживает uninstall-local и uninstall-hook. Нет отдельных деинсталляций для exec и data, поскольку эти функции не имеют дополнительной функциональности. Отметим, что цель uninstall не является заменой инструментов для работы с пакетами.

13. Очистка

GNU Makefile Standards задаёт множество разных правил очистки.

Обычно файлы для очистки Automake выбирает автоматически. При этом используются переменные, которые можно задать для управления очисткой, - MOSTLYCLEANFILES, CLEANFILES, DISTCLEANFILES, MAINTAINERCLEANFILES.

Если очистка включает не только удаление жёстко заданного набора файлов, можно задать дополнительные правила со своими командами. Это делается путём задания правил для цели mostlyclean-local, clean-local, distclean-local или maintainer-clean-local (23.1. Расширение правил Automake). Распространенным случаем является удаление каталога, например, созданного тестами.

```
clean-local:
        -rm -rf testSubDir
```

Поскольку make принимает лишь один набор правил для данной цели, можно расширить создание правил за счёт указания отдельной цели в качестве зависимости.

```
clean-local: clean-local-check
.PHONY: clean-local-check
clean-local-check:
        -rm -rf testSubDir
```

Поскольку стандарты GNU не всегда чётко указывают, какие файлы следует удалять по какому правилу, здесь приведены некоторые рекомендации, которые изначально сформулировал Франсуа Пинар (François Pinard).

- Если команда make создала что-то перестраиваемое (например, файл .o), правилу mostlyclean следует удалять такой объект.
- В остальных случаях созданное командой make следует удалять цели clean.
- Если команда configure создала объект, цели distclean следует удалять его.

- Если сопровождающий создал объект (например, файл .info), цели maintainer-clean следует удалять его. Однако этой цели не следует удалять ничего, что нужно для выполнения команд ./configure && make.

Рекомендуется следовать этим правилам в файлах Makefile.am.

14. Распространение

14.1. Основы дистрибутивов

Правило dist в созданном файле Makefile.in можно использовать для создания сжатого (gzip) архива tar и других архивов. Имя файла выбирается автоматически на основе переменных PACKAGE и VERSION, заданных вызовом AC_INIT или (устаревшего) макроса AM_INIT_AUTOMAKE с двумя аргументами (6.4.1. Макросы общего пользования). Точнее говоря, архив называется ${PACKAGE}-${VERSION}.tar.gz. Для управления работой gzip можно использовать переменную make GZIP_ENV, в которой по умолчанию установлено значение --best.

В большинстве случаев файлы для распространения автоматически выбирает Automake. Все файлы с исходным кодом, а также Makefile.am и Makefile.in автоматически включаются в архив. Automake также включает список часто используемых файлов, включаемых при их обнаружении в текущем каталоге (реально или как цель в правиле Makefile.am). Этот список выводится по команде automake --help. Отметим, что некоторые файлы из этого списка на деле распространяются лишь при выполнении некоторых условий (например, файлы config.h.top и config.h.bot распространяются лишь при использовании AC_CONFIG_HEADERS([config.h]) в configure.ac). Файлы, читаемые сценарием configure (исходные файлы, соответствующие файлам, указанным в разных макросах Autoconf, таких как AC_CONFIG_FILES и его «братья»), распространяются автоматически. Файлы, включённые в Makefile.am (оператор include) или configure.ac (m4_include), а также вспомогательные сценарии, установленные командой automake --add-missing, также распространяются.

Тем не менее, иногда остаются файлы, которые нужно распространить, но они не включаются автоматически. Эти файлы следует указывать в переменной EXTRA_DIST, куда можно включать файлы из подкаталогов. Можно также указать в переменной каталог и он будет целиком рекурсивно скопирован в дистрибутив. Отметим, что при этом копируются все файлы, включая, например, каталоги Subversion (.svn) а также версии CVS/RCS управляющих файлов, поэтому такую возможность следует использовать с осторожностью. Однако можно воспользоваться свойством dist-hook для смягчения проблемы (14.3. «Ловушка» dist).

При указании SUBDIRS программа Automake будет рекурсивно включать подкаталоги в дистрибутив. Если переменная задаётся условно (20. Конструкции с условием), Automake обычно будет включать все каталоги, которые могут появиться в SUBDIRS. Если нужно задать включение некоторых каталогов по условию, можно задать переменную DIST_SUBDIRS с точным списком каталогов для включения в дистрибутив (7.2. Условные подкаталоги).

14.2. Тонкая настройка распространения

Иногда нужна тонкая настройка исключаемых из дистрибутива файлов, например, некоторые создаваемые автоматически файлы не нужно распространять. Для таких случаев Automake предоставляет префиксы тонкой настройки dist и nodist. Любая первичная переменная или _SOURCES может указываться с префиксом dist_ для включения в дистрибутив и nodist_ - для исключения. Ниже показано, как включить данные и исключить один файл.

```
dist_data_DATA = distribute-this
bin_PROGRAMS = foo
nodist_foo_SOURCES = do-not-distribute.c
```

14.3. «Ловушка» dist

Иногда полезна возможность изменить дистрибутив перед его упаковкой. При наличии правила dist-hook оно выполняется после заполнения каталога дистрибутива, но до создания архива. Одним из вариантов применения этого правила является удаление ненужных файлов с рекурсией в каталоге, заданном переменной EXTRA_DIST.

```
EXTRA_DIST = doc
dist-hook:
        rm -rf `find $(distdir)/doc -type d -name .svn`
```

Отметим, что заданию dist-hook не следует считать, что обычные файлы в дистрибутиве открыты для записи. Это может не выполняться при создании пакетов из дерева кода, доступного лишь для чтения, или при выполнении команды make distcheck. По тем же причинам заданию не следует предполагать, что подкаталоги, включённые в каталог дистрибутива в результате их включения в EXTRA_DIST, открыты для записи. Если задание dist-hook хочет изменить содержимое имеющегося файла (или каталога EXTRA_DIST) в каталоге дистрибутива, нужно сначала явно разрешить запись в него, как показано ниже.

```
EXTRA_DIST = README doc
dist-hook:
        chmod u+w $(distdir)/README $(distdir)/doc
        echo "Distribution date: `date`" >> README
        rm -f $(distdir)/doc/HACKING
```

Для создания правил dist-hook пригодятся переменные $(distdir) и $(top_distdir). Переменная $(distdir) указывает каталог, куда правило dist будет копировать файлы из текущего каталога перед созданием архива. При работе из каталога верхнего уровня distdir = $(PACKAGE)-$(VERSION), а при работе из foo/ в корневом каталоге distdir = ../$(PACKAGE)-$(VERSION)/foo. Переменная $(distdir) может указывать абсолютный или относительный путь.

Переменная $(top_distdir) всегда указывает на корневой каталог дистрибутива. Для каталога верхнего уровня она совпадает с $(distdir), а в подкаталоге foo/ top_distdir = ../$(PACKAGE)-$(VERSION)'. Переменная $(top_distdir) может указывать абсолютный или относительный путь.

Отметим, что для пакетов, вложенных с использованием AC_CONFIG_SUBDIRS (7.4. Вложенные пакеты), переменные $(distdir) и $(top_distdir) указываются относительно пакета, где используется команда make dist.

14.4. Проверка дистрибутива

Automake создаёт правило distcheck, которое может помочь в обеспечении корректной работы создаваемого дистрибутива. Если немного упростить, можно сказать, что сначала это правило создаёт дистрибутив, затем, работая с ним, выполняет перечисленные ниже действия:

- попытка сборки VPATH (2.2.6. Параллельная сборка (VPATH)) с srcdir и всем содержимым в режиме read-only;
- запуск тестов (make check) на свежей сборке;
- установка пакета во временный каталог (make install) и запуск тестов в нем (make installcheck);
- проверка корректности удаления (make uninstall) и очистки (make distclean) пакета;
- создание ещё одного архива для проверки самодостаточности дистрибутива.

Все эти действия выполняются во временном каталоге. Отметим, что точная структура такого каталога (расположение доступных лишь для чтения источников, именование временных каталогов сборки и установки, глубина вложенности и т. п.) следует считать деталями реализации, которые могут время от времени меняться.

DISTCHECK_CONFIGURE_FLAGS

Сборка пакета включает запуск сценария ./configure. Если нужно передать сценарию дополнительные флаги, следует задать их в переменной AM_DISTCHECK_CONFIGURE_FLAGS в файле Makefile.am на верхнем уровне. Пользователь может добавить или переопределить эти флаги указанием переменной DISTCHECK_CONFIGURE_FLAGS в строке команды make. Следует отметить, что команде make distcheck требуется полный контроль над опциями сценария configure --srcdir и --prefix, поэтому их нельзя переопределить в переменной AM_DISTCHECK_CONFIGURE_FLAGS или DISTCHECK_CONFIGURE_FLAGS.

Отметим также, что разработчикам рекомендуется делать свой код собираемым без специальных опций configure, т. е., обычно от пользователя не должно требоваться указание переменной AM_DISTCHECK_CONFIGURE_FLAGS. Однако в некоторых случаях использование этой переменной оправдано. Пример этого представлен в GNU m4, где по умолчанию отключено экспериментальное и редко используемое свойство changeword. Для использования этой функции перед использованием make distcheck запустить configure с параметром —with-changeword для корректной компиляции кода changeword. GNU m4 также использует переменную AM_DISTCHECK_CONFIGURE_FLAGS для нагрузочных тестов использования --program-prefix=g, поскольку в какой-то момент в системе сборки m4 возникла ошибка, в результате которой команда make installcheck ошибочно предполагала возможность проверки m4 вслепую, а не свежеустановленную программу gm4.

distcheck-hook

Если правило distcheck-hook определено в файле Makefile.am верхнего уровня, оно будет вызываться целью distcheck после распаковки нового дистрибутива, но до настройки конфигурации и сборки пакета. Правило distcheck-hook может делать почти все, но нужна некоторая осторожность. Обычно правило применяется для проверки потенциальных ошибок дистрибутива, не замеченных стандартным механизмом. Отметим, что distcheck-hook, как и AM_DISTCHECK_CONFIGURE_FLAGS, DISTCHECK_CONFIGURE_FLAGS не используются в Makefile.am субпакетов, но флаги из AM_DISTCHECK_CONFIGURE_FLAGS и DISTCHECK_CONFIGURE_FLAGS передаются сценариям configure в субпакетах.

distcleancheck

В части возможных ошибок дистрибутива distcheck гарантирует, что правило distclean на деле удаляет все собранные файлы. Это выполняется командой make distcleancheck в конце сборки VPATH. По умолчанию distcleancheck запускает distclean, а затем проверяет очистку дерева сборки запуском $(distcleancheck_listfiles). Обычно эта проверка находит созданные файлы, которые не были включены в DISTCLEANFILES (13. Очистка).

Для большинства пакетов distcleancheck работает корректно, в остальных случаях нужно переопределить правило distcleancheck или переменную $(distcleancheck_listfiles). Например, для полного запрета distcleancheck можно добавить в Makefile.am верхнего уровня правило, показанное ниже.

```
distcleancheck:
        @:
```

Если цель distcleancheck должна игнорировать собранные файлы, которые не были удалены по причине включения в дистрибутив, следует добавить приведённое ниже определение.

```
distcleancheck_listfiles = \
  find . -type f -exec sh -c 'test -f $(srcdir)/$$1 || echo $$1' \
       sh '{}' ';'
```

Это определение не применяется по умолчанию, поскольку требование Makefile повторно собирать некоторые файлы, собранные пользователем, обычно является ошибкой (27.5. Ошибки distclean).

distuninstallcheck

Цель distcheck также проверяет корректность работы правила uninstall для обычной и DESTDIR-сборки. Это выполняется вызовом make uninstall и последующей проверки дерева установки на предмет оставшихся файлов. По умолчанию проверка выполняется правилом distuninstallcheck, а список файлов дерева установки создаётся командой $(distuninstallcheck_listfiles). Правила и переменная позволяют изменить поведение distcheck. Например, для полного отключения проверки можно задать

```
distuninstallcheck:
        @:
```

14.5. Типы распространения

Automake создаёт правила для создания архивов дистрибутива в перечисленных ниже форматах.

dist-gzip

Создаёт архив gzip из файла tar. Это единственный формат, включенный по умолчанию.

dist-bzip2
Создаёт архив bzip2 из файла tar. Формат bzip2 зачастую снижает размер архива по сравнению с gzip. По умолчанию это правило создаёт архив bzip2 с опцией сжатия -9, для изменения можно установить переменную среды BZIP2, например, make dist-bzip2 BZIP2=-7.

dist-lzip
Создаёт архив lzip из файла tar. Формат lzip зачастую снижает размер архива по сравнению bzip2.

dist-xz
Создаёт архив xz из файла tar. Формат xz зачастую снижает размер архива по сравнению bzip2. По умолчанию это правило создаёт архив xz с опцией сжатия -e, для изменения можно установить переменную среды XZ_OPT, например, make dist-xz XZ_OPT=-ve.

dist-zip
Создаёт архив zip.

dist-tarZ
Создаёт архив tar, сжатый устаревшей программой compress. Опция устарела и будет удалена из Automake 2.0.

dist-shar
Создаёт архив shar. Формат устарел и применять его не рекомендуется, опция будет удалена из Automake 2.0.

Правило dist (и его прежний синоним dist-all) создаёт архивы во всех включённых форматах (17.2. Список опций Automake). По умолчанию используется лишь формат dist-gzip.

15. Поддержка тестов

Automake может генерировать код для обработки двух типов тестов, один из которых основан на интеграции со схемой dejagnu, другой (используется чаще) - на использовании базовых тестовых сценариев с активацией их заданием специальной переменной TESTS. Второй вариант поддерживает разные варианты сложности и настройки, позволяя выполнять тесты одновременно, используя такие протоколы тестирования как TAP, а также задание пользовательских драйверов и исполнителей тестов. Тестирование запускается командой make check.

15.1. Общие вопросы тестирования

Тестирование предназначено для проверки поведения программы или системы по сравнению с ожидаемым (например, известные данные на входе дают ожидаемый вывод, ошибки корректно обрабатываются и выводятся, прежние ошибки не повторяются). Минимальную единицу тестирования называют тестом или контрольным примером. Определение и границы тестового примера существенно зависят от принятой парадигмы тестирования и/или используемой схемы. Набор контрольных примеров для данной программы или системы составляет комплект тестов.

Комплект тестов является программой или программной компонентой для выполнения (всех или части) заданных контрольных примеров, анализа результатов и создания отчёта или реестра. Детали процесса и взаимодействие разработчика или пользователя с ним могут существенно различаться и не определяются здесь.

Тест считается успешным (pass), когда поведение соответствует ожидаемому и неудачным (fail) в противном случае. Иногда тесты могут основываться на непереносимых инструментах или предварительных условиях, а также могут просто не иметь смысла в той или иной системе (например, проверка связанных с Windows свойств в GNU/Linux). В таких случаях тесты пропускаются (skip) без запуска, а результат не учитывается. Пропуски обычно указываются в отчёте, чтобы пользователь знал о реально выполненных проверках.

Нередко, особенно на начальных этапах разработки, отказы части тестов происходят по известным причинам и разработчик пока не хочет устранять их. В таких случаях лучше указать такие отказы как ожидаемые (xfail). Если такой тест проходит, в отчёте он отмечается как неожиданной успешный (xpass).

Многие среды и схемы тестирования различают отказы тестов и серьёзные ошибки. Отказом теста считается несоответствие поведения ожидаемому, а серьёзной ошибкой - сбой организации тестового сценария или иные неожиданные или нежелательные ситуации (например, ошибка сегментации в тестируемой программе).

15.2. Простые тесты

15.2.1. Тесты на основе сценариев

При задании специальной переменной TESTS её значение считается списком программ или сценариев, используемых для тестирования. При соответствующих обстоятельствах TESTS может также содержать список файлов данных для передачи одному или нескольким тестовым сценариям, заданных другими способами (так называемые «компиляторы журналов», 15.2.3. Параллельные тесты). Тестовые сценарии могут выполняться последовательно или одновременно, по умолчанию применяется параллельное тестирование, обеспечиваемое реализацией make (если оно поддерживается).

По умолчанию в качестве результатов набора тестов рассматривается лишь статус завершения тестовых сценариев. Однако Automake поддерживает более сложные протоколы тестирования, как стандартные (15.4. Использование протокола TAP), так и пользовательские (15.3. Драйверы тестов). Отметим, что эти протоколы невозможно включить для последовательного тестирования. Далее рассматриваются в основном тесты без протокола, поскольку протоколам тестирования посвящён отдельный параграф 15.3. Драйверы тестов. Когда протокол тестирования не применяется, статус 0 на выходе сценария означает успешный результат, 77 - пропуск теста, 99 - серьёзную ошибку, а все остальные значения - отказ.

Можно задать переменную XFAIL_TESTS для списка тестов (обычно подмножество TESTS), в которых заранее предполагается отказ. Это будет инвертировать результаты указанных тестов (при условии, что пропуски и серьёзные ошибки не будут затронуты). Можно также задать трактовку серьёзных ошибок как обычных отказов, задав непустое значение в переменной DISABLE_HARD_ERRORS. Однако следует отметить, что для тестов на основе более сложных протоколов точное влияние переменных XFAIL_TESTS и DISABLE_HARD_ERRORS может меняться и даже возможно их полное игнорирование (например, в тестах с использованием TAP нет возможности отключить серьёзные ошибки и переменная DISABLE_HARD_ERRORS просто не работает).

Результат каждого теста, запускаемого сценариями из TESTS, будет выводиться на стандартное устройство stdout вместе с именем теста. Для протоколов тестирования, которые поддерживают множество тестов в сценарии

(например, TAP) в дополнение к имени тестового сценария ожидается номер, идентификатор и краткое описание отдельных тестов. Возможные результаты (15.1. Общие вопросы тестирования) могут иметь значение PASS, FAIL, SKIP, XFAIL, XPASS, ERROR. Ниже приведён пример вывода набора тестов, использующего обычные сценарии и TAP.

```
PASS: foo.sh
PASS: zardoz.tap 1 - Daemon started
PASS: zardoz.tap 2 - Daemon responding
SKIP: zardoz.tap 3 - Daemon uses /proc # SKIP /proc is not mounted
PASS: zardoz.tap 4 - Daemon stopped
SKIP: bar.sh
PASS: mu.tap 1
XFAIL: mu.tap 2 # TODO frobnication not yet implemented
```

Сводка по набору тестов (по крайней мере число запущенных, пройденных и отказавших тестов) выводится по завершении работы. Если стандартное устройство вывода подключено к подходящему терминалу, результаты теста и сводка выводятся с цветовым выделением. Разработчик и пользователь могут отключить цветовой вывод, передав в команде make переменную AM_COLOR_TESTS=no. Пользователь может задать цветовой вывод даже без подключения терминала, задав AM_COLOR_TESTS=always. Следует также отметить, что некоторые реализации make при работе в параллельном режиме могут различаться по семантике, что может препятствовать автоматическому определению возможностей подключённого терминала. В этом случае пользователю придётся задать переменную AM_COLOR_TESTS=always для получения цветового вывода.

Тестовые программы, которым нужны файлы данных, должны искать их в srcdir (переменная make и переменная среды, доступной для тестов), чтобы тесты работали даже при сборке в отдельном каталоге и, в частности, для правила distcheck (14.4. Проверка дистрибутива).

Переменные AM_TESTS_ENVIRONMENT и TESTS_ENVIRONMENT можно использовать для запуска кода инициализации и настройки окружения для тестовых сценариев. Первая переменная предназначена для разработчиков и может быть задана в файле Makefile.am. Другая переменная предназначена для пользователя, который может с её помощью расширить или переопределить первую переменную, однако при этом для переносимости требуется, чтобы непустая переменная AM_TESTS_ENVIRONMENT завершалась символом двоеточия.

Переменную AM_TESTS_FD_REDIRECT можно использовать для задания перенаправлений файловых дескрипторов в тестовых сценариях. Можно подумать, что для этого подходит переменная AM_TESTS_ENVIRONMENT, но опыт показывает, что это препятствует переносимости. Основным препятствием являются оболочки Korn, которые обычно устанавливают флаг close-on-exec для файловых дескрипторов, открытых внутренней функцией exec, что делает конструкции вида AM_TESTS_ENVIRONMENT = exec 9>&2; неэффективными. Проблема возникает также в некоторых оболочках Bourne, например, HP-UX /bin/sh.

```
AM_TESTS_ENVIRONMENT = \
## Some environment initializations are kept in a separate shell
## file 'tests-env.sh', which can make it easier to also run tests
## from the command line.
  . $(srcdir)/tests-env.sh; \
## On Solaris, prefer more POSIX-compliant versions of the standard
## tools by default.
  if test -d /usr/xpg4/bin; then \
    PATH=/usr/xpg4/bin:$$PATH; export PATH; \
  fi;
## With this, the test scripts will be able to print diagnostic
## messages to the original standard error stream, even if the test
## driver redirects the stderr of the test scripts to a log file
## before executing them.
AM_TESTS_FD_REDIRECT = 9>&2
```

Отметим, что переменная AM_TESTS_ENVIRONMENT в силу исторических причин и особенностей реализации не поддерживается для последовательных тестов (15.2.2. Последовательные тесты (устарели)).

Automake гарантирует создание каждого файла, указанного в переменной TESTS, до его запуска. В переменной можно указать исходные файлы и производные программы (или сценарии), созданное правило будет видно в каталогах srcdir и ‘.’. Например, может возникнуть потребность выполнения в качестве теста программы C. Для этого нужно включить имя теста в TESTS и check_PROGRAMS, а затем указать его как любую другую программу. Программы, перечисленные в check_PROGRAMS (а также в check_LIBRARIES, check_LTLIBRARIES...) собираются только по команде make check, а не make all. Здесь следует указывать любые программы, требуемые для теста, которые не нужно собирать по команде make all. Отметим, что check_PROGRAMS не включается автоматически в TESTS, поскольку в check_PROGRAMS обычно задаются используемые тестами программы, а не сами тесты. Можно задать TESTS = $(check_PROGRAMS), если все программы являются контрольными примерами.

15.2.2. Последовательные тесты (устарели)

Отметим сначала, что сегодня настоятельно не рекомендуется применять такие тесты, используя взамен параллельные, которые описаны ниже. Тем не менее возникают ситуации, когда преимущества параллельных тестов не важны и одновременное тестирование может даже вызывать проблемы. В таких случаях можно использовать простые и надёжные последовательные тесты. Этот режим включается опцией Automake serial-tests и тесты просто запускаются один за другим с записью результатов, если это нужно.

В силу исторических причин и особенностей реализации переменная AM_TESTS_ENVIRONMENT не поддерживается в этом режиме (игнорируется) и работает лишь переменная TESTS_ENVIRONMENT, которая считается зарезервированной для разработчиков. Это сделано для того, чтобы при последовательном тестировании можно было определить TESTS_ENVIRONMENT при вызове интерпретатора, через который будут выполняться тесты. Например, для запуска тестов с помощью Perl можно использовать

```
TESTS_ENVIRONMENT = $(PERL) -Mstrict -w
TESTS = foo.pl bar.pl baz.pl
```

Важно отметить, что описанное здесь применение TESTS_ENVIRONMENT непригодно для параллельных тестов, где обеспечивается более изящное решение, дающее такой же эффект и дополнительное преимущество за счёт освобождения переменной TESTS_ENVIRONMENT для пользователя. Менее серьёзным ограничением последовательного тестирования является неразличимость простых отказов и серьёзных ошибок. Это обусловлено историческими причинами и может быть исправлено в будущих версиях Automake.

15.2.3. Параллельные тесты

По умолчанию Automake генерирует параллельные (одновременные) тесты. Это включает автоматический сбор результатов тестовых сценариев в журнал (.log), одновременное выполнение тестов с опцией make -j, заданием зависимостей между тестами, возврат к незавершённым тестам и фиксацию серьёзных ошибок. Параллельное тестирование выполняется путём создания набора правил make для запуска тестовых сценариев, указанных в переменной TESTS и сохранение вывода каждого сценария в журнальном файле .log, а также результатов (и других «метаданных», 15.3.3. API для сторонних драйверов) в файлах .trs. Файл .log содержит весь вывод теста, выдаваемый на устройства stdoutput и stderr. Файл .trs содержит результаты контрольных примеров, выполняемых сценарием. Для параллельных тестов создаётся также общий журнал, указанный переменной TEST_SUITE_LOG (по умолчанию test-suite.log) и имеющий расширение .log. Этот файл зависит от всех файлов .log и .trs, указанных в переменной TESTS.

Как и для описанных выше последовательных тестов, по умолчанию выводится 1 строка результата каждого выполненного теста и краткая сводка по завершении тестирования. Однако вывод stdout и stderr перенаправляется ф журнальный файл на уровне теста, поэтому вывод параллельных тестов не перемешивается. Вывод отказавших тестов собирается в общий файл test-suite.log. Если установлена переменная VERBOSE, содержимое этого файла выводится после сводки.

Каждая пара файлов .log и .trs создаётся по завершении соответствующего теста. Набор log-файлов указывается в доступной лишь для чтения переменной TEST_LOGS, которая по умолчанию совпадает с TESTS. Расширения исполняемых файлов, если они имеется (8.20. Поддержка расширений исполняемых файлов), а также любые суффиксы, указанные в TEST_EXTENSIONS, удаляются и добавляется суффикс .log. Результат становится неопределённым, если имя файла завершается объединением нескольких суффиксов. По умолчанию TEST_EXTENSIONS имеет значение .test, которое может переопределить пользователь. В этом случае расширение должно начинаться с точки, за которой следует буква и может следовать любое число букв и цифр. Например, расширения .sh, .T и .t1 являются корректными, а .x-y, .6c и .t.1 недопустимы. Важно отметить, что имеющиеся ограничения (которые вряд ли будут отменены) ведут к тому, что подстановки configure в определение TESTS могут работать лишь при преобразовании в список тестов, имеющих суффиксы из переменной TEST_EXTENSIONS.

Для тестов, соответствующих расширению .EXT, указанному в TEST_EXTENSIONS, можно предоставить пользовательский «исполнитель тестов», используя переменную EXT_LOG_COMPILER (расширение указывается заглавными буквами), передавая опции в переменной AM_EXT_LOG_FLAGS и разрешая пользователю передавать опции в EXT_LOG_FLAGS. В результате все тесты с указанным расширением будут вызваны указанным «исполнителем». Для тестов без зарегистрированного расширения можно применять переменные LOG_COMPILER, AM_LOG_FLAGS и LOG_FLAGS. Например,

```
TESTS = foo.pl bar.py baz
TEST_EXTENSIONS = .pl .py
PL_LOG_COMPILER = $(PERL)
AM_PL_LOG_FLAGS = -w
PY_LOG_COMPILER = $(PYTHON)
AM_PY_LOG_FLAGS = -v
LOG_COMPILER = ./wrapper-script
AM_LOG_FLAGS = -d
```

будет вызывать $(PERL) -w foo.pl, $(PYTHON) -v bar.py и ./wrapper-script -d baz для создания файлов foo.log, bar.log и baz.log, соответственно. Файлы foo.trs, bar.trs и baz.trs создаются автоматически, как побочный результат.

Важно отметить, что в отличие от последовательных тестов, переменные AM_TESTS_ENVIRONMENT и TESTS_ENVIRONMENT не могут применяться для определения пользовательского «исполнителя» и взамен должны применяться переменные LOG_COMPILER и LOG_FLAGS (или их аналоги для расширений).

```
## This is WRONG!
AM_TESTS_ENVIRONMENT = PERL5LIB='$(srcdir)/lib' $(PERL) -Mstrict -w

## Do this instead.
AM_TESTS_ENVIRONMENT = PERL5LIB='$(srcdir)/lib'; export PERL5LIB;
LOG_COMPILER = $(PERL)
AM_LOG_FLAGS = -Mstrict -w
```

По умолчанию запускаются все тесты набора, но имеется несколько способов ограничить номенклатуру тестов.

- Переменная TESTS позволяет задать подмножество выполняемых тестов, как показано ниже.

  ```
  env TESTS="foo.test bar.test" make -e check
  ```

  Однако эта команда будет безусловно переписывать файл test-suite.log, уничтожая результаты предыдущих тестов, что может оказаться нежелательным для пакетов с продолжительным тестированием. К счастью эту проблему легко решить переопределением переменной TEST_SUITE_LOG по время теста. Например,

  ```
  env TEST_SUITE_LOG=partial.log TESTS="..." make -e check
  ```

  обеспечит запись результатов отдельных тестов в файл partial.log, не трогая test-suite.log.

- Можно установить переменную TEST_LOGS. По умолчанию эта переменная определяется при работе make из значения TESTS, как описано выше. Можно задать, например

  ```
  set x subset*.log; shift
  env TEST_LOGS="foo.log $*" make -e check
  ```

  По умолчанию набор тестов удаляет все старые файлы .log и .trs отдельных тестов при запуске для регенерации этих файлов с новыми результатами. Переменная RECHECK_LOGS указывает набор удаляемых файлов .log (и косвенно, .trs). По умолчанию RECHECK_LOGS совпадает с TEST_LOGS, что означает необходимость перепроверки всех файлов. Переопределяя эту переменную, можно указать тесты, которые нужно пересмотреть. Например, можно перезапустить лишь те тесты, которые устарели (т. е. старее обязательных тестовых файлов), установив в переменной пустое значение, как показано ниже.

  ```
  env RECHECK_LOGS= make -e check
  ```

- Можно повторить все тесты, которые закончились отказом или неожиданно прошли, введя команду make recheck из каталога тестов. Эта условная цель должным образом устанавливает RECHECK_LOGS перед запуском.

Для гарантии упорядоченного тестирования даже по команде make -jN, нужно указать зависимости между файлами .log как делается с обычными зависимостями make. Например зависимость foo-execute.test от foo-compile.test можно указать в виде

```
TESTS = foo-compile.test foo-execute.test
foo-execute.log: foo-compile.log
```

Заданный порядок гарантирует результаты требуемых тестов, поэтому foo-execute.test будет запущен даже при отказе или пропуске foo-compile.test. Кроме того, следует отметить, что такое задание зависимостей в настоящее время работает лишь для тестов, имена которых имеют суффиксы, указанные в TEST_EXTENSIONS.

Тесты без указанных зависимостей могут выполняться одновременно по команде make -jN, поэтому следует убедиться в том, что они готовы к параллельному выполнению.

Комбинация отложенного выполнения и корректных зависимостей между тестами и их источниками может применяться для эффективного тестирования модулей в процессе разработки. Для дальнейшего ускорения цикла разработки и тестирования может быть полезно задать компилируемые программы в переменной EXTRA_PROGRAMS вместо check_PROGRAMS, поскольку первый вариант допускает чередование компиляции с выполнением тестов (однако EXTRA_PROGRAMS не очищается автоматически, см. 3.3. Схема именования).

Переменные TESTS и XFAIL_TESTS могут включать условные части, а также подстановки configure. В последнем случае имеются некоторые ограничения - подставленные имена тестов должны иметь непустой суффикс (например, .test), чтобы можно было применить одно из правил выведения, созданных automake. Для литеральных имён. тестов automake может создавать правила на уровне цели, чтобы преодолеть это ограничение.

Отметим, что в настоящее время невозможно использовать $(srcdir)/ или $(top_srcdir)/ в переменной TESTS. Это техническое ограничение нужно для предотвращения генерации журналов тестирования в дереве источников и его следствием является невозможность задать распределенные тесты с самогенерацией на основе явных правил переносимым между реализациями make способом (конфликт с семантикой FreeBSD и OpenBSD). При сомнениях можно воспользоваться GNU make или обойти эту проблему с помощью правил выведения при генерации тестов.

15.3. Драйверы тестов

15.3.1. Обзор поддержки сторонних драйверов тестов

В Automake, начиная с версии 1.12, параллельные тесты позволяют авторам применять сторонние драйверы тестов, если стандартные драйверы не подходят или не поддерживают нужный протокол тестирования.

Ожидается, что сторонний драйвер будет корректно выполнять переданную уму тестовую программу (включая переданные для неё аргументы, если они имеются) для анализа её работы и результата, создавать файлы .log и .trs, связанные с запуском теста, и выводить результат теста на консоль. Ответственность за выполнение и корректность перечисленных функций ложится на автора драйвера, Automake здесь ничем не поможет. Тем не менее, код Automake для создания отчётов поддерживает драйверы тестов с множеством результатов одного сценария, если эти результаты регистрируются корректно (15.3.3.2. Создание журнала и запись результатов).

Детали определения и анализа результатов отдельного тестового сценария оставлены драйверам. Некоторые драйверы могут рассматривать лишь статус завершения тестового сценария (это делает, например, принятый по умолчанию драйвер параллельных тестов, описанный выше). Другие могут реализовать более сложные и развитые протоколы тестирования и интерпретировать вывод каждого тестового сценария (примерами служат TAP и SubUnit).

Важно отметить, что при использовании сторонних драйверов большая часть описанной выше инфраструктуры параллельного тестирования продолжает работать. Это включает:

- список тестовых сценариев в TESTS с возможностью переопределения в TESTS или TEST_LOGS;
- одновременное выполнение тестов по команде make -j;
- поддержка файлов .log и .trs для каждого файла и создание на их основе отчёта о тесте (.log);
- перепроверка целей (RECHECK_LOGS) и отложенный повтор тестов;
- зависимости между тестами;
- поддержка переменных check_* (check_PROGRAMS, check_LIBRARIES, ...);
- использование переменной среды VERBOSE для расширенного вывода информации об отказах тестов;
- задание и исполнение TESTS_ENVIRONMENT, AM_TESTS_ENVIRONMENT, AM_TESTS_FD_REDIRECT;
- задание базовых и связанных с расширениями переменных LOG_COMPILER и LOG_FLAGS.

Точная семантика цветового вывода, XFAIL_TESTS и серьёзных ошибок определяется драйверами тестов.

15.3.2. Объявление сторонних драйверов тестов

Сторонние драйверы объявляются в переменных make LOG_DRIVER или EXT_LOG_DRIVER (расширение EXT должно быть TEST_EXTENSIONS), которые должны быть заданы для программ и сценариев, использующих драйверы для запуска тестов, записи результатов и создания отчётов о тестах с указанным расширением (или без расширения в случае LOG_DRIVER). В одном файле Makefile.am можно объявить разные драйверы тестов. Отметим также, что переменные LOG_DRIVER не заменяют LOG_COMPILER и два набора тестов могут существовать совместно.

Переменные AM_LOG_DRIVER_FLAGS (для разработчиков) и LOG_DRIVER_FLAGS (для пользователей) могут служить для задания флагов, передаваемых при каждом вызове LOG_DRIVER. При этом пользовательские флаги будут иметь более высокий приоритет. Аналогично, для каждого расширения EXT из TEST_EXTENSIONS флаги, указанные в AM_EXT_LOG_DRIVER_FLAGS и EXT_LOG_DRIVER_FLAGS, передаются при вызовах EXT_LOG_DRIVER.

15.3.3. API для сторонних драйверов

Описанные здесь API являются в значительной степени экспериментальными и могут существенно измениться в будущем. Основной характеристикой API является поддержка разной архитектуры, семантики и деталей реализации возможных в среде параллельного тестирования.

15.3.3.1. Аргументы команд для сторонних драйверов

Сторонний драйвер может опираться на разные опции и аргументы команд, передаваемые ему автоматически созданными Automake тестовыми наборами. Однако обязательно понимать их (хотя интерпретация семантики может различаться в разных драйверах и даже не реализоваться некоторыми из них). Список опций приведён ниже.

--test-name=NAME
: Имя теста с префиксом VPATH (при наличии) удаляется. Имя может включать суффикс и каталог (например, sub/foo.test) и предназначено в основном для вывода отчёта на консоль (15.3.3.3. Вывод в процессе тестирования).

--log-file=PATH.log
: Файл .log, который драйвер должен создавать (15.2.3. Параллельные тесты). При включении каталога (например, sub/foo.log) набор тестов будет обеспечивать наличие такого каталога до вызова драйвера теста.

--trs-file=PATH.trs
: Файл .trs, который драйвер должен создавать (15.2.3. Параллельные тесты). При включении каталога (например, sub/foo.log) набор тестов будет обеспечивать наличие такого каталога до вызова драйвера теста.

--color-tests={yes|no}
: Управляет цветовым представлением консольного вывода (15.2.1. Тесты на основе сценариев).

--expect-failure={yes|no}
: Указывает, ожидается ли отказ тестируемой программы.

--enable-hard-errors={yes|no}
: Указывает, следует ли обрабатывать серьёзные ошибки в тестируемой программе как обычные отказы (по умолчанию должно быть yes). Трактовка серьёзных ошибок зависит от протокола тестирования и соглашений.

--
: Явное завершение списка опций.

Первый аргумент, не являющийся опцией, переданный драйверу теста, указывает программу для запуска, а последующие задают опции и аргументы, передаваемые этой программе.

Отметим, что семантика опций --color-tests, --expect-failure и —enable-hard-errors определяется драйвером. Тем не менее, следует обеспечивать поведение, похожее или совместимое с поведением принятого по умолчанию драйвера.

15.3.3.2. Создание журнала и запись результатов

Драйвер должен корректно генерировать файлы, заданные опциями --log-file и —trs-file (даже при отказе или аварийном завершении программы). Файлы .log в идеале должны включать весь вывод тестируемой программы, а также могут содержать дополнительную информацию, которая может помочь в отладке или отчёте об ошибках. В остальном формат файла остаётся свободным. Файлы .trs служат для регистрации некоторых метаданных с помощью полей reStructuredText. Предполагается использование этих метаданных разными способами в параллельных тестах, например, для подсчёта результатов и вывода отчёта о тесте при выполнении команды make recheck. Нераспознанные метаданные в .trs сейчас игнорируются, но это может измениться в будущем. Список метаданных приведён ниже.

:test-result:
: Драйвер должен использовать это поле для регистрации результатов каждого контрольного примера, запускаемого сценарием. В одном файле .trs может присутствовать несколько полей :test-result:. Это позволяет протоколам тестирования включать в один сценарий множество тестов. Распознаваемыми результатами тестов являются PASS, XFAIL, SKIP, FAIL, XPASS и ERROR. За этими результатами при указании с :test-result: может следовать текст с именем и/или кратким описанием соответствующего теста. Этот текст игнорируется при создании файла test-suite.log и генерации отчёта о тесте.

:recheck:
: Если это поле присутствует и содержит значение no, соответствующий тест не будет выполняться по команде make recheck. Наличие нескольких полей :recheck: в файле .trs может сделать поведение неопределённым.

:copy-in-global-log:
: Если это поле присутствует и содержит значение no, содержимое файла .log не будет копироваться в test-suite.log. Возможность отказа от копирования обусловлена тем, что несмотря на полезность этих данных при отладке и анализе ошибок, это может приводить к пустому расходу места в обычной ситуации (например, при успешном тестировании). Наличие нескольких полей :copy-in-global-log: в одном файле .trs может вести к неопределённости.

:test-global-result:
: Служит для объявления «глобального» результата набора тестов. Сейчас значение поля применяется лишь для отчёта в журнале $(TEST_SUITE_LOG) и разрешает достаточно свободную форму. Например, сценарий с 10 тестами, из которых 6 прошли успешно, а 4 были пропущены, может использовать в этом поле значения PASS/SKIP, а сценарий с 19 тестами и 1 отказом может указать значение ALMOST PASSED. Наличие нескольких полей :test-global-result: в одном файле .trs может вести к неопределённому поведению.

Предположим, например, что файл .trs содержит приведённые ниже строки.

```
:test-result: PASS server starts
:global-log-copy: no
:test-result: PASS HTTP/1.1 request
:test-result: FAIL HTTP/1.0 request
:recheck: yes
:test-result: SKIP HTTPS request (TLS library wasn't available)
:test-result: PASS server stops
```

Соответствующий сценарий будет повторен командой make check, добавит 5 результатов в сводку (3 успешных теста, 1 отказ и 1 пропуск), а содержимое соответствующего файла .log не будет копироваться в test-suite.log.

15.3.3.3. Вывод в процессе тестирования

Сторонние драйверы также заинтересованы в выводе на stdout информации информации о результатах теста, когда она становится доступной. В зависимости от применяемого протокола могут также выводиться причины отказов и пропусков, а также полезные данные диагностики (следует помнить, что строк на экране не так много и избыточный

вывод будет захламлять экран). Точный формат вывода остаётся за драйвером и тот может даже не выводить ничего, оставляя все тестовому сценарию (неразумно и упомянуто лишь для разъяснения предоставляемой гибкости).

Отметим, что согласованность вывода важна и к ней следует стремиться, не отходя далеко от вывода встроенных драйверов Automake. В частности, вывод тестового набора следует окрашивать, передавая драйверу опцию --color-tests. С другой стороны, при использовании широко распространённого протокола тестирования разумно обеспечивать согласованный с этим протоколом вывод.

15.4. Использование протокола TAP

15.4.1. Введение в TAP

TAP (Test Anything Protocol) обеспечивает простой текстовый интерфейс между тестирующими модулями или программами и набором тестов. Тесты (процедуры TAP) записывают результат в простом формате на устройство stdout, а сценарий теста (потребитель TAP) анализирует и интерпретирует эти результаты, представляет их пользователю и/или регистрирует для дальнейшего анализа. Детали могут зависеть от тестового набора. Тесты Automake выводят результаты на консоль (15.2.1. Тесты на основе сценариев) и используют файлы .trs (15.2.3. Параллельные тесты) для хранения результатов и связанных метаданных. Кроме того, тесты пытаются обеспечить совместимость с имеющимися и распространёнными утилитами (например, prove) хотя бы в простых случаях.

Протокол TAP начинался как тестовый набор для Perl, но сегодня стандартизован (в основном) и имеет независимые реализации в разных языках, включая C, C++, Perl, Python, PHP, Java. Полуофициальная спецификация протокола TAP доступна в документации Test::Harness::TAP. Очевидно, что наиболее подходящие примеры использования TAP следует искать в тестах perl и модулей perl. Есть и сторонние пакеты, применяющие тесты TAP, например, git.

15.4.2. Использование TAP с тестами Automake

Драйвер TAP из комплекта Automake требует от разработчика некоторых действий (возможно этого не потребуется в будущих версиях). Нужно вручную извлечь сценарий tap-driver.sh из дистрибутива Automake и скопировать его в дерево кода, а также использовать в Automake поддержку сторонних драйверов, чтобы тесты использовали сценарий tap-driver.sh и программу awk, найденную макросом AM_INIT_AUTOMAKE для запуска созданных TAP тестов. Помимо параметров, общих для всех драйверов тестов Automake (15.3.3.1. Аргументы команд для сторонних драйверов), tap-driver.sh поддерживает перечисленные ниже опции, имена которых выбраны с учётом совместимости с prove.

--ignore-exit
: Заставляет драйвер теста игнорировать статус завершения тестовых сценариев. По умолчанию драйвер будет сообщать об ошибке при возврате ненулевого кода. Опция влияет также на обработку кода, связанного с прерыванием теста по сигналу.

--comments
: Задаёт драйверу вывод диагностики TAP (строки, начинающиеся с #) в процессе работы теста. По умолчанию диагностика TAP копируется только в файл .log.

--no-comments
: Отменяет опцию --comments.

--merge
: Указывает драйверу объединять вывод stderr и stdout. Это нужно для отображения диагностики теста в корректном порядке относительно результатов, особенно при отладке, когда тестовый shell-сценарий запускается с трассировкой. Однако опция может приводить к путанице в тесте, если вывод stderr похож на результат теста.

--no-merge
: Отменяет опцию --merge.

--diagnostic-string=STRING
: Меняет строку указания диагностики TAP с принятого по умолчанию # на STRING. Это может быть полезно в основанных на TAP сценариях тестов с ограниченным контролем и подробным выводом (например, в результате получения вывода от вовлечённых в тест других инструментов), который может ошибочно считаться диагностикой TAP. Проблему можно решить путём переопределения строки, указывающей диагностику TAP и невозможной в выводе теста. Следует отметить отсутствие функции в «официальном» протоколе TAP.

Ниже приведён пример настройки и использования драйвера TAP.

```
% cat configure.ac
AC_INIT([GNU Try Tap], [1.0], [bug-automake@gnu.org])
AC_CONFIG_AUX_DIR([build-aux])
AM_INIT_AUTOMAKE([foreign -Wall -Werror])
AC_CONFIG_FILES([Makefile])
AC_REQUIRE_AUX_FILE([tap-driver.sh])
AC_OUTPUT

% cat Makefile.am
TEST_LOG_DRIVER = env AM_TAP_AWK='$(AWK)' $(SHELL) \
                  $(top_srcdir)/build-aux/tap-driver.sh
TESTS = foo.test bar.test baz.test
EXTRA_DIST = $(TESTS)

% cat foo.test
#!/bin/sh
echo 1..4 # Number of tests to be executed.
echo 'ok 1 - Swallows fly'
echo 'not ok 2 - Caterpillars fly # TODO metamorphosis in progress'
echo 'ok 3 - Pigs fly # SKIP not enough acid'
echo '# I just love word plays ...'
echo 'ok 4 - Flies fly too :-)'

% cat bar.test
#!/bin/sh
echo 1..3
echo 'not ok 1 - Bummer, this test has failed.'
echo 'ok 2 - This passed though.'
echo 'Bail out! Ennui kicking in, sorry...'
echo 'ok 3 - This will not be seen.'
```

```
% cat baz.test
#!/bin/sh
echo 1..1
echo ok 1
# Exit with error, even if all the tests have been successful.
exit 7

% cp PREFIX/share/automake-APIVERSION/tap-driver.sh .
% autoreconf -vi && ./configure && make check
...
PASS: foo.test 1 - Swallows fly
XFAIL: foo.test 2 - Caterpillars fly # TODO metamorphosis in progress
SKIP: foo.test 3 - Pigs fly # SKIP not enough acid
PASS: foo.test 4 - Flies fly too :-)
FAIL: bar.test 1 - Bummer, this test has failed.
PASS: bar.test 2 - This passed though.
ERROR: bar.test - Bail out! Ennui kicking in, sorry...
PASS: baz.test 1
ERROR: baz.test - exited with status 7
...
Please report to bug-automake@gnu.org
...
% echo exit status: $?
exit status: 1

% env TEST_LOG_DRIVER_FLAGS='--comments --ignore-exit' \
      TESTS='foo.test baz.test' make -e check
...
PASS: foo.test 1 - Swallows fly
XFAIL: foo.test 2 - Caterpillars fly # TODO metamorphosis in progress
SKIP: foo.test 3 - Pigs fly # SKIP not enough acid
# foo.test: I just love word plays...
PASS: foo.test 4 - Flies fly too :-)
PASS: baz.test 1
...
% echo exit status: $?
exit status: 0
```

15.4.3. Несовместимости с другими анализаторами и драйверами TAP

Драйвер TAP и набор тестов в составе Automake имеют некоторые несовместимости, о которых следует знать.

- Директива Bail out! не останавливает весь набор тестов, прерывая лишь работающий сценарий. Это не соответствует спецификации TAP, но обеспечивает совместимость (и общий код) с концепцией серьёзных ошибок в принятом по умолчанию драйвере тестов.
- Не поддерживаются директивы version и pragma.
- Опция --diagnostic-string позволяет менять строку маркировки диагностики TAP (по умолчанию #). Стандартный протокол TAP не разрешает это, поэтому при использовании этой опции диагностика может быть потеряна для других инструментов, таких как prove и Test::Harness.
- Возможны и другие, пока не замеченные несовместимости.

15.4.4. Ссылки на информацию о протоколе TAP

- Test::Harness::TAP - официальная (в основном документация для формата и протокола TAP.
- prove - наиболее известный консольный драйвер тестов TAP из состава perl и Test::Harness.
- TAP wiki.
- Test::Tutorial - введение в тестирование для программистов perl.
- Стандартные библиотеки perl Test::Simple и Test::More на основе TAP.
- C TAP Harness - проект на основе C с реализацией TAP producer и TAP consumer.
- tap4j - проект на основе Java с реализацией TAP producer и TAP consumer.

15.5. Тесты DejaGnu

Если dejagnu присутствует в AUTOMAKE_OPTIONS, предполагается основанный на dejagnu набор тестов. Переменная DEJATOOL содержит список имён., передаваемых по одному как аргумент --tool при вызовах runtest. По умолчанию это имена пакетов. В переменной RUNTESTDEFAULTFLAGS задаются флаги --tool и --srcdir, передаваемые dejagnu по умолчанию (их можно переопределить). Переменные EXPECT и RUNTEST также можно переопределить для задания специфических значений проекта. Например, это может потребоваться при тестировании инструментов компиляции, поскольку принятые по умолчанию значения не учитывают имён. host и target.

Содержимое пользовательской переменной (3.6. Пользовательские переменные) RUNTESTFLAGS передаётся вызову runtest. Если флаги runtest нужно установить в Makefile.am, следует использовать AM_RUNTESTFLAGS.

Automake будет генерировать правила для создания локального файла site.exp с различными переменными, найденными сценарием configure. Файл автоматически считывается программой DejaGnu. Пользователь пакета может редактировать файл для настройки набора тестов. Однако это не то место, где автору пакета следует задавать новые переменные, они должны определяться в реальном коде набора тестов, а файл site.exp не следует распространять.

Тем не менее, при наличии у автора пакета оснований обработать файл site.exp в процессе работы make, это можно сделать через переменную EXTRA_DEJAGNU_SITE_CONFIG, файлы из которой будут считаться предварительными условиями для site.exp, а их содержимое будет добавлено в конец этого файла (в порядке указания в переменной EXTRA_DEJAGNU_SITE_CONFIG). Отметим, что эти файлы не распространяются по умолчанию.

15.6. Тесты установки

Цель installcheck доступна пользователю как способ запуска любых тестов после установки пакета. Тесты можно добавить, написав правило installcheck-local.

16. Повторная сборка Makefile

Automake генерирует правила автоматической пересборки Makefile, configure и других производных файлов, подобных Makefile.in. При использовании AM_MAINTAINER_MODE в файле configure.ac эти правила автоматической пересборки включаются только в режиме сопровождающего (maintainer).

Иногда удобно добавлять в правила пересборки для configure или config.status дополнительные зависимости с помощью переменных CONFIGURE_DEPENDENCIES и CONFIG_STATUS_DEPENDENCIES. Переменные следует указывать во всех Makefile дерева (эти два правила пересборки выводятся во все эти файлы), поэтому проще и безопасней использовать макрос AC_SUBST из configure.ac. Например, приведённое ниже выражение обеспечит повторный запуск configure при каждом изменении version.sh.

```
AC_SUBST([CONFIG_STATUS_DEPENDENCIES], ['$(top_srcdir)/version.sh'])
```

Отметим $(top_srcdir)/ в имени. Переменная применяется во всех файлах Makefile, поэтому её значение должно иметь смысл на любом уровне иерархии сборки.

Не следует путать переменные CONFIGURE_DEPENDENCIES и CONFIG_STATUS_DEPENDENCIES. Первая добавляет зависимости в правило configure, результатом чего является запуск autoconf. Это переменную не следует применять часто, поскольку automake отслеживает файлы m4_include. Однако она может быть полезна при работе с m4_esyscmd и аналогичными макросами, дающими побочные эффекты. Следует помнить, что взаимодействия с кэш-переменной autom4te достаточно сложны и могут приводить к поломкам. Можно отключить кэш при работе с CONFIGURE_DEPENDENCIES. Переменная CONFIG_STATUS_DEPENDENCIES добавляет зависимости в правило config.status, что ведёт к запуску configure. Поэтому в данную переменную следует включать любой источник, чтение которого может быть побочным результатом запуска configure (например, version.sh из приведённого примера).

Сценарии version.sh сегодня использовать не рекомендуется. Они применяются в основном при автоматическом обновлении версии пакета сценарием. Ниже приведён пример файла configure.ac в «старом стиле».

```
AC_INIT
. $srcdir/version.sh
AM_INIT_AUTOMAKE([name], $VERSION_NUMBER)
...
```

Здесь version.sh является фрагментом оболочки, устанавливающим VERSION_NUMBER. Проблема этого примера заключается в том, что automake не может отследить зависимости (указанные version.sh в переменной CONFIG_STATUS_DEPENDENCIES, и распространение файла зависит от пользователя) и используется устаревшая форма AC_INIT и AM_INIT_AUTOMAKE. Переход к новому синтаксису не прост, поскольку переменные оболочки не разрешены в аргументах AC_INIT. Рекомендуется заменить version.sh файлом M4, включаемым в configure.ac

```
m4_include([version.m4])
AC_INIT([name], VERSION_NUMBER)
AM_INIT_AUTOMAKE
...
```

Файл version.m4 может содержать что-то вроде m4_define([VERSION_NUMBER], [1.2]). В результате automake позаботится о зависимостях при задании правила пересборки и автоматически распространит файл. Однако autoconf будет перезапускаться при каждом роста номера версии, тогда как в старом вариант нужно повторить лишь configure.

17. Смена поведения Automake

17.1 Опции

Свойства Automake можно задавать с помощью опций. За исключением указанных случаев, опции могут задаваться одним из нескольких способов. Большинство опций применяется на уровне файла Makefile при указании в специальной переменной AUTOMAKE_OPTIONS в таких файлах. Некоторые из таких опций имеют смысл лишь в файле Makefile.am верхнего уровня. Опции применяются глобально ко всем обрабатываемым Makefile, указанным в первом аргументе AM_INIT_AUTOMAKE в файле configure.ac, а некоторые опции, которые требуют менять сценарий configure, могут быть указаны только здесь. Обычно опции, заданные в AUTOMAKE_OPTIONS имеют преимущество перед заданными в AM_INIT_AUTOMAKE, которые имеют преимущество перед опциями командной строки.

Необходимо учитывать взаимодействие уровня строгости и категорий предупреждений. Обычно предупреждения, предполагающие строгость, переопределяются теми, которые заданы явными опциями. Например, если предупреждения portability отключены по умолчанию для строгости foreign, приведённая ниже строка включит их.

```
AUTOMAKE_OPTIONS = -Wportability foreign
```

Однако уровень строгости из контекста с более высоким приоритетом переопределит все явно указанные предупреждения из менее приоритетного контекста, например, если в configure.ac указано

```
AM_INIT_AUTOMAKE([-Wportability])
```

а в Makefile.am

```
AUTOMAKE_OPTIONS = foreign
```

предупреждения portability будут отключены файлом Makefile.am.

17.2. Список опций Automake

gnits
gnu
foreign
Задаёт уровень строгости. Опция gnits предполагает readme-alpha и check-news.
check-news
Едет к отказу make dist, если текущий номер версии не задан в нескольких первых строках файла NEWS.
dejagnu
Задаёт генерацию правил, связанных с dejagnu (15.5. Тесты DejaGnu).
dist-bzip2
Переводить dist-bzip2 в dist.

dist-lzip
Переводить dist-lzip в dist.

dist-xz
Переводить dist-xz в dist.

dist-zip
Переводить dist-zip в dist.

dist-shar
Переводить dist-shar в dist. Опция устарела, поскольку формат shar считается устаревшим и проблематичным. Поддержка опции будет исключена в Automake 2.0.

dist-tarZ
Переводить dist-tarZ в dist. Опция устарела, поскольку формат compress, поскольку формат shar считается устаревшим. Поддержка опции будет исключена в Automake 2.0.

filename-length-max=99
Прерывать работу, если обнаружены имена длиннее 99 символов в процессе выполнения make dist. Такие имена файлов в общем случае считаются не переносимыми в архивах. (см. опции tar-v7 и tar-ustar). Опцию следует применять в Makefile.am верхнего уровня или как аргумент AM_INIT_AUTOMAKE в configure.ac, в иных случаях она игнорируется, равно как игнорируется во вложенных пакетах (7.4. Вложенные пакеты).

info-in-builddir
Указывает Automake размещать созданные файлы .info в builddir, а не srcdir. Это может сделать сборки VPATH с некоторыми не-GNU реализациями make нестабильными.

no-define
Опция имеет смысл лишь при передаче в качестве аргумента AM_INIT_AUTOMAKE и предотвращает включение переменных PACKAGE и VERSION в AC_DEFINE. Отметим, что переменные останутся в переменных оболочки, заданных configure и переменных make в Makefile. Это сделано для совместимости с прежними версиями.

no-dependencies
Похожа на опцию командной строки --ignore-deps, но полезна в случаях отсутствия необходимых битов для автоматического отслеживания зависимостей (8.19. Автоматическое отслеживание зависимостей). В этом случае эффект заключается в отключении отслеживания зависимостей.

no-dist
Отключает создание кода для цели dist и полезна при использовании пакетом своих методов распространения.

no-dist-gzip
Не переводить dist-gzip в dist.

no-exeext
Если Makefile.am задаёт правило для цели foo, будет переопределено правило для foo$(EXEEXT). Это требуется в случае пустого значения EXEEXT и по умолчанию automake выдаёт ошибку, которую данная опция блокирует. Опция предназначена лишь для случаев, когда заранее известно, что пакет не будет переноситься в Windows или иную ОС, использующую расширения имён. исполняемых файлов.

no-installinfo
Созданный файл Makefile.in не будет задавать сборку и установку страниц info по умолчанию. Цели info и install-info остаются доступными. Опция отключена при уровне строгости gnu и выше.

no-installman
Созданный файл Makefile.in не будет задавать сборку и установку страниц man по умолчанию. Цель install-man сохраняется. Опция отключена при уровне строгости gnu и выше.

nostdinc
Может применяться для запрета стандартных опций -I, которые обычно представляются Automake автоматически.

no-texinfo.tex
Не требовать texinfo.tex даже при наличии в каталоге файлов texinfo.

serial-tests
Разрешает старые последовательные тесты для переменной TESTS (15.2.2. Последовательные тесты (устарели)).

parallel-tests
Включает в переменную TESTS параллельные тесты (15.2.3. Параллельные тесты) и нужна лишь для совместимости со старыми версиями, поскольку эти тесты включены по умолчанию.

readme-alpha
Если выпуск является предварительным (alpha) и есть файл README-alpha, он добавляется в дистрибутив. При наличии опции предполагается, что номера опций имеют одну из двух форм - MAJOR.MINOR.ALPHA, где все элементы являются цифрами (последняя точка и число опускаются в финальном выпуске) или MAJOR.MINORALPHA, где ALPHA - символ, опускаемый в финальном выпуске.

std-options
Задаёт правилу installcheck проверять поддержку установленными сценариями и программами опций --help и --version, а также обеспечивает базовую проверку зависимостей при работе после установки. В некоторых случаях программы (сценарии) следует исключать из проверки. Например, false (из GNU coreutils) никогда не возвращает успех даже с опцией --help или --version. Такие программы можно указать в переменной AM_INSTALLCHECK_STD_OPTIONS_EXEMPT. Указанные программы (не сценарии) должны иметь расширение $(EXEEXT) для Windows и OS/2. Например, при сборке false как программы и true.sh как сценария (оба не поддерживают опции --help и --version) можно задать

```
AUTOMAKE_OPTIONS = std-options
bin_PROGRAMS = false ...
bin_SCRIPTS = true.sh ...
AM_INSTALLCHECK_STD_OPTIONS_EXEMPT = false$(EXEEXT) true.sh
```

subdir-objects
Задаёт размещение объектов в подкаталоге каталога сборки, соответствующем подкаталогу исходного файла. Например, для файла subdir/file.cxx объектным файлом будет subdir/file.o.

tar-v7
tar-ustar
tar-pax
Эти взаимоисключающие опции задают формат архива для make dist (архив tar сжимается в зависимости от опций no-dist-gzip, dist-bzip2, dist-lzip, dist-xz, dist-tarZ). Опции должны передаваться как аргументы AM_INIT_AUTOMAKE (6.4. Макросы Autoconf из пакета Automake), поскольку они могут требовать дополнительных проверок конфигурации. Automake будет сообщать о таких опциях в переменной AUTOMAKE_OPTIONS.

Опция tar-v7 выбирает старый формат tar V7 и принята по умолчанию. Этот формат понимают все реализации tar и для него поддерживаются имена длиной до 99 символов. Более длинные имена могут вызывать проблемы в некоторых реализациях tar, а другие могут создавать архивы с ошибками или использовать непереносимые расширения. Кроме того, формат V7 не позволяет сохранять пустые каталоги. При использовании этого формата следует задавать опцию filename-length-max=99 для перехвата длинных имён.

Опция tar-ustar задаёт формат ustar, определённый в POSIX 1003.1-1988. Формат достаточно стар и малопереносим. В 2018 г. он поддерживался командой tar в GNU, FreeBSD, NetBSD, OpenBSD, AIX, HP-UX, Solaris. Формат поддерживает пустые каталоги и может сохранять файлы с именами размером до 256 символов при условии, что имя можно по границе каталога разделить на две части, первая из которых не больше 155 байтов. Поэтому во многих случаях максимальный размер имён. будет меньше 256 символов.

Опция tar-pax задаёт новый формат pax, определённый в POSIX 1003.1-2001 и не ограничивающий размер имён. Однако этот формат новый и его применение следует ограничивать лишь современными платформами. В 2018 г. формат поддерживался командой tar в GNU, FreeBSD, OpenBSD и не работал в NetBSD, AIX, HP-UX, Solaris.

Сценарий configure знает несколько способов создания указанных форматов. Сценарий не будет прерван при отсутствии архиватора (пакет можно собрать), но задача make dist завершится отказом.

VERSION

Позволяет задать номер версии (например, 0.30). Если Automake не новее указанной версии, создание файлов Makefile.in будет заблокировано.

-WCATEGORY

--warnings=CATEGORY

Эти опции ведут себя как их варианты для командной строки (5. Создание Makefile.in) и позволяют контролировать вывод предупреждений на уровне файлов. Можно также настроить некоторые предупреждения для проекта в целом, например, задав AM_INIT_AUTOMAKE([-Wall]) в файле configure.ac.

Нераспознанные опции указываются программой automake. Если нужно применить опцию во всем дереве, можно задать макрос AM_INIT_AUTOMAKE в файле (6.4. Макросы Autoconf из пакета Automake).

18. Прочие правила

18.1. Взаимодействие с etags

В некоторых случаях Automake создаёт правила генерации файлов TAGS для использования с GNU Emacs. При наличии исходного кода или заголовков C, C++ или Fortran 77 для каталога создаются правила tags и TAGS с использованием всех файлов из первичных переменных _SOURCES, _HEADERS и _LISP. Отметим, что создаваемые исходные файлы, которые не нужно распространять, должны быть указаны в переменных вида nodist_noinst_HEADERS или nodist_PROG_SOURCES, иначе они игнорируются.

Правило tags будет выводиться в верхний каталог дерева и при запуске из этого каталога команда make tags будет создавать файл TAGS, включающий ссылки на все файлы TAGS в подкаталогах. Правило tags создаётся также при задании переменной ETAGS_ARGS, которая предназначена для использования в каталогах, содержащих помечаемые исходные файлы, которые etags не понимает. Пользователь может задать ETAGSFLAGS для передачи etags дополнительных флагов. Доступен также макрос AM_ETAGSFLAGS для файлов Makefile.am.

Ниже показано, как Automake генерирует теги для источников и узлов в файле Texinfo.

```
ETAGS_ARGS = automake.in --lang=none \
 --regex='/^@node[ \t]+\([^,]+\)/\1/' automake.texi
```

При добавлении имён. файлов в ETAGS_ARGS может возникнуть желание задать TAGS_DEPENDENCIES. Содержимое этой переменной напрямую добавляется в зависимости для правила tags.

Automake также генерирует правило ctags, которое может служить для сборки файлов tags в стиле vi. Переменная CTAGS указывает имя вызываемой программы (по умолчанию ctags), а CTAGSFLAGS может применяться пользователем для задания дополнительных флагов. Доступен также макрос AM_CTAGSFLAGS в Makefile.am.

Automake создаёт правило ID, которое будет запускать для источника. Это поддерживается лишь на уровне каталогов.

Правило cscope создаёт список исходных файлов и запускает cscope для сборки базы данных с инвертированным индексом. Переменная CSCOPE задаёт имя вызываемой программы (по умолчанию cscope), а CSCOPEFLAGS и CSCOPE_ARGS могут служить для передачи дополнительных флагов и имён. Можно также использовать макрос AM_CSCOPEFLAGS в Makefile.am. Отметим, что поддержка cscope в Automake в настоящее время может не работать корректно при сборке VPATH с отличными от GNU реализациями make implementations.

Automake создаёт правила для поддержки программы GNU Global Tags program. Правило GTAGS запускает gtags и помещает результат в каталог сборки на верхний уровень. Переменная GTAGS_ARGS содержит аргументы gtags.

18.2. Обработка новых расширений имён. файлов

Иногда полезно задать новое неявное правило для обработки неизвестных Automake типов файлов.

Предположим, что компилятор создаёт из файлов .foo объектные файлы .o. Можно задать правило для суффикса

```
.foo.o:
        foocc -c -o $@ $<
```

Тогда можно напрямую указывать файлы .foo в переменной _SOURCES и предполагать корректный результат.

```
bin_PROGRAMS = doit
doit_SOURCES = doit.foo
```

Это был наиболее простой и распространённый случай. В других случаях нужно будет помочь Automake выяснить суффиксы для включения в правило. Обычно это связано с расширениями, не начинающимися с точки. Затем нужно лишь поместить список новых суффиксов в переменную SUFFIXES перед созданием неявного правила. Например, приведённое ниже определение предотвращает в Automake ошибочную интерпретацию правила .idlC.cpp: как попытки преобразовать файлы .idlC в .cpp.

```
SUFFIXES = .idl C.cpp
.idlC.cpp:
        # whatever
```

Как можно увидеть, переменная SUFFIXES похожа на специальную цель .SUFFIXES для make. Не следует трогать цель .SUFFIXES напрямую, но можно использовать SUFFIXES, чтобы программа Automake создавала список суффиксов для .SUFFIXES. Вслед за содержимым SUFFIXES в создаваемом списке размещаются найденные Automake суффиксы, которых ещё нет в списке.

19. Директива include

Automake поддерживает директиву include для включения других фрагментов Makefile при работе automake. Фрагменты считываются и обрабатываются automake, а не make. Как и у условных выражениях, make не знает об include.

include $(srcdir)/file

Включить фрагмент, заданный относительно текущего каталога источников.

include $(top_srcdir)/file

Включить фрагмент, заданный относительно верхнего уровня каталога источников.

Отметим, что при включении фрагмента внутри условной конструкции условие относится ко всему фрагменту.

Включаемые таким способом фрагменты Makefile всегда распространяются, так как они нужны для сборки Makefile.in.

Внутри фрагмента конструкция %reldir% заменяется каталогом фрагмента относительно базы Makefile.am, а %canon_reldir% заменяется канонизированной (3.5. Именование производных переменных) формой %reldir%. Для удобства %D% служит синонимом %reldir%, а %C% - %canon_reldir%. Особенность состоит в том, что при размещении фрагмента в одном каталоге с базовым Makefile.am (%reldir% = .) переменные %reldir% и %canon_reldir% преобразуются в пустую строку, поглощая следующий символ / или _.

Фрагмент makefile может иметь вид

```
bin_PROGRAMS += %reldir%/mumble
%canon_reldir%_mumble_SOURCES = %reldir%/one.c
```

20. Конструкции с условием

Automake поддерживает простые конструкции с условием, которые отличаются от условных конструкций GNU Make. В Automake условия проверяются во время работы сценария configure и влияют на трансляцию файлов Makefile.in в Makefile. Проверка основывается на опциях, переданных configure и результатах, полученным сценарием при изучении системы. Условия в GNU Make проверяются во время работы make и основаны на переменных, переданных программе make или заданных в Makefile. Условные конструкции Automake работают с любой программой make.

20.1. Использование условных конструкций

Перед использованием условия его нужно задать макросом AM_CONDITIONAL в configure.ac (6.4. Макросы Autoconf из пакета Automake). Макрос AM_CONDITIONAL (CONDITIONAL, CONDITION) принимает имя условия CONDITIONAL, которое должно начинаться с буквы и включать лишь буквы, цифры и символы подчёркивания, а также не может принимать значение TRUE или FALSE. Условие CONDITION (подходит для оператора оболочки if) проверяется в процессе работы configure. Важно отметить, что каждый макрос AM_CONDITIONAL должен вызываться при каждом запуске . Если макрос AM_CONDITIONAL вызывается по условию (например, в операторе if), результат может привести к путанице в automake. Условия обычно зависят от опций, предоставляемых пользователем сценарию configure. Ниже приведён пример условия, дающего в результате true, если пользователь указал опцию --enable-debug.

```
AC_ARG_ENABLE([debug],
[  --enable-debug    Turn on debugging],
[case "${enableval}" in
  yes) debug=true ;;
  no)  debug=false ;;
  *) AC_MSG_ERROR([bad value ${enableval} for --enable-debug]) ;;
esac],[debug=false])
AM_CONDITIONAL([DEBUG], [test x$debug = xtrue])
```

Ниже приведён пример использования условия в файле Makefile.am.

```
if DEBUG
DBG = debug
else
DBG =
endif
noinst_PROGRAMS = $(DBG)
```

Этот тривиальный пример можно обработать с помощью EXTRA_PROGRAMS (8.1.4. Условная компиляция программ).

В операторе if может проверяться лишь одна переменная и разрешено обращение с помощью знака !, а оператор else можно опускать. Условия допускают вложенность произвольной глубины. Можно задать аргумент для else, который в этом случае должен быть отрицанием условия в текущем операторе if. Аналогично можно задать условие, завершаемое строкой endif.

```
if DEBUG
DBG = debug
else !DEBUG
DBG =
endif !DEBUG
```

Несбалансированные условия являются ошибками. Для операторов if, else и endif не следует использовать отступ.

Ветвь else в приведённых выше примерах может быть опущена, поскольку назначение пустой строки не определённой ранее переменной ничего не меняет.

Чтобы обеспечить доступ к условию, заданному AM_CONDITIONAL в configure.ac, и разрешить условие AC_CONFIG_FILES, можно применить AM_COND_IF. Макрос AM_COND_IF (CONDITIONAL, [IF-TRUE], [IF-FALSE]) при условии CONDITIONAL выполняет IF-TRUE, в ином случае - IF-FALSE. Если любая из ветвей содержит AC_CONFIG_FILES, это заставит automake вывести правила для соответствующих файлов только при данном условии.

Макросы AM_COND_IF могут быть вложенными при корректном использовании m4. Ниже приведён пример задания условного файла config.

```
AM_CONDITIONAL([SHELL_WRAPPER], [test "x$with_wrapper" = xtrue])
AM_COND_IF([SHELL_WRAPPER],
           [AC_CONFIG_FILES([wrapper:wrapper.in])])
```

20.2. Ограничения условных конструкций

Условия должны содержать завершённые операторы, такие как определения переменных или правил. Automake не может работать с условиями внутри определений переменных и даже не может диагностировать такие ситуации

```
# Automake не поймёт такой синтаксис
AM_CPPFLAGS = \
  -DFEATURE_A \
if WANT_DEBUG
  -DDEBUG \
endif
  -DFEATURE_B
```

Однако определение можно реализовать с помощью AM_CPPFLAGS

```
if WANT_DEBUG
  DEBUGFLAGS = -DDEBUG
endif
AM_CPPFLAGS = -DFEATURE_A $(DEBUGFLAGS) -DFEATURE_B
```

или

```
AM_CPPFLAGS = -DFEATURE_A
if WANT_DEBUG
AM_CPPFLAGS += -DDEBUG
endif
AM_CPPFLAGS += -DFEATURE_B
```

Дополнительная информация и примеры приведены в параграфах 7.2. Условные подкаталоги, 8.1.3. Условная компиляция исходного кода, 8.1.4. Условная компиляция программ, 8.3.3. Условная сборка библиотек Libtool, 8.3.4. Библиотеки Libtool с условными источниками.

21. Молчаливая работа make

21.1. Подробный вывод по умолчанию

Обычно при выполнении правил, связанных с целью make выводит каждое правило перед его выполнением. Такое поведение существует достаточно долго и даже предписывается стандартом POSIX, но нарушает принцип UNIX «молчание - золото»

> Когда программе нечего сказать интересного или удивительного, ей следует молчать. Хорошие программы Unix выполняют работу незаметно с минимальной суетой и беспокойством. Молчание - золото!

Хотя многословие make теоретически может быть полезным для отслеживания ошибок и причин непонятных отказов, оно также может скрывать предупреждения и сообщения об ошибках от привлекаемых make инструментов, топя их в потоке неинтересной и малополезной информации. Это может сильно раздражать, особенно разработчиков, которые обычно хорошо знают происходящее «за кулисами». Подробный вывод make для них представляет в основном шум, скрывающий потенциально важные предупреждения.

21.2. Как заставить make замолчать

Ниже описаны некоторые распространённые приёмы обеспечения молчаливой работы make с их преимуществами и недостатками. В параграфе 21.3. Как Automake может «заглушить» make показано, как Automake может помочь в этом.

make -s

Эта опция просто отключает в make вывод правил перед их выполнением. Опция обязательна для POSIX, поддерживается всеми и назначение её легко понять. Но у этого подхода есть ряд серьёзных ограничений. Во-первых, он работает по принципу «все или ничего», что далеко не всегда приемлемо. Кроме того, при использовании флага -s вывод make может даже не позволять заметить ошибок или связанных с ними правил.

make >/dev/null || make

Этот вариант следует правилу «молчание - золото». Предупреждения от stderr проходят, а выходной отчёт печатается лишь в случае ошибки и в этом случае ему следует быть достаточно подробным, чтобы определить причину и место ошибки. Однако два вызова make в одной строке могут скрывать ошибки (особенно чередующиеся) или менять ожидаемую семантику вызовов make, явно усложняя отладку и поиск ошибок.

make --no-print-directory

Этот метод относится к GNU make. Опция —no-print-directory отключает вывод рабочего каталога при вызове дочерних процессов make (сообщения о входе и выходе для каталогов), снижая объем информации. Однако опыт показывает, что это затрудняет отладку проектов с большой глубиной вложенности. Опция --no-print-directory автоматически активируется при задании флага -s.

21.3. Как Automake может «заглушить» make

Приведённые выше советы могут быть иной раз полезны, но у них имеются серьёзные недостатки. Поэтому automake включает более развитый и гибкий способ снизить объем выводимой make информации (для большинства правил).

Сравним обычный вывод make и вывод со включёнными «правилами тишины».

```
% cat Makefile.am
bin_PROGRAMS = foo
foo_SOURCES = main.c func.c
% cat main.c
int main (void) { return func (); }  /* func used undeclared */
% cat func.c
int func (void) { int i; return i; } /* i used uninitialized */
```

Вывод make обычно очень детален и это зачастую скрывает предупреждения компилятора.

```
% make CFLAGS=-Wall
gcc -DPACKAGE_NAME=\"foo\" -DPACKAGE_TARNAME=\"foo\" ...
-DPACKAGE_STRING=\"foo\ 1.0\" -DPACKAGE_BUGREPORT=\"\" ...
-DPACKAGE=\"foo\" -DVERSION=\"1.0\" -I. -Wall -MT main.o
-MD -MP -MF .deps/main.Tpo -c -o main.o main.c
main.c: In function 'main':
main.c:3:3: warning: implicit declaration of function 'func'
mv -f .deps/main.Tpo .deps/main.Po
gcc -DPACKAGE_NAME=\"foo\" -DPACKAGE_TARNAME=\"foo\" ...
-DPACKAGE_STRING=\"foo\ 1.0\" -DPACKAGE_BUGREPORT=\"\" ...
-DPACKAGE=\"foo\" -DVERSION=\"1.0\" -I. -Wall -MT func.o
-MD -MP -MF .deps/func.Tpo -c -o func.o func.c
func.c: In function 'func':
func.c:4:3: warning: 'i' used uninitialized in this function
mv -f .deps/func.Tpo .deps/func.Po
gcc -Wall -o foo main.o func.o
```

Очистим сборку для повтора с «чистого листа».

```
% make clean
test -z "foo" || rm -f foo
rm -f *.o
```

Правила молчанию включены и вывод стал меньше, но информативней и предупреждения компилятора видны.

```
% make V=0 CFLAGS=-Wall
  CC     main.o
main.c: In function 'main':
main.c:3:3: warning: implicit declaration of function 'func'
  CC     func.o
func.c: In function 'func':
func.c:4:3: warning: 'i' used uninitialized in this function
  CCLD   foo
```

В проектах с использованием libtool правила тишины могут автоматически включать опцию libtool --silent.

```
% cat Makefile.am
lib_LTLIBRARIES = libx.la

% make # Both make and libtool are verbose by default.
...
libtool: compile: gcc -DPACKAGE_NAME=\"foo\" ... -DLT_OBJDIR=\".libs/\"
  -I. -g -O2 -MT libx.lo -MD -MP -MF .deps/libx.Tpo -c libx.c -fPIC
  -DPIC -o .libs/libx.o
mv -f .deps/libx.Tpo .deps/libx.Plo
/bin/sh ./libtool --tag=CC --mode=link gcc -g -O2 -o libx.la -rpath
  /usr/local/lib libx.lo
libtool: link: gcc -shared .libs/libx.o -Wl,-soname -Wl,libx.so.0
  -o .libs/libx.so.0.0.0
libtool: link: cd .libs && rm -f libx.so && ln -s libx.so.0.0.0 libx.so
...

% make V=0
  CC     libx.lo
  CCLD   libx.la
```

Для созданных Automake файлов Makefile пользователь может изменить уровень вывода при запуске configure и make:

- configure --enable-silent-rules обеспечивает создание правил с менее подробным выводом, опция --disable-silent-rules задаёт обычный вывод;
- при работе make заданный configure вывод можно переопределить, например, make V=1 задаёт подробный вывод, а make V=0 - менее детальный.

Отметим, что «правила тишины» по умолчанию выключены и пользователю нужно явно включить их в команде configure или make. Это считается хорошим тоном, поскольку позволяет пользователю возможность управления.

Разработчик, желающий снизить объем выводимой информации, может включить правила тишины по умолчанию, вызвав макрос AM_SILENT_RULES([yes]) из configure.ac. Если пользователь хочет включить правила тишины по умолчанию, он может указать в файле config.site переменную enable_silent_rules со значением yes. Это не помешает управления выводом в командах configure и make.

Для переносимости между разными реализациями make авторам пакетов не рекомендуется устанавливать переменную V в файле Makefile.am, чтобы позволить пользователю переопределять значения и для подкаталогов.

Для более эффективной работы в текущей реализации этой функции обычно применяется преобразование вложенной переменной $(VAR1$(V)), которое не требуется стандартом POSIX 2008, но широко применяется на практике. В редких реализациях make, не поддерживаемых преобразование вложенных переменных, правила молчания определяются при работе configure и не могут быть переопределены для make. В будущих версиях POSIX вероятно придётся пересмотреть преобразование вложенных переменных, поэтому ограничение может со временем исчезнуть.

Для задания тишины в пользовательских правилах имеется два варианта, рассмотренных ниже.

- Можно использовать предопределённую переменную AM_V_GEN как префикс команд, которым следует выводить строку состояния в режиме тишины и AM_V_at - для команд, которым не следует выводить ничего. При подробном выводе обе переменные преобразуются в пустые строки.
- Можно заставить задание замолчать безоговорочно с помощью @ и затем применять переменную AM_V_P, чтобы узнать режим вывода make, настраивая уровень подробности вывода задания, как показано ниже.

```
generate-headers:
        ... [команды, задающие переменную среды '$headers'] ...; \
    if $(AM_V_P); then set -x; else echo " GEN   [headers]"; fi; \
    rm -f $$headers && generate-header --flags $$headers
```

- Можно добавить свои переменные для показа строк по вашему выбору. Приведённый ниже фрагмент определение эквивалента переменной AM_V_GEN'

```
pkg_verbose = $(pkg_verbose_@AM_V@)
pkg_verbose_ = $(pkg_verbose_@AM_DEFAULT_V@)
pkg_verbose_0 = @echo PKG-GEN $@;
```

```
foo: foo.in
        $(pkg_verbose)cp $(srcdir)/foo.in $@
```

В заключение отметим, что даже при включённых правилах тишины опция --no-print-directory нужна GNU make, если не нужно видеть сообщений о входе и выходе для каталогов.

22. Опции --gnu и --gnits

Опция --gnu (или gnu в переменной AUTOMAKE_OPTIONS) заставляет automake проверять указанные ниже условия.

- Наличие файлов INSTALL, NEWS, README, AUTHORS и ChangeLog, а также одного из файлов COPYING.LIB, COPYING.LESSER или COPYING в каталоге верхнего уровня.

  Если задана опция --add-missing, automake будет добавлять базовый вариант файла INSTALL и файл COPYING с текстом текущей версии GNU General Public License на момент выпуска Automake (сейчас версия 3). Однако имеющийся COPYING никогда не переписывается программой automake.

- Опции no-installman и no-installinfo запрещены.

В будущем опция --gnu может быть расширена дополнительными проверками (желательно ознакомиться с требованиями стандартов GNU). Кроме того, могут требоваться некоторые нестандартные программы для различных правил сопровождающих, например, в будущем может потребоваться pathchk для цели make dist.

Опция --gnits выполняет все проверки --gnu и ряд дополнительных.

- Команда make installcheck проверяет реальный вывод командами --help и --version справочной информации и номера версии. Это опция std-options (17. Смена поведения Automake).
- Команда make dist проверяет наличие файла NEWS и его обновления до текущей версии.
- Проверяется переменная VERSION на предмет соответствия стандартам Gnits.
- Если переменная VERSION указывает предварительный (alpha) выпуск и в каталоге верхнего уровня имеется файл README-alpha, этот файл включается в распространение. Это делается только в режиме --gnits, поскольку лишь в этом режиме формат версии ограничен, поэтому только в нем Automake может автоматически решить вопрос включения файла README-alpha.
- Наличие файла THANKS.

23. Когда Automake не достаточно

В некоторых случаях Automake не может полностью решить задачу и приходиться писать правила вручную.

23.1. Расширение правил Automake

За небольшими исключениями (например, переменные _PROGRAMS, TESTS, XFAIL_TESTS, которые переписываются для добавления в $(EXEEXT)), содержимое файла Makefile.am дословно копируется в Makefile.in. Семантика копирования означает, что многие проблемы можно обойти, просто добавив некоторые переменные и правила make в файл Makefile.am. Automake игнорирует такие добавления. Поскольку Makefile.in создаётся на основе данных, собранных из разных мест (Makefile.am, configure.ac, automake), могут возникнуть конфликты между определениями правил или переменных. При сборке Makefile.in программа automake учитывает указанные ниже приоритеты оставляя за пользователем последнее слово.

- Пользовательские переменные в Makefile.am имеют приоритет на AC_SUBST из configure.ac, а переменные AC_SUBST - над переменными, заданными automake.
- Заданное пользователем правило переопределяет любой правило automake для той же цели.

Такая семантика позволяет точно настроить принятые по умолчанию установки Automake или заменить некоторые правила. Переопределение правил Automake зачастую нежелательно, особенно на верхнем уровне пакета с подкаталогами. Опция -Woverride (5. Создание Makefile.in) полезна для поиска переопределений вручную. Отметим, что Automake не различает правила с командами и правила лишь с зависимостями. Поэтому невозможно добавить зависимости для заданной automake цели без переопределения всего правила. Однако некоторые полезные цели имеют вариант -local, который можно задать в Makefile.am. Automake будет дополнять стандартную цель предоставленными пользователем целями. Локальные версии поддерживаются для целей all, info, dvi, ps, pdf, html, check, install-data, install-dvi, install-exec, install-html, install-info, install-pdf, install-ps, uninstall, installdirs, installcheck и разных целей clean (mostlyclean, clean, distclean, maintainer-clean). Цели uninstall-exec-local и uninstall-data-local отсутствуют, используется просто uninstall-local, поскольку нет смысла удалять лишь данные или исполняемые файлы. Ниже приведён пример удаления подкаталога по команде make clean (13. Очистка).

```
clean-local:
        -rm -rf testSubDir
```

Может возникнуть желание использовать install-data-local для установки файла в жёстко заданное место, но этого следует избегать (27.10. Установка в жёстко заданные места).

Для целей -local нет особой гарантии порядка выполнения, обычно они запускаются рано но при параллельной работе make это не гарантируется.

У некоторых правил есть способ запустить другое правило, называемый ловушкой (hook), при этом ловушки всегда выполняются после основного правила. Ловушки именуются по основной цели с суффиксом -hook и поддерживаются для целей install-data, install-exec, uninstall, dist и distcheck. Ниже приведён пример создания жёсткой ссылки на установленную программу.

```
install-exec-hook:
        ln $(DESTDIR)$(bindir)/program$(EXEEXT) \
           $(DESTDIR)$(bindir)/proglink$(EXEEXT)
```

Хотя жёсткие ссылки дешевле и переносимее символьных, они работают не всегда (например, в OS/2 не ln). В идеале следует вернуться к cp -p, когда ln не работает. Если приемлемы символьные ссылки, проще всего добавить AC_PROG_LN_S в configure.ac и использовать $(LN_S) в Makefile.am. Ниже показано, как можно установить версионную копию программы с использованием $(LN_S).

```
install-exec-hook:
        cd $(DESTDIR)$(bindir) && \
          mv -f prog$(EXEEXT) prog-$(VERSION)$(EXEEXT) && \
          $(LN_S) prog-$(VERSION)$(EXEEXT) prog$(EXEEXT)
```

Отметим переименование программы, в результате чего новая версия будет удалять символьную ссылку, а не реальный файл. Переход в целевой каталог (cd) служит для создания относительных ссылок. При создании install-exec-hook или install-data-hook следует учитывать, что различие исполняемых файлов и данных основано на каталогах, а не первичных переменных (2.2.7. Раздельная установка). Поэтому foo_SCRIPTS будет устанавливаться install-data, а barexec_SCRIPTS - install-exec.

23.2. Сторонние файлы Makefile

В большинстве проектов все Makefile создаются Automake, но иногда во вложенных каталогах нужны созданные вручную Makefile. Например, каталог может включать сторонний проект со своей системой сборки без Automake. Можно указать любые каталоги в переменной SUBDIRS или DIST_SUBDIRS для указания наличия в них Makefile, распознающих приведённые ниже рекурсивные цели. При запуске одной из таких целей будет выполняться рекурсия во все каталоги, поэтому сторонние Makefile должны поддерживать эти цели.

all
: Сборка пакета целиком (задано по умолчанию для создаваемых Automake Makefile, но не обязательно сторонних).

distdir
: Копировать файлы для распространения в $(distdir) перед созданием архива. Цель не нужна при использовании опции no-dist (17. Смена поведения Automake). Переменные $(top_distdir) и $(distdir) (14.3. «Ловушка» dist) могут быть переданы из внешней программы в субпакет при вызове цели distdir. Эти переменные задаются для каталога, в который выполняется рекурсия, поэтому они готовы к использованию.

install
install-data
install-exec
uninstall
: Установить или удалить установленные файлы (12. Установка).

install-dvi
install-html
install-info
install-ps
install-pdf
: Установить документацию в указанном формате (11.1. Texinfo).

installdirs
: Создать каталоги для установки, не устанавливая файлов.

check
installcheck
: Проверка пакета (15. Поддержка тестов).

mostlyclean
clean
distclean
maintainer-clean
: Правила очистки (13. Очистка).

dvi
pdf
ps
info
html
: Сборка документации в разных форматах (11.1. Texinfo).

tags
ctags
: Сборка TAGS и CTAGS (18.1. Взаимодействие с etags).

Проекты с Gettext являются хорошим примером использования сторонних Makefile с Automake. Makefile из gettextize помещает в каталоги po/ и intl/ рукописные Makefile, реализующие указанные выше цели. Таким образом они могут быть добавлены в SUBDIRS пакетов Automake. Каталоги, указанные в DIST_SUBDIRS, но не в SUBDIRS, должны иметь лишь правила distclean, maintainer-clean и distdir (7.2. Условные подкаталоги). Обычно многие из этих правил не имеют отношения к стороннему субпроекту, но нужны для работы пакета в целом. Наличие ничего не делающих правил является нормой в сторонних пакетах, поэтому при отсутствии документации или поддержки тегов в стороннем пакете можно просто дополнить его Makefile, как показано ниже.

```
EMPTY_AUTOMAKE_TARGETS = dvi pdf ps info html tags ctags
.PHONY: $(EMPTY_AUTOMAKE_TARGETS)
$(EMPTY_AUTOMAKE_TARGETS):
```

Другим аспектом взаимодействия со сторонними системами сборки является поддержка сборки VPATH (2.2.6. Параллельная сборка (VPATH)). Обычно при отсутствии в субпакете поддержки VPATH пакет целиком не поддерживает сборки VPATH. Это означает, что make distcheck не будет работать, поскольку эта цель опирается на сборки VPATH. Это устраивает многие (не все пользователи Automake слышали о make distcheck), но другие предпочтут обновить имеющиеся Makefile для поддержки VPATH. Для этого не обязательно требуется Automake и достаточно Autoconf. Необходимые подстановки @srcdir@, @top_srcdir@ и @top_builddir@ определяются configure при обработке Makefile и они не задаются Makefile, подобно упомянутым выше $(distdir) и $(top_distdir).

Иногда неудобно редактировать сторонний Makefile для добавления указанных выше целей. Например, это может быть сделано для упрощения перехода к новым версиям стороннего проекта. Можно пойти иным путём. Если предполагается GNU make одним из вариантов является добавление в подкаталог GNUmakefile с требуемыми целями

и включением Makefile. Чтобы это работало со сборкой VPATH, GNUmakefile должен размещаться в каталоге сборки. Проще всего это сделать путём создания файла GNUmakefile.in и его обработки макросом AC_CONFIG_FILES из внешнего пакета. Например, если Makefile определяет все цели, кроме документации, а цель check называется test, можно создать GNUmakefile (или GNUmakefile.in) вида

```
# Сначала включается реальный Makefile
include Makefile
# Затем определяются цели, требуемые для файлов Automake Makefile
.PHONY: dvi pdf ps info html check
dvi pdf ps info html:
check: test
```

Другая идея без использования include заключается в создании промежуточного (proxy) Makefile, диспетчеризующего правила в реальный Makefile с целью $(MAKE) -f Makefile.real $(AM_MAKEFLAGS) target (для переименования исходного Makefile) или командой cd subdir && $(MAKE) $(AM_MAKEFLAGS) target (для сохранения проекта на один каталог глубже). Промежуточный файл Makefile можно создать с помощью Automake. Для этого нужны лишь цели -local (23.1. Расширение правил Automake), выполняющие диспетчеризацию. Доступны и другие функции Automake, поэтому можно разрешить Automake распространение или установку. Ниже приведён возможный файл Makefile.am.

```
all-local:
        cd subdir && $(MAKE) $(AM_MAKEFLAGS) all
check-local:
        cd subdir && $(MAKE) $(AM_MAKEFLAGS) test
clean-local:
        cd subdir && $(MAKE) $(AM_MAKEFLAGS) clean

# Предполагается, что пакет знает как установить себя
install-data-local:
        cd subdir && $(MAKE) $(AM_MAKEFLAGS) install-data
install-exec-local:
        cd subdir && $(MAKE) $(AM_MAKEFLAGS) install-exec
uninstall-local:
        cd subdir && $(MAKE) $(AM_MAKEFLAGS) uninstall

# Распространять файлы отсюда
EXTRA_DIST = subdir/Makefile subdir/program.c ...
```

Доведя эту идею до крайности, можно игнорировать систему сборки субпроекта и собрать все из промежуточного файла Makefile.am. Это может быть разумно, если нудна сборка VPATH, а субпроект её не поддерживает.

24. Распространение Makefile.in

Automake не задаёт ограничений на распространение файлов Makefile.in. Авторам пакетов рекомендуется распространять свои работы их на условиях, подобных лицензии GPL, но для этого не требуется использовать Automake. Некоторые из файлов, автоматически устанавливаемых при использовании опции --add-missing, попадают под действие GPL. Однако для них имеются специальные исключения, позволяющие распространять файлы с пакетом, независимо от выбранного лицензирования.

25. Версии Automake API

Новые выпуски Automake обычно включают исправления ошибок и новые функции, но к сожалению могут вносить новые ошибки и несовместимости. Этим обусловлены 4 причины, по которым пакету может требоваться конкретная версия Automake. Ситуация усложняется при поддержке большого дерева пакетов, которым могут требоваться разные версии Automake. Раньше это требовало от разработчика (иногда и от пользователя) установки нескольких версий Automake в разных местах и соответствующего переключения $PATH для каждого пакета.

Начиная с версии 1.6, Automake устанавливает версионные исполняемые файлы, что позволяет иметь несколько версий Automake в одном каталоге $prefix, и выбирать нужную версию Automake командами вида automake-1.6 или automake-1.7 без манипуляция с $PATH. Кроме того, Makefile, созданные Automake 1.6 будут явно использовать automake-1.6 в правилах пересборки. Значение 1.6 в automake-1.6 указывает версию Automake API, а не самой программы. При выпуске, например, Automake 1.6.1 с исправлением ошибок версия API останется 1.6 и пакет, работавший с Automake 1.6, продолжит работать с 1.6.1, а после этого ожидается выпуск с исправлением ошибок. Если пакет опирается на исправление ошибки или функцию, добавленную в выпуске, можно передать эту версию Automake, чтобы старые выпуски не применялись. Например, можно указать в файле configure.ac

```
AM_INIT_AUTOMAKE([1.6.1])    dnl Require Automake 1.6.1 or better.
```

или в отдельном файле Makefile.am

```
AUTOMAKE_OPTIONS = 1.6.1   # Require Automake 1.6.1 or better.
```

Automake будет выдавать сообщение об ошибке, если его версия старше запрашиваемой.

Что есть в API

Определить программный интерфейс Automake не так просто. По сути, он должен включать как минимум все документированные переменные и цели, которые можно применять в Makefile.am, и описывать связанное с ними поведение (например, точки запуска -hook), командный интерфейс automake и aclocal, ...

Чего нет в API

Недокументированные переменные, цели и опции команд не являются частью API и следует избегать их применения, поскольку они могут меняться от версии к версии. Если недокументированные возможности нужны, следует обратиться по адресу для возможного документирования и выполнения их в тестовом комплекте (test-suite).

26. Перевод пакета на новую версию Automake

Automake поддерживает в пакете 3 типа файлов:

- aclocal.m4;
- Makefile.in;

- дополнительные инструменты, такие как install-sh и py-compile.

Файлы aclocal.m4 создаются программой aclocal и содержат некоторые макросы M4 от Automake. Дополнительные инструменты устанавливаются при необходимости командой automake --add-missing. Файлы Makefile.in создаются из Makefile.am командой automake на основе определений макросов M4 из aclocal.m4, а также действий дополнительных инструментов. Поскольку все эти файлы тесно связаны, важно регенерировать их при переходе к новому выпуску Automake. Обычно для этого применяется приведённая ниже последовательность команд.

```
aclocal # с нужными опциями (такими как -I m4)
autoconf
automake --add-missing --force-missing
```

или более коротким путём

```
autoreconf -vfi
```

Опция --force-missing обеспечивает обновление дополнительных инструментов (5. Создание Makefile.in). Важно обновить все эти файлы при каждом обновлении Automake, даже если новый выпуск лишь исправляет ошибки. Например, исправление ошибки может включать изменения правил, создаваемых в Makefile.in и поддержку макросов M4, копируемых в aclocal.m4.

В настоящее время automake может диагностировать ситуации, когда файл aclocal.m4 был создан другой версией aclocal, однако актуальность дополнительных инструментов не проверяется. Т. е. automake скажет о необходимости повторного запуска aclocal, но не сообщит о необходимости использовать --force-missing. Разумно прочитать файл NEWS перед обновлением. В этом файле указываются все различия - новые функции, устаревшие конструкции, известные несовместимости и способы обхода.

27. Часто задаваемые вопросы об Automake

Здесь рассмотрены некоторые вопросы, часто задаваемые в почтовых конференциях.

27.1. CVS и генерируемые файлы

Распространение сгенерированных файлов

Пакеты, собранные с Autoconf и Automake, распространяются с некоторыми сгенерированными файлами, такими как configure и Makefile.in. Эти файлы создаются в системе разработчика и распространяются для того, чтобы пользователям не приходилось устанавливать некоторые инструменты, нужные для сборки. Другие сгенерированные файлы (сканеры Lex, анализаторы Yacc, документация Info) распространяются с аналогичными целями. Automake выводит в Makefile правила для пересборки этих файлов. Например, make будет запускать autoconf для пересборки configure при каждом изменении configure.ac. Это делает разработку более безопасной и обеспечивает соответствие сценария configure действительному файлу configure.ac. Поскольку сгенерированные файлы из пакетов могут устаревать, а tar сохраняет временные метки, правила пересборки не запускаются при распаковке и первой сборке пакета.

CVS и временные метки

Если не используется ключевое слово CVS (что требует обновления файлов при фиксации), CVS сохраняет временные метки при операциях cvs commit и cvs import -d. При извлечении файлов по команде cvs checkout временные метки устанавливаются по выбранному выпуску. Однако по команде cvs update файлы получат дату обновления, а не исходную временную метку выпуска. Это сделано для того, чтобы обеспечить уведомление make об обновлении исходных файлов. Такой сдвиг временных меток становится хлопотным, когда источники и сгенерированные файлы хранятся в CVS. Поскольку CVS обрабатывает файлы в лексическом порядке, файл configure.ac будет новее configure после команды cvs update, обновляющей оба файла, дае если файл configure был новей configure.ac при проверке. Вызов make приведёт в результате к ненужной пересборке.

Сопровождающие делятся на два клана - те, кто хранит все распространяемые файлы в CVS, и те, кто держит генерируемые файлы за пределами CVS.

Все файлы в CVS

- Репозиторий CVS содержит все распространяемые файлы и можно извлечь прежнюю версию целиком.
- Сопровождающий может видеть изменения генерируемых файлов (например, изменение Makefile.in при обновлении Automake).
- Пользователю не нужно иметь autotools для сборки загруженного проекта, можно работать как с архивом.
- При использовании команды cvs update для обновления копии вместо cvs checkout для извлечения свежей версии, временные метки будут неточными, что активирует некоторые правила пересборки с попыткой запуска инструментов разработки, таких как autoconf и automake.

  Обращения к таким инструментам выполняются путём вызова сценария missing (27.2. Сценарий missing и режим AM_MAINTAINER_MODE), поэтому пользователь увидит много описательных предупреждения об отсутствующих или устаревших инструментах и, возможно, предложений о способах установки вместо ошибок command not found или (что хуже) непонятных сообщений от старых версий требуемых инструментов.

  Сопровождающие, заинтересованные в собираемости своих пакетов после выборки CVS даже у пользователей, не имеющих специфических инструментов, могут предоставить вспомогательный сценарий (или улучшенный вариант bootstrap) для корректировки временных меток после команды cvs update или git checkout для предотвращения ненужной пересборки. Если в проекте фиксируются созданные Autotools файлы, а также сгенерированные файлы .info, такой сценарий может иметь вид

  ```
  #!/bin/sh
  # fix-timestamp.sh: prevents useless rebuilds after "cvs update"
  sleep 1
  # aclocal-generated aclocal.m4 depends on locally-installed
  # '.m4' macro files, as well as on 'configure.ac'
  touch aclocal.m4
  sleep 1
  ```

```
# autoconf-generated configure depends on aclocal.m4 and on
# configure.ac
touch configure
# so does autoheader-generated config.h.in
touch config.h.in
# and all the automake-generated Makefile.in files
touch `find . -name Makefile.in -print`
# finally, the makeinfo-generated '.info' files depend on the
# corresponding '.texi' files
touch doc/*.info
```

- В распределенной среде вероятно использование разработчиками разных версий инструментов и повторные сборки, вызванные потерей временных меток приведут к фиктивным изменениям в генерируемых файлах. Есть несколько решений этой проблемы:
  - всем разработчиками следует применять одни версии, чтобы собранные заново файлы были идентичны файлам в CVS (это сложно при использовании разных версий);
  - использовать сценарий корректировки временных меток после (в команде GCC такой сценарий есть);
  - в файле configure.ac использовать макрос AM_MAINTAINER_MODE, отключающий по умолчанию все правила пересборки (27.2. Сценарий missing и режим AM_MAINTAINER_MODE).
- Кроме ложных пересборок могут возникать и обратные ситуации, когда обработка меток CVS показывает, что устаревший файл не является таковым. Предположим, например, что разработчик изменил Makefile.am и заново собрал Makefile.in, а перед фиксацией обоих файлов решил ещё раз изменить Makefile.am (без пересборки Makefile.in). Это изменение делает файл Makefile.in устаревшим. CVS обрабатывает файлы в алфавитном порядке и при использовании другим разработчиком команды cvs update файл Makefile.in окажется новее Makefile.am и разработчик не увидит, что файл Makefile.in реально устарел.

Сгенерированные файлы вне CVS

Одним из способов «примирить» CVS и make является хранение сгенерированных файлов вне CVS, т. е. не включать в число управляемых CVS файлов цели Makefile (производные файлы). Это позволяет разработчику не думать об изменениях в генерируемых файлах и не отслеживать их версии (при условии совместимости). Кроме того, не теряются временные метки, не пропускаются изменения исходных файлов как в рассмотренном выше примере.

Недостаток заключается в том, что репозиторий CVS не является точной копией дистрибутива и пользователи должны устанавливать дополнительные средства разработки (возможно, определённых версий) перед сборкой пакета. Но, в конце концов, задача CVS заключается в управлении версиями, а не распространении файлов.

Возможность разработчиков применять разные версии инструментов может скрывать ошибки при работе в распределенной среде. Разработчики применяют (и тестируют) свои сгенерированные файлы вместо выпускаемых фактически. Подготавливающий архив разработчик может применять версию инструментов, которая создаёт ложные выходные данные (например, не переносимые файлы C), что могли бы заметить другие разработчики при использовании одной версии инструментов.

Сторонние файлы

Ещё один класс составляют файлы, которые распространяются с пакетом, но поддерживаются отдельно. Эти файлы не создают проблем с временными метками и не рассматривались выше. Например, такие инструменты как gettextize и autopoint (изи Gettext) или libtoolize (из Libtool) могут устанавливать или обновлять файлы пакета. Эти файлы, независимо от хранения в CVS, вызывают аналогичные опасения в части несоответствия версий инструментов.

27.2. Сценарий missing и режим AM_MAINTAINER_MODE

missing

Сценарий missing является оболочкой для нескольких инструментов разработки, предназначенной для уведомления пользователей об отсутствии нужных инструментов. Типичными инструментами сопровождающего служат autoconf, automake, bison и т. п. Поскольку создаваемые этими инструментами файлы распространяются с пакетом, эти инструменты не требуются при сборке пользователем и не проверяются в сценарии configure.

Однако при запуске правила пересборки с участием отсутствующих инструментов сценарий missing предупреждает пользователя, указывая, как можно получить нужный инструмент (по крайней мере широко известный, например, makeinfo или bison). Это удобней для пользователя, чем простое применение правил перестройки с выводом сообщений вида «sh: TOOL: command not found». Подобные предупреждения missing выдаёт и при обнаружении у пользователя слишком старых инструментов (эта задача сложнее отсутствия нужных инструментов, поэтому такие предупреждения могут выдаваться не всегда).

Если нужный инструмент установлен, missing пытается запустить его, но не будет продолжать попытки после отказа. Это удобно при разработке, поскольку фиксирует отказы. Однако пользователи, у которых отсутствует или устарел нужный инструмент, получат ошибку, когда вызванное правило пересборки остановит процесс. Такие случаи используют в качестве аргументов сторонники режима AM_MAINTAINER_MODE.

AM_MAINTAINER_MODE

AM_MAINTAINER_MODE позволяет управлять правилами повторной сборки. При AM_MAINTAINER_MODE([enable]) правила включены по умолчанию. Если правила отключены, при наличии AM_MAINTAINER_MODE в configure.ac и запуске команды ./configure && make, программа make никогда не будет пытаться заново создать configure, Makefile.in, вывод Lex или Yacc и т. п. Таким образом, отключается повторная сборка файлов, которые обычно распространяются и пользователю не нужно их обновлять. Пользователь может изменить принятый по умолчанию режим опциями команды configure --enable-maintainer-mode или --disable-maintainer-mode.

AM_MAINTAINER_MODE применяется для того, чтобы пользователей не раздражали потери временных меток (27.1. CVS и генерируемые файлы) или потому, что правила пересборки не созданы и следует явно запускать инструменты. Режим AM_MAINTAINER_MODE также позволяет отключить некоторые пользовательские правила сборки по условию. Некоторые разработчики применяют это для отключений правил, требующих экзотических инструментов.

Несколько лет назад Франсуа Пинар представил аргументы против макроса AM_MAINTAINER_MODE, большая часть которых относилась к безопасности. Удаление зависимостей веде к независимым сборкам - изменение исходных файлов не будет влиять на генерируемые файлы и может создавать путаницу, если не будет замечено. Франсуа отметил, что безопасность не следует оставлять лишь на сопровождающих (как предлагает --enable-maintainer-mode), скорее напротив. Если один пользователь изменил Makefile.am, нужно изменить Makefile.in или выдать предупреждение (для чего Automake использует missing) и самым последним делом является игнорирование изменений без уведомления пользователя (как при отключении пересборки с помощью AM_MAINTAINER_MODE).

Джим Мейринг (Jim Meyering) - создатель AM_MAINTAINER_MODE принял аргументы Франсуа и отказался от AM_MAINTAINER_MODE в своих пакетах. Однако многие продолжают использовать AM_MAINTAINER_MODE, поскольку это помогает при работе с проектами, где файлы хранятся в системе управления версиями и сценария missing не достаточно, если используются разные версии инструментов.

27.3. Почему Automake не поддерживает шаблоны?

Разработчики ленивы и предпочли бы использовать шаблоны в Makefile.am, чтобы не обновлять эти файлы при каждом добавлении, удалении или переименовании файла. Однако есть ряд причин отказа от поддержки шаблонов.

- При использовании CVS (или аналога) разработчики должны помнить о необходимости использовать cvs add или cvs rm. Поэтому обновление Makefile.am быстро становится рефлексом. И наоборот, если приложение не компилируется в результате того, что файл Makefile.am не был обновлён, это напомнит о cvs add.
- Использование шаблонов упрощает распространение ошибок. Например, код, с которым экспериментирует разработчик (скажем, тестовый пример) может неожиданно стать частью дистрибутива.
- При использовании шаблонов легко пропустить по ошибке некоторые файлы. Например, один разработчик создаёт новый файл, использует его во многих метах, но забывает зафиксировать. Другой разработчик может проверить незавершённый проект и даже окажется способным выполнить команду make dist, несмотря на отсутствие файла, но получит сообщение об отсутствующих файлах.
- Шаблоны не переносимы в некоторые отличные от GNU реализации make. Например в NetBSD make не преобразует * в предварительных условиях для цели.
- Наконец, действительно трудно забыть о добавлении файлов в Makefile.am, поскольку такие файлы не будут компилироваться и устанавливаться, что не позволит даже протестировать их.

Однако эти аргументы скорей философские и можно привести серьёзные возражения для них или аргументы в пользу шаблонов. Поэтому следует рассмотреть и техническую сторону вопроса - переносимость. Хотя $(wildcard …) работает в GNU make, это не переносится в другие реализации make. Единственным способом поддержки $(wildcard …) в Automake является преобразование $(wildcard …) в процессе работы automake. Полученные файлы Makefile.in будут переносимыми, поскольку в них указаны все файлы и не используется $(wildcard ...). Однако это требует от разработчиков запускать automake при каждом добавлении, удалении или переименовании файла. По сравнению с редактированием Makefile.am это очень маленький выигрыш. Конечно, проще и быстрее набрать automake; make, чем emacs Makefile.am; make. Но пока никто не удосужился добавить поддержку этого. Кое-кто использует сценарии для генерации списка файлов в Makefile.am или отдельных фрагментах Makefile.

Даже если переносимость не волнует и желание использовать $(wildcard …) сохраняется в силу нацеленности лишь на GNU Make, следует помнить, что во многих местах программе Automake нужно точно знать, какие файлы следует обрабатывать. Если Automake не знает, как преобразовать $(wildcard ...), шаблоны не удастся применить в таких местах. $(wildcard ...) является «чёрным ящиком» по сравнению с подстановкой AC_SUBST, используемой Automake. Можно получать предупреждения об использовании конструкций $(wildcard ...) путём указания флага -Wportability.

27.4. Ограничения для имён. файлов

Automake пытается поддерживать все типы имён. файлов, включая имена с необычными символами и слишком длинные. Однако некоторые ограничения задают операционные системы и применяемые инструменты. Большинство ОС запрещает использовать null-символы в именах файлов, а / служит разделителем каталогов. Также требуется использование кодировки символов в соответствии с настройкой locale. Automake соблюдает такие ограничения. В переносимых пакетах следует соблюдать для имён. требования POSIX, позволяющие использовать в именах буквы ASCII, цифры, а также символы «_», «.» и «-» с разделением компонент имени символом /. Имена компонент не могут начинаться с -. Переносимые имена файлов POSIX не могут включать компоненты размером более 14 байтов, но сегодня достаточно безопасно применять ограничение XOPEN в 255 байтов. POSIX ограничивает размер полного имени (все компоненты) 255 байтами (XOPEN - 1023), но можно ограничивать размер имён. в архивах 99 байтами для предотвращения проблем несовместимости со старыми версиями tar.

При отступлении от этих правил (например, символы не-ASCII или слишком длинные имена), установщики могут столкнуться с проблемами по причинам, не связанным с Automake. Тем не менее, следует помнить и об ограничениях Automake. Эти ограничения нежелательны, но некоторые из низ, по-видимому, присущи базовым инструментам, таким как Autoconf, Make, M4 и командные процессоры. Ограничения делятся на 3 категории - для каталогов установки и сборки, а также для имён. файлов.

Символы

```
newline " # $ ' `
```

не следует применять в именах каталогов установки, например, включать а параметр опции configure --prefix.

Для каталогов сборки эти ограничения сохраняется, а также в их имена не следует включать символы

```
& @ \
```

Например, в полном имени каталога с исходными кодами не должно быть таких символов.

Имена исходных и устанавливаемых файлов (например, main.c) ограничены ещё сильнее и для них следует выполнять правила POSIX/XOPEN. Кроме того, при возможном переносе в отличные от POSIX среды следует избегать имён., отличающихся лишь регистром символов (например, makefile и Makefile). Ограничение 8.3 (DOS) можно забыть.

27.5. Ошибки distclean

Здесь описана диагностика, которая может возникать при работе с командой make distcheck. Как отмечено в параграфе 14.4. Проверка дистрибутива, make distcheck пытается собрать и проверить пакет на предмет ошибок, подобных этой. Команда make distcheck выполняет сборку VPATH для пакета (2.2.6. Параллельная сборка (VPATH)), а затем вызывает make distclean. Оставшиеся в каталоге сборки файлы перечисляются после такой ошибки. Диагностика охватывает два типа ошибок:

- файлы, забытые distclean;
- распространяемые файлы, по ошибке собранные заново.

Первые файлы не распространяются, поэтому для исправления ошибки достаточно пометить их для очистки (13. Очистка). Вторая ошибка не всегда понятна, поэтому рассмотрим её на примере. Предположим, что пакет содержит программу, для которой нужно создать страницу man с помощью help2man. Программа GNU help2man создаёт простые страницы руководств из вывода --help и --version других команд. Поскольку мы не хотим требовать от пользователя установки help2man, созданная страница man включена в дистрибутив, как показано ниже.

```
# Фиктивный файл Makefile.am
bin_PROGRAMS = foo
foo_SOURCES = foo.c
dist_man_MANS = foo.1

foo.1: foo$(EXEEXT)
        help2man --output=foo.1 ./foo$(EXEEXT)
```

Это будет распространять страницу man, однако make distcheck завершится отказом с ошибкой

```
ERROR: files left in build directory after distclean:
./foo.1
```

Почему повторно собирается foo.1? Хотя файл foo.1 распространяется, он зависит от нераспространяемого файла foo$(EXEEXT), который собирает пользователь и поэтому файл всегда кажется новее распространяемого foo.1. Команда make distcheck обнаружила несогласованность в пакете. Планировалось распространять foo.1, чтобы пользователи могли не устанавливать help2man, однако правило всегда перестраивает файл и пользователю требуется help2man. Нужно сделать так, чтобы файл foo.1 не перестраивался пользователями или отказаться от его распространения.

В более общем случае правило заключается в том, что распространяемые файлы не должны зависеть от нераспространяемых. При распространении сгенерированного файла нужно распространять его источники.

Одним из путей распространить в этом примере файл foo.1 является исключение зависимости от foo$(EXEEXT). Например, предположим, что вывод foo --version и foo --help не меняются, пока не изменится foo.c или configure.ac. Тогда можно указать в файле Makefile.am

```
bin_PROGRAMS = foo
foo_SOURCES = foo.c
dist_man_MANS = foo.1

foo.1: foo.c $(top_srcdir)/configure.ac
        $(MAKE) $(AM_MAKEFLAGS) foo$(EXEEXT)
        help2man --output=foo.1 ./foo$(EXEEXT)
```

В результате foo.1 не будет перестраиваться при каждом изменении foo$(EXEEXT). Вызов make обеспечивает актуальность foo$(EXEEXT) до вызова help2man. Другим способом является использование разных каталогов для исполняемых файлов и страниц man с организаций SUBDIRS так, чтобы исполняемые файлы собирались раньше man.

Можно отказаться от распространения foo.1. В этом случае хорошо иметь зависимость foo.1 от foo$(EXEEXT), поскольку оба нужно перестроить. Однако будет невозможно собрать пакет в среде кросс-компиляции, поскольку сборка foo.1 включает вызов foo$(EXEEXT).

Другим случаем, когда возникают такие ошибки, является распространение файлов, созданных инструментами, собранными пакетом. Например,

```
distributed-file: built-tools distributed-sources
        build-command
```

следует заменить на

```
distributed-file: distributed-sources
        $(MAKE) $(AM_MAKEFLAGS) built-tools
        build-command
```

или отказаться от распространения distributed-file, если кросс-компиляция не важна.

Резюмируем рассмотренные в примерах вопросы:

- распространяемые файлы не должны зависеть от нераспространяемых;
- распространяемые файлы следует распространять с их зависимостями;
- если файл предназначен для пересборки пользователем, нет смысла его распространять.

В безнадёжных случаях можно отключить эту проверку, путём установки distcleancheck_listfiles в соответствии с параграфом 14.4. Проверка дистрибутива. Предварительно следует убедиться в понимании причины жалоб make distcheck. Установка distcleancheck_listfiles не исправляет ошибки, а скрывает их. Всегда можно сделать лучше.

27.6. Порядок переменных флагов

В чем разница между AM_CFLAGS, CFLAGS и mumble_CFLAGS? Почему automake выводит CPPFLAGS после AM_CPPFLAGS в строке компиляции? Разве не должно быть наоборот? Сценарий configure добавляет флаги предупреждений в CXXFLAGS. Почему при добавлении флага в конце AM_CXXFLAGS он помещается в начало?

Переменные флагов компиляции

В этом параграфе рассматриваются все приведённые выше вопросы. В примерах рассматривается в основном CPPFLAGS, но реально это относится ко всем флагам, применяемым в Automake - CCASFLAGS, CFLAGS, CPPFLAGS,

CXXFLAGS, FCFLAGS, FFLAGS, GCJFLAGS, LDFLAGS, LFLAGS, LIBTOOLFLAGS, OBJCFLAGS, OBJCXXFLAGS, RFLAGS, UPCFLAGS, YFLAGS.

Переменные CPPFLAGS, AM_CPPFLAGS и mumble_CPPFLAGS могут служить для передачи флагов препроцессору C (фактически и другим языкам, таким как C++ или Fortran с предварительной обработкой). Переменная CPPFLAGS является пользовательской (3.6. Пользовательские переменные), AM_CPPFLAGS относится к Automake, а mumble_CPPFLAGS - к цели mumble (переменная на уровне цели, см. 8.4. Переменные для программ и библиотек).

Automake всегда использует две переменные при компиляции исходных файлов C. При компиляции объектного файла для цели mumble первой переменной будет mumble_CPPFLAGS (если она задана) или AM_CPPFLAGS. Второй переменной всегда служит CPPFLAGS. В приведённом ниже примере

```
bin_PROGRAMS = foo bar
foo_SOURCES = xyz.c
bar_SOURCES = main.c
foo_CPPFLAGS = -DFOO
AM_CPPFLAGS = -DBAZ
```

xyz.o будет компилироваться с флагами $(foo_CPPFLAGS) $(CPPFLAGS) (поскольку xyz.o является частью foo), а main.o - с $(AM_CPPFLAGS) $(CPPFLAGS) (поскольку нет переменной для цели bar).

Разница между mumble_CPPFLAGS и AM_CPPFLAGS достаточно очевидна, поэтому сосредоточимся на CPPFLAGS. Это пользовательская переменная, т. е. она служит пользователю для управления компиляцией пакета. Переменная, наряду с другими, указана в конце вывода команды configure --help.

Например, для добавления /home/my/usr/include в путь поиска компилятора C можно указать

```
./configure CPPFLAGS='-I /home/my/usr/include'
```

и этот флаг будет распространён в правила компиляции всех Makefile.

Пользовательские переменные нередко переопределяются в процессе работы make. Многие установщики делают это с переменной prefix, но может быть полезно менять и для флаги компилятора. Например, при отладке проекта C++ может потребоваться отключить оптимизацию в определённом объектном файле, как показано ниже

```
rm file.o
make CXXFLAGS=-O0 file.o
make
```

Причина появления $(CPPFLAGS) после $(AM_CPPFLAGS) или $(mumble_CPPFLAGS) в команде компиляции заключается в том, что последнее слово должно оставаться за пользователем. Это может обрести больше смысла, если посмотреть на флаг CXXFLAGS=-O0 в примере выше, который должен переопределить AM_CXXFLAGS или mumble_CXXFLAGS (а также заменить прежнее значение CXXFLAGS).

Никогда не следует переопределять пользовательские переменные (такие как CPPFLAGS) в Makefile.am. Для обнаружения таких ошибок служит команда automake -Woverride. Даже команды вида

```
CPPFLAGS = -DDATADIR=\"$(datadir)\" @CPPFLAGS@
```

являются ошибочными. Хотя эта команда сохраняет заданное configure значение CPPFLAGS, определение DATADIR потеряется, если пользователь переопределит CPPFLAGS из команды make. Определение

```
AM_CPPFLAGS = -DDATADIR=\"$(datadir)\"
```

показывает все, что нужно знать об использовании флагов на уровне цели.

Не следует добавлять опции к пользовательским переменным в configure по той же причине. Иногда нужно изменить эти переменные для выполнения теста , но впоследствии значения следует вернуть. И напротив, нормально менять переменные AM_ внутри configure, при использовании для них AC_SUBST, но это требуется достаточно редко, если действительно не нужно менять принятые по умолчанию определения переменных AM_ во всех Makefile.

Рекомендуется задавать дополнительные флаги в отдельных переменных. Например, можно создать макрос Autoconf для создания набора опций предупреждений для компилятора C и использовать AC_SUBST для подстановки в WARNINGCFLAGS. Можно также создать макрос Autoconf для определения флагов компилятора и компоновщика, которые следует применять при компоновке с библиотекой libfoo, и использовать AC_SUBST для подстановки в LIBFOOCFLAGS и LIBFOOLDFLAGS. Затем Makefile.am может использовать эти флаги, как показано ниже.

```
AM_CFLAGS = $(WARNINGCFLAGS)
bin_PROGRAMS = prog1 prog2
prog1_SOURCES = ...
prog2_SOURCES = ...
prog2_CFLAGS = $(LIBFOOCFLAGS) $(AM_CFLAGS)
prog2_LDFLAGS = $(LIBFOOLDFLAGS)
```

В этом примере обе программы компилируются с флагами, подставленными в $(WARNINGCFLAGS), а prog2 дополнительно получает флаги, нужные для компоновки с libfoo. Отметим, что указание AM_CFLAGS в переменной CFLAGS на уровне цели является гарантией применения AM_CFLAGS к каждой цели в Makefile.in.

Такое использование переменных обеспечивает полный контроль за порядком флагов. Например, если в переменной $(WARNINGCFLAGS) есть флаг, который нужно отменить для конкретной цели, можно использовать что-то вроде prog1_CFLAGS = $(AM_CFLAGS) -no-flag. Если бы все флаги были принудительно добавлены в CFLAGS, не было бы возможности отменить один флаг. Это ещё одна причина сохранять пользовательские переменные для пользователя.

Переменная из примера LIBFOO_LDFLAGS (с подчёркиванием) не была указана, поскольку это заставило бы Automake думать, что это переменная на уровне цели (как mumble_LDFLAGS) для некой необъявленной цели LIBFOO.

Другие переменные

В Automake есть другие переменные, следующие похожим принципам для пользовательских опций. Например, правила Texinfo (11.1. Texinfo) используют MAKEINFOFLAGS и AM_MAKEINFOFLAGS, тесты DejaGnu (15.5. Тесты DejaGnu) - RUNTESTDEFAULTFLAGS и AM_RUNTESTDEFAULTFLAGS, правила tags и ctags (18.1. Взаимодействие с etags) - ETAGSFLAGS, AM_ETAGSFLAGS, CTAGSFLAGS и AM_CTAGSFLAGS, правила Java (8.16. Компиляция файлов Java с помощью gcj) - JAVACFLAGS и AM_JAVACFLAGS. Ни одно из этих правил (пока) не поддерживает флагов для цели.

В некоторой степени даже AM_MAKEFLAGS (7.1. Рекурсия каталогов) подчиняется этой схеме именования. Незначительное отличие состоит в том, что MAKEFLAGS передаётся субкоманде make неявно самой командой make.

ARFLAGS (8.2. Сборка библиотек) обычно задаётся Automake и не имеет AM_ или аналога на уровне цели.

Не следует считать, что наличие переменной для каждой цели предполагает существование AM_ или пользовательской переменной. Например, переменная уровня цели mumble_LDADD переопределяет переменную LDADD уровня Makefile (не является пользовательской) и mumble_LIBADD существует лишь на уровне цели (8.4. Переменные для программ и библиотек).

27.7. Переименование объектных файлов

Это происходит при использовании флагов компиляции на уровне цели. Объектные файлы переименовываются на случай совпадения имён. с файлами тех же источников, скомпилированныи с другими флагами. Рассмотрим пример

```
bin_PROGRAMS = true false
true_SOURCES = generic.c
true_CPPFLAGS = -DEXIT_CODE=0
false_SOURCES = generic.c
false_CPPFLAGS = -DEXIT_CODE=1
```

Очевидно, что две программы собираются из одного источника, но было бы плохо использовать один объект, потому что generic.o нельзя собрать одновременно с -DEXIT_CODE=0 и -DEXIT_CODE=1. Поэтому automake создаёт правила для двух разных объектов - true-generic.o и false-generic.o.

На деле automake не проверяет использование общих источников при решении вопроса о переименовании объектов. Программа просто переименовывает все объекты цели, если видит использование флагов компиляции на уровне цели. Использование общих объектных файлов нормально, если не применяются фланги компиляции на уровне цели. Например, true и false используют один объект version.o в следующем примере.

```
AM_CPPFLAGS = -DVERSION=1.0
bin_PROGRAMS = true false
true_SOURCES = true.c version.c
false_SOURCES = false.c version.c
```

Отметим, что на переименование объектов влияет также _SHORTNAME (8.4. Переменные для программ и библиотек).

27.8. Флаги компиляции на уровне объекта

Automake поддерживает флаги компиляции на уровне программ и библиотек (8.4. Переменные для программ и библиотек, 27.6. Порядок переменных флагов). Это позволяет задать флаги компиляции для всех файлов цели. Например,

```
bin_PROGRAMS = foo
foo_SOURCES = foo.c foo.h bar.c bar.h main.c
foo_CFLAGS = -some -flags
```

Объекты foo-foo.o, foo-bar.o и foo-main.o компилируются с -some -flags. Отметим, что foo_CFLAGS даёт флаги для использования при компиляции всех источников C программы foo, ничего не задавая отдельно для foo.c или foo-foo.o.

Если foo.c нужно скомпилировать в foo.o с особыми флагами, ситуация усложняется. Обычно флаги на уровне программы здесь не применимы напрямую. Предполагается что-то вроде флагов на уровне объекта, которые будут применяться лишь для foo-foo.o. Automake не поддерживает этого, однако можно легко обмануть программу, используя библиотеку с единственным объектом и своими флагами компиляции.

```
bin_PROGRAMS = foo
foo_SOURCES = bar.c bar.h main.c
foo_CFLAGS = -some -flags
foo_LDADD = libfoo.a
noinst_LIBRARIES = libfoo.a
libfoo_a_SOURCES = foo.c foo.h
libfoo_a_CFLAGS = -some -other -flags
```

Здесь foo-bar.o и foo-main.o компилируются с -some -flags, а libfoo_a-foo.o с -some -other -flags. Затем все объекты можно скомпоновать в foo. Этого можно добиться и с помощью вспомогательных библиотек Libtool, например, задав noinst_LTLIBRARIES = libfoo.la (8.3.5. Вспомогательные библиотеки Libtool).

Другая заманчивая идея заключается в реализации флагов на уровне объекта путём переопределения флагов компиляции, которые automake будет создавать для этих файлов. Automake не будет задавать правило для цели, которая определена, поэтому можно думать о задании правила foo-foo.o: foo.c. Делать это не рекомендуется по причине вероятных ошибок. Например, если добавить такое правило в первый пример выше, это создаст проблемы при удалении foo_CFLAGS (foo.c в этом случае будет компилироваться как foo.o вместо foo-foo.o, см. 27.7. Переименование объектных файлов). Для поддержки отслеживания зависимостей, двух расширений .o и .obj, а также всех других переменных флагов, вовлечённых в компиляцию, потребуется в конечном итоге изменить копию правила, выведенного ранее automake для этого файла. Если новый выпуск Automake создаст иное правило, снова придётся обновлять копию вручную.

27.9. Инструменты обработки с объёмным выводом

В этом параграфе рассматривается использование make с инструментами, создающими много выходных файлов. Это не привязано к Automake и может применяться в обычных файлах Makefile. Предположим, что имеется программа foo, которая читает файл data.foo и создаёт файлы data.c и data.h. нужно написать правило Makefile, фиксирующую зависимость «один к двум». Естественное правило некорректно

```
# Это является ошибкой
data.c data.h: data.foo
        foo data.foo
```

Приведённое правило говорит о том, что каждый из файлов data.c и data.h зависит от data.foo и каждый можно собрать запуском foo data.foo. Иными словами, это эквивалентно

```
# Этого не нужно.
data.c: data.foo
        foo data.foo
```

```
data.h: data.foo
        foo data.foo
```

Это означает, что программа foo может быть запущена дважды. Обычно она дважды не запускается, поскольку реализация make достаточно эффективна для проверки наличия второго файла после сборки первого и увидит имеющийся файл. Однако в нескольких случаях повторных запуск произойдёт. в любом случае.

- Наиболее тревожным является случай параллельной работы make. Если файлы data.c и data.h собираются параллельно, одновременно будут выполняться две команды foo data.foo, что нехорошо.
- Другой случай возникает, когда зависимость (здесь data.foo) является фиктивной целью (или зависит от такой).

Решение для параллельной работы make, но не для фиктивных зависимостей показано ниже.

```
data.c data.h: data.foo
        foo data.foo
data.h: data.c
```

Эти правила эквивалентны

```
data.c: data.foo
        foo data.foo
data.h: data.foo data.c
        foo data.foo
```

Поэтому в параллельном режиме make должны быть последовательно собраны data.c и data.h и будет обнаружено, что второй запуск не требуется после завершения первого.

Предложенного варианта достаточно в большинстве случаев, однако он слабо расширяется для большого числа файлов (требуется общая упорядоченность в соответствии с зависимостями) и нужно искать другое решение.

```
# остаётся ещё проблема с этим.
data.c: data.foo
        foo data.foo
data.h: data.c
```

Идея состоит в запуске foo data.foo только при необходимости обновления data.c, но далее будет указано, что data.h зависит от data.c. Т. е., если требуется data.h, а файл data.foo устарел, зависимость от data.c вызовет сборку. Это почти идеально, но предположим, что созданы файлы data.h и data.c, а затем data.h удалён. Тогда команда make data.h не соберёт заново data.h. Приведённое выше правило говорит лишь, что файл data.c должен быть актуален по отношению к data.foo и это условие выполняется. Нужно правило, вызывающее перестройку при отсутствии data.h.

```
data.c: data.foo
        foo data.foo
data.h: data.c
## Восстановление после удаления $@
        @if test -f $@; then :; else \
          rm -f data.c; \
          $(MAKE) $(AM_MAKEFLAGS) data.c; \
        fi
```

Приведённую схему можно расширить для обработки большего числа файлов на входе и выходе. Один из выходов выбирается в качестве свидетельства успешного завершения команды - он зависит от всех входов, а остальные выходы зависят от него. Например, если foo вдобавок читает data.bar, а также создаёт data.w и data.x, можно задать

```
data.c: data.foo data.bar
        foo data.foo data.bar
data.h data.w data.x: data.c
## Восстановление после удаления $@
        @if test -f $@; then :; else \
          rm -f data.c; \
          $(MAKE) $(AM_MAKEFLAGS) data.c; \
        fi
```

Однако в этом случае возникает три мелких проблемы. Одна связана с порядком временных меток data.h, data.w, data.x и data.c, другая - с состязанием в восстановлении при параллельном запуске нескольких экземпляров make, а третья - с правилом рекурсии, прерывающим make -n в случае работы с GNU make (и некоторыми другими реализациями), поскольку это может удалять data.h без необходимости.

Разберёмся с первой проблемой - foo выводит 4 файла, но порядок их создания неизвестен. Предположим, что data.h создаётся раньше data.c. Тогда возникает странная ситуация и при следующем запуске make файл data.h будет старше data.c, что вызовет второе правил и оболочка будет выполнять команду if...fi, но в действительности будет выполняться ветвь then, т. е. ничего не произойдёт. Иными словами, поскольку выбранный свидетель не является первым файлом, созданным командой foo, make запустит оболочку, которая ничего не сделает. Простым решением является исправление временных меток, как показано ниже.

```
data.c: data.foo data.bar
        foo data.foo data.bar
data.h data.w data.x: data.c
        @if test -f $@; then \
          touch $@; \
        else \
## Восстановление после удаления $@
          rm -f data.c; \
          $(MAKE) $(AM_MAKEFLAGS) data.c; \
        fi
```

Другим решением является использование в качестве свидетеля выделенного файла, а не вывода foo.

```
data.stamp: data.foo data.bar
        @rm -f data.tmp
        @touch data.tmp
        foo data.foo data.bar
        @mv -f data.tmp $@
data.c data.h data.w data.x: data.stamp
## Восстановление после удаления $@
        @if test -f $@; then :; else \
          rm -f data.stamp; \
          $(MAKE) $(AM_MAKEFLAGS) data.stamp; \
        fi
```

Файл data.tmp создаётся до запуска foo, поэтому его временная метка будет старше всех выходных файлов foo. Этот файл переименовывается в data.stamp после завершения работы foo, поскольку мы не хотим обновлять data.stamp в случае отказа foo.

В этом решении ещё сохраняется вторая проблема — состязание в правиле восстановления. Если после успешной сборки пользователь удалит файлы data.c и data.h, а затем запустит make j, программа make может начать восстановление обоих файлов в параллель. Если два экземпляра правила выполнят $(MAKE) $(AM_MAKEFLAGS) data.stamp одновременно, сборка приведёт к отказу (например, 2 правила создадут data.tmp, но лишь одно сможет переименовать его). Правда, такие ситуации не возникают при обычной сборке и бывают лишь в случаях повреждения дерева сборки. Здесь data.c и data.h были явно удалены без удаления data.stamp и других выходных файлов. Команды make clean; make всегда будут восстанавливать из таких состояний (даже при параллельной сборке) поэтому можно подумать, что правило восстановления предназначено исключительно для того, чтобы помочь при непараллельном использовании make. Исправление требует иного механизма блокировки для гарантированного применения лишь одного правила восстановления, повторно собирающего data.stamp. Можно представить что-то вроде

```
data.c data.h data.w data.x: data.stamp
## Восстановление после удаления $@
        @if test -f $@; then :; else \
          trap 'rm -rf data.lock data.stamp' 1 2 13 15; \
## mkdir является переносимой проверкой
          if mkdir data.lock 2>/dev/null; then \
## Код, выполняемый первым процессом.
            rm -f data.stamp; \
            $(MAKE) $(AM_MAKEFLAGS) data.stamp; \
            result=$$?; rm -rf data.lock; exit $$result; \
          else \
## Код, выполняемый следующими процессами.
## Ожидание завершения первого процесса.
            while test -d data.lock; do sleep 1; done; \
## Успешно лишь при успехе первого процесса.
            test -f data.stamp; \
          fi; \
        fi
```

Использование выделенного свидетеля (data.stamp) очень удобно, когда список выходных файлов заранее неизвестен. В качестве иллюстрации рассмотрим приведённые ниже правила для компиляции множества файлов *.el в файлы *.elc одной командой. И неважно, как задаётся цель ELFILES (она должна быть непустой в соответствии с POSIX).

```
ELFILES = one.el two.el three.el ...
ELCFILES = $(ELFILES:=c)

elc-stamp: $(ELFILES)
        @rm -f elc-temp
        @touch elc-temp
        $(elisp_comp) $(ELFILES)
        @mv -f elc-temp $@

$(ELCFILES): elc-stamp
        @if test -f $@; then :; else \
## Восстановление после удаления $@
          trap 'rm -rf elc-lock elc-stamp' 1 2 13 15; \
          if mkdir elc-lock 2>/dev/null; then \
## Код, выполняемый первым процессом.
            rm -f elc-stamp; \
            $(MAKE) $(AM_MAKEFLAGS) elc-stamp; \
            rmdir elc-lock; \
          else \
## Код, выполняемый следующими процессами.
## Ожидание завершения первого процесса.
            while test -d elc-lock; do sleep 1; done; \
## Успешно лишь при успехе первого процесса.
            test -f elc-stamp; exit $$?; \
          fi; \
        fi
```

Во всех приведённых решениях сохраняется третья проблема - нарушается обещание не вносить реальных изменений в дерево по команде make -n. Для решений без файлов блокировки можно разделить правила восстановления на две команды, одна из которых выполняет все, кроме рекурсии, а другая рекурсивно вызывает $(MAKE). Решения с блокировкой могут воздействовать на содержимое переменной MAKEFLAGS, но переносимость этого не очевидна. Например,

```
ELFILES = one.el two.el three.el ...
ELCFILES = $(ELFILES:=c)

elc-stamp: $(ELFILES)
        @rm -f elc-temp
        @touch elc-temp
        $(elisp_comp) $(ELFILES)
        @mv -f elc-temp $@

$(ELCFILES): elc-stamp
## Восстановление после удаления $@
        @dry=; for f in x $$MAKEFLAGS; do \
          case $$f in \
            *=*|--*);; \
            *n*) dry=:;; \
          esac; \
        done; \
        if test -f $@; then :; else \
          $$dry trap 'rm -rf elc-lock elc-stamp' 1 2 13 15; \
          if $$dry mkdir elc-lock 2>/dev/null; then \
## Код, выполняемый первым процессом.
            $$dry rm -f elc-stamp; \
            $(MAKE) $(AM_MAKEFLAGS) elc-stamp; \
            $$dry rmdir elc-lock; \
          else \
## Код, выполняемый следующими процессами.
## Ожидание завершения первого процесса.
            while test -d elc-lock && test -z "$$dry"; do \
              sleep 1; \
            done; \
```

```
    ## Успешно лишь при успехе первого процесса.
               $$dry test -f elc-stamp; exit $$?; \
             fi; \
           fi
```

Для полноты следует отметить, что GNU make позволяет задавать правила с множеством выходных файлов, посредством шаблонных правил. Такие правила не рассматриваются здесь по причине непереносимости, но могут быть удобны в пакетах, предполагающих GNU make.

27.10. Установка в жёстко заданные места

Почему при попытке применить приведённое правило для установки файла конфигурации make distcheck даёт отказ?

```
# Не делайте так.
install-data-local:
        $(INSTALL_DATA) $(srcdir)/afile $(DESTDIR)/etc/afile
```

Пакету нужно поместить данные в установочный каталог другого пакета. Этот каталог легко определить в configure, но при установке файлов туда возникает отказ make distcheck. Что делать?

У этих вариантов установки есть общая черта - отказ make distcheck обусловлен попыткой установки в жёстко заданные места. Во втором случае путь на задан жёстко в пакете, но является жёстко заданным в системе (или в инструменте, предоставляющем путь). Пока путь не использует ни одну из стандартных переменных ($(prefix), $(bindir), $(datadir) и т. п.), эффект будет сохраняться и пользователь не сможет установить файл.

У обычного (не root) пользователя, желающего установить пакет, обычно нет права помещать что-либо в /usr или /usr/local. В результате применяется что-то вроде ./configure --prefix ~/usr для установки в каталог ~/usr. Если пакет пытается установить что-то по заданному жёстко пути (например, /etc/afile), независимо от значения --prefix установка приведёт к отказу. Команда make distcheck выполняет установку с --prefix и поэтому тоже завершается отказом.

Ниже приведено несколько простых решений. Пример install-data-local для установки /etc/afile лучше заменить на

```
sysconf_DATA = afile
```

По умолчанию sysconfdir имеет значение $(prefix)/etc, поскольку этого требуют стандарты GNU. Когда такой пакет устанавливается в файловую систему FHS, установщик задаёт --sysconfdir=/etc. Сопровождающему не нужно заботиться о таких правилах сайта - просто используется стандартная переменная каталога для установки файлов, которую установщик легко может переопределить в соответствии с правилами сайта.

Установка файлов для использования другими пакетами несколько сложнее. Рассмотрим пример установки общей библиотеки, которая является модулем расширения Python. Если спросить Python, куда устанавливать библиотеку, ответ будет иметь вид

```
% python -c 'from distutils import sysconfig;
             print sysconfig.get_python_lib(1,0)'
/usr/lib/python2.5/site-packages
```

Если действительно указать этот абсолютный путь для установки общей библиотеки, пользователь (не root) не сможет установить пакет и distcheck завершится отказом. Рассмотрим другой вариант. Функция sysconfig.get_python_lib() действительно принимает третий аргумент, заменяющий префикс установки Python.

```
% python -c 'from distutils import sysconfig;
             print sysconfig.get_python_lib(1,0,"${exec_prefix}")'
${exec_prefix}/lib/python2.5/site-packages
```

Можно использовать этот новый путь. В этом случае

- пользователь root может установить пакет с тем же --prefix, что у Python (поведение предыдущей попытки);
- остальные пользователи также могут установить пакет и у них будет модуль расширения в каталоге, который Python не смотрит, но это можно решить через переменные окружения (если установлены сценарии, использующие эту библиотеку, можно просто сказать Python, куда смотреть, в начале сценария).

Макрос AM_PATH_PYTHON использует похожие команды для заданий $(pythondir) и $(pyexecdir) (10.5. Python).

Конечно, не все инструменты настолько развиты, как Python в части подстановки PREFIX. Поэтому другая стратегия заключается в указании части каталога установки, которая должна сохраняться. Ниже показано, как макрос AM_PATH_LISPDIR (10.1. Emacs Lisp) находит $(lispdir).

```
$EMACS -batch -Q -eval '(while load-path
  (princ (concat (car load-path) "\n"))
  (setq load-path (cdr load-path)))' >conftest.out
lispdir=`sed -n
  -e 's,/$,,'
  -e '/.*\/lib\/x*emacs\/site-lisp$/{
        s,.*/lib/\(x*emacs/site-lisp\)$,${libdir}/\1,;p;q;
      }'
  -e '/.*\/share\/x*emacs\/site-lisp$/{
        s,.*/share/\(x*emacs/site-lisp\),${datarootdir}/\1,;p;q;
      }'
  conftest.out`
```

Просто выбирается первый каталог вида */lib/*emacs/site-lisp или */share/*emacs/site-lisp в пути поиска emacs и подставляется ${libdir} или ${datadir}. Случай emacs выглядит сложным, поскольку в нем обрабатывается список и предлагается две возможных схемы. В остальном все просто, а преимущества для обычных пользователей компенсируют дополнительный вызов sed.

27.11. Отладка правил Make

Правила и деревья зависимостей, создаваемые automake, могут быть достаточно сложными и создавать для разработчиков проблемы, когда что-то пошло не так. В дополнение к предоставляемым командой make опций отладки имеется ещё ряд способов эффективной отладки файлов, созданных automake.

- Если в пакете включён сокращённый вывод с помощью правил тишины (21.3. Как Automake может «заглушить» make), можно использовать make V=1 для просмотра выполняемых команд.

- Команда make -n может показать, что было сделано без реального выполнения этого. Однако следует помнить, что не будут учитываться команды с префиксом +, а при использовании GNU make - команды со строками $(MAKE) и ${MAKE}. Как правило, это полезно для просмотра действий рекурсивных правил, но это означает, что в ваших правилах не следует смешивать рекурсию с действиями, которые меняют файлы[1]. Кроме того, отметим, что GNU make обновит предварительные условия для Makefile даже при наличии -n.
- Команда make SHELL="/bin/bash -vx" может помочь при отладке сложных правил.
- Команда echo 'print: ; @echo "$(VAR)"' | make -f Makefile -f - print может помочь в проверке преобразованных значений переменных. Может потребоваться указание отличной от print цели, если та уже применяется или есть файл с таким именем.
- На странице http://bashdb.sourceforge.net/remake/ описана изменённая команда GNU make, называемая remake, которая копирует связанные с GNU make файлы Makefiles и позволяет отслеживать их выполнение, проверять переменные и вызывать правила в интерактивном режиме, подобно отладчику.

27.12. Информирование об ошибках

Любая нетривиальная программа имеет ошибки и Automake не является исключением. Разработчики не могу обещать исправить и даже признать ошибку, но прежде всего о ней нужно узнать. Для этого пользователям при обнаружении ошибок следует информировать о них разработчиков. Перед отправкой сообщения об ошибке разумно узнать, не указал ли кто-нибудь на неё раньше. Для этого можно посмотреть GNU Bug Tracker и почтовые конференции bug-automake. Ранее для информировании об ошибках использовалась база данных Gnats и сообщения о некоторых ошибках уже могут быть в ней (не следует использовать эту базу для новых сообщений).

Если ошибка ещё не известна, следует сообщить о ней. Важно сообщать об ошибках так, чтобы это было полезно и информативно. Для этого следует ознакомиться с документами How to Report Bugs Effectively и How to Ask Questions the Smart Way, что поможет вам и разработчикам сэкономить время. Для сообщений об ошибках, запроса новых функций и других предложений, следует использовать адрес bug-automake@gnu.org. Это будет создавать новую запись в системе отслеживания ошибок. В сообщении следует указать используемые версии Autoconf и Automake, а в идеале приложить файлы Makefile.am и configure.ac для воспроизведения проблемы. При отказах тестов следует предоставить файл test-suite.log.

Приложение A. Копирование этого руководства

A.1. GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

[1] *В правилах Automake dist и distcheck была ошибка и они создавали каталоги даже с опцией -n (исправлена в Automake 1.11.*

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “publisher” means any person or entity that distributes copies of the Document to the public.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all of these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (C) YEAR YOUR NAME.
>
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled ``GNU Free Documentation License''.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru