# A guide to robotpkg

Anthony Mallet — anthony.mallet@laas.fr

November 13, 2026

Copyright ©2006-2011,2013 LAAS/CNRS.
Copyright ©1997-2010 The NetBSD Foundation, Inc.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE NETBSD FOUNDATION, INC. AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

# Contents


**1 Introduction** **1**
- 1.1 What is robotpkg? . . . 1
- 1.2 Why robotpkg? . . . 2
- 1.3 Supported platforms . . . 3
- 1.4 Overview . . . 4
- 1.5 Terminology . . . 4
- 1.6 Roles involved in robotpkg . . . 5
- 1.7 Typography . . . 5

**2 The robotpkg user's guide** **7**
- 2.1 Where to get robotpkg and how to keep it up-to-date . . . 7
  - 2.1.1 Getting the binary bootstrap kit . . . 8
  - 2.1.2 Getting robotpkg for source compilation . . . 8
  - 2.1.3 Keeping robotpkg up-to-date . . . 9
- 2.2 Bootstrapping robotpkg . . . 9
  - 2.2.1 Bootstrapping via the binary kit . . . 9
  - 2.2.2 Bootstrapping from source . . . 9
- 2.3 Using robotpkg . . . 11
  - 2.3.1 Building packages from source . . . 11
  - 2.3.2 Building packages from a repository checkout . . . 13
  - 2.3.3 Installing binary packages . . . 14
  - 2.3.4 Removing packages . . . 15
  - 2.3.5 Getting information about installed packages . . . 15
  - 2.3.6 Other administrative functions . . . 15
  - 2.3.7 Available `make` targets . . . 15
- 2.4 Configuring robotpkg . . . 17
  - 2.4.1 Selecting build options . . . 17
  - 2.4.2 Selecting build alternatives . . . 18
  - 2.4.3 Defining collections of packages . . . 19
  - 2.4.4 Package specific configuration variables . . . 20
  - 2.4.5 General configuration variables . . . 20
  - 2.4.6 Variables affecting the build process . . . 21
  - 2.4.7 Additional flags to the compiler . . . 22

2.5 Creating binary packages for everything . . . . . 22
2.5.1 Initial setup . . . . . 22
2.5.2 Running bulk builds . . . . . 23
2.5.3 Generating pretty reports . . . . . 24
2.5.4 Automated bulk builds . . . . . 24

**3 The robotpkg developer's guide** **25**
3.1 Package files, directories and contents . . . . . 25
3.1.1 Makefile . . . . . 25
3.1.2 distinfo . . . . . 28
3.1.3 PLIST . . . . . 28
3.1.4 patches/* . . . . . 29
3.2 General operation . . . . . 29
3.2.1 Adding build options to a package . . . . . 29
3.2.2 Customizing the PLIST . . . . . 31
3.2.3 Customizing the semi-automatic PLIST generation . . 33
3.2.4 Incrementing versions when fixing an existing package 33
3.2.5 Substituting variable text in the package files . . . . . 34
3.3 The build phase . . . . . 34

# 1

# Introduction

## 1.1 What is robotpkg?

The robotics research community has always been developing a lot of software, in order to illustrate theoretical concepts and validate algorithms on board real robots. A great amount of this software was made freely available to the community, especially for Unix-based systems, and is usually available in form of the source code. Therefore, before such software can be used, it needs to be configured to the local system, compiled and installed. This is exactly what The Robotics Packages Collection (robotpkg) does. robotpkg also has some basic commands to handle binary packages, so that not every user has to build the packages for himself, which is a time-costly, cumbersome and error-prone task.

The robotpkg project was initiated in the Laboratory for Analysis and Architecture of Systems (CNRS/LAAS), France. The motivation was, on the one hand, to ease the software maintenance tasks for the robots that are used there. On the other hand, roboticists at CNRS/LAAS have always fostered an open-source development model for the software they were developing. In order to help people working with the laboratory to get the LAAS software running outside the laboratory, a package management system was necessary.

Although robotpkg was an innovative project in the robotics community (it started in 2006), a lot of general-purpose software packages management systems were readily available at this time for a great variety of Unix-based systems. The main requirements that we wanted robotpkg to fullfill were listed and the best existing package management system was chosen as a starting point. The biggest requirement was the capacity of the system to adapt to the nature of the robotic software, being available mostly in form

of source code only (no binary packages), with unfrequent stable releases. robotpkg had thus to deal mostly with source code and automate the compilation of the packages. The system chosen as a starting point was The NetBSD Packages Collection (pkgsrc). robotpkg can be considered as a fork of this project and it is still very similar to pkgsrc in many points, although some simplifications were made in order to provide a tool geared toward people that are not computer scientists but roboticists.

Due to its origins, robotpkg provides many packages developed at LAAS. It is however not limited to such packages and contains, in fact, quite some other software useful to roboticists. Of course, robotpkg is not meant to be a general purpose packaging system (although there would be no technical restriction to this) and will never contain widely available packages that can be found on any modern Unix distribution. Yet, robotpkg currently contains roughly one hundred and fifty packages, including:

- architecture/genom - The LAAS Generator of Robotic Components
- architecture/openrtm - The robotic distributed middleware from AIST, Japan
- middleware/yarp - The “other”, yet famous, robot platform
- ...just to name a few.

## 1.2 Why robotpkg?

robotpkg provides the following key features:

- Easy building of software from source as well as the creation and installation of binary packages. The source and latest patches are retrieved from a master download site, checksum verified, then built on your system.
- All packages are installed in a consistent directory tree, including binaries, libraries, man pages and other documentation.
- Package dependencies, including when performing package updates, are handled automatically.
- The installation prefix, acceptable software licenses and build-time options for a large number of packages are all set in a simple, central configuration file.
- The entire framework source (not including the package distribution files themselves) is freely available under a BSD license, so you may extend and adapt robotpkg to your needs, like robotpkg was adapted from pkgsrc.

One question often asked by people is “why was robotpkg forked from pkgsrc instead of integrating the packages into pkgsrc?”. This is indeed a very good question and the following paragraphs try to answer it.

First, robotpkg is not meant to be a replacement for the system's package management tool (it does not superseeds pkgsrc, dpkg, macports etc.). The goal is to package software that is not widely available on a platform, and which is mostly "lab software" (generally of lesser quality than widely available software). Those packages change (a lot) more often, and more drastically. Thus, robotpkg is a little bit closer to a "development" tool than pkgsrc. Other “system packages” are correctly handled by a number of packaging tools, and there is no need for a new tool.

Currently, pkgsrc mixes both infrastructure and packages descriptions themselves. For someone working on e.g. Linux, checking-out the whole pkgsrc tree would be cumbersome: it would be redundant with the base Linux package system, plus it would be difficult to isolate the specific robotic packages from the rest (the rest usually being available in the base system). robotpkg currently suffers from the same symptom: this may change in the future if the need for several package repositories becomes blatant.

robotpkg provides a number of features not available in pkgsrc (and probably not really useful to pkgsrc either). The most important feature is to be able to detect “system packages”, that are considered as "external software not in robotpkg but usually available on a unix system". pkgsrc has a similar system but much more limited – to a few base packages only. This is so because pkgsrc is a full-fledged package system. Thus, it aims at being self contained, while robotpkg does not.

Finally, there are a number of additions/changes to the pkgsrc infrastructure that correspond to legitimate users requests and the specifc workflow in which robotpkg is used. For instance, robotpkg provides the possibility to generate an archive of a package from a specific tag in a source repository “on the fly” or just bypass the archive generation and work directly from the source repository to install the software. This later workflow is not encouraged, but it is convenient to quickly test a -current version of some software to see if it causes any problem. Those features could be ported back to pkgsrc if the pkgsrc team would find them useful. In the meantime, robotpkg provides a good testbed for them.

Still, robotpkg directly uses many of the pkgsrc tools unchanged and the binary packages are fully compatible.

## 1.3 Supported platforms

robotpkg consists of a source distribution. After retrieving the required source, you can be up and running with robotpkg in just minutes!

robotpkg does not have much requirements by itself and it can work on

a wide variety of systems as long as they provide a GNU-make utility, a working C-compiler and a small, reasonably standard subset of Unix commands (like sed, awk, find, grep ...). However, individual packages might have their specific requirements. The following platforms have been reported to be supported reasonably well:

| Platform | Version |
|---|---|
| Fedora | 25 or above |
| Ubuntu | 12.04 or above |
| Debian | 7 or above |
| NetBSD | 6 or above |
| Darwin | Partial support - infrastructure works, individual packages may not |

Any other Unix-like platform should usually work.

## 1.4 Overview

This document is divided into three parts. The first one, The robotpkg user's guide, describes how one can use one of the packages in the Robotics Package Collection, either by installing a precompiled binary package, or by building one's own copy using robotpkg. The second part, The robotpkg developer's guide, explains how to prepare a package so it can be easily built by other users without knowing about the package's building details. The third part, **??**, is intended for those who want to understand how robotpkg is implemented.

## 1.5 Terminology

Here is a description of all the terminology used within this document.

**Package** A set of files and building instructions that describe what's necessary to build a certain piece of software using robotpkg. Packages are traditionally stored under `/opt/robotpkg`.

**robotpkg** This is the The Robotics Package Collection. It handles building (compiling), installing, and removing of packages.

**Distfile** This term describes the file or files that are provided by the author of the piece of software to distribute his work. All the changes necessary to build are reflected in the corresponding package. Usually the distfile is in the form of a compressed tar-archive, but other types are possible, too. Distfiles are usually stored below `/opt/robotpkg/distfiles`.

**Precompiled/binary package** A set of binaries built with robotpkg from a distfile and stuffed together in a single `.tgz` file so it can be installed on machines of the same machine architecture without the need to recompile. Packages are usually generated in `/opt/robotpkg/packages`.

Sometimes, this is referred to by the term "package" too, especially in the context of precompiled packages.

**Program** The piece of software to be installed which will be constructed from all the files in the distfile by the actions defined in the corresponding package.

## 1.6 Roles involved in robotpkg

**robotpkg users** The robotpkg users are people who use the packages provided by robotpkg. Typically they are student working in robotics. The usage of the software that is *inside* the packages is not covered by the robotpkg guide.

There are two kinds of robotpkg users: Some only want to install pre-built binary packages. Others build the robotpkg packages from source, either for installing them directly or for building binary packages themselves. For robotpkg users, Part 2, The robotpkg user's guide, should provide all necessary documentation.

**package maintainers** A package maintainer creates packages as described in Part 3, The robotpkg developer's guide.

**infrastructure developers** These people are involved in all those files that live in the `mk/` directory and below. Only these people should need to read through Part **??**, **??**, though others might be curious, too.

## 1.7 Typography

When giving examples for commands, shell prompts are used to show if the command should/can be issued as root, or if "normal" user privileges are sufficient. We use a `#` for root's shell prompt, and a `%` for users' shell prompt, assuming they use the C-shell or tcsh.

# 2

# The robotpkg user's guide

Basically, there are two ways of using robotpkg. The first is to only install the package tools and to use binary packages that someone else has prepared. The second way is to install the programs from source. Then you are able to build your own packages, and you can still use binary packages from someone else. Sections in this document will detail both approaches where appropriate.

## 2.1 Where to get robotpkg and how to keep it up-to-date

Before you download and extract the files, you need to decide where you want to extract them and where you want robotpkg to install packages. By defaut, the `/opt/openrobots` directory is used. In the rest of this document, the installation path is called the *prefix*.

`robotpkg` will *never* require administration privileges by itself. We thus recommend that you do not install or run robotpkg as the root user. If something ever goes really wrong, it might go less wrong if it is not running as root. If you want to install to the default location `/opt/openrobots`, we recommend that you create this directory owned by a regular user.

Creating or using `/opt/openrobots` typically requires administration (*a.k.a.* "`root`") privileges. If you don't have such privileges (or if you want to install to a different location), you have to unpack the sources and install the binary packages in another prefix. If you don't have any special administration rights on the target machine, a safe bet is to choose the `$HOME/openrobots` location, as the `$HOME` directory will always be writable

by yourself.

Any prefix will work, but please note that you should choose an installation path which is dedicated to robotpkg packages and not shared with other programs (e.g., we do not recommend to use a prefix of `/usr`). Also, you should not try to add any of your own files or directories (such as `src/`) below the prefix tree. This will prevent possible conflicts between programs and other files installed by the package system and whatever else may have been installed there.

Finally, the installation path shall not contain white-space or other characters that are interpreted specially by the shell and some other programs: use only letters, digits, underscores and dashes.

The rest of this document will assume that you are using `/opt/openrobots` as the prefix. You should adapt this path to whatever prefix you choosed.

### 2.1.1 Getting the binary bootstrap kit

At the moment, the binary bootstrap kit is not available. Please get the `robotpkg` sources as described in the next section.

### 2.1.2 Getting robotpkg for source compilation

`robotpkg` sources are distributed *via* the `git` software content management system. `git` will probably be readily available on your system but if you don't have it installed or if you are unsure about it, contact your local system administrator.

There are two download methods: the anonymous one and the authenticated one:

- Anonymous download is the recommended method if you don't intend to work on the robotpkg infrastructure itself, nor commit any changes or packages additions back to the robotpkg main repository. Furthermore, the possibility to send contributions via patches is still open.

  As your regular user, simply run in a shell:

  ```
  % cd /opt/openrobots
  % git clone git://git.openrobots.org/robots/robotpkg
  % # or
  % git clone https://git.openrobots.org/robots/robotpkg.git
  ```

- Authenticated download requires a valid login on the main robotpkg repository, and will give you full commit access to this repository. Simply run the following:

```
% cd /opt/openrobots
% git clone ssh://git@git.openrobots.org/robots/robotpkg
```

### 2.1.3 Keeping robotpkg up-to-date

`robotpkg` is a living thing: updates to the packages are made perdiodicaly, problems are fixed, enhancements are developed... If you downloaded the robotpkg sources via git, you should keep it up-to-date so that you get the most recent packages descriptions. This is done by running `git pull` in the robotpkg source directory:

```
% cd /opt/openrobots/robotpkg
% git pull
```

When you update robotpkg, the git program will only touch those files that are registered in the git repository. That means that any packages that you created on your own will stay unmodified. If you change files that are managed by git, later updates will try to merge your changes with those that have been done by others. See the `git-pull` manual for details.

If you want to be informed of package additions and other updates, a public mailing list is available for your reading pleasure. Go to `https://sympa.laas.fr/sympa/info/robotpkg` for more information and subscription.

## 2.2 Bootstrapping robotpkg

Once you have downloaded the robotpkg sources or the binary bootstrap kit as described in Section 2.1, Where to get robotpkg and how to keep it up-to-date, a minimal set of the administrative package management utilities must be installed on your system before you can use robotpkg. This is called the "bootstrap phase" and should be done only once, the very first time you download robotpkg.

### 2.2.1 Bootstrapping via the binary kit

At the moment, the binary bootstrap kit is not available. Please bootstrap `robotpkg` as described in the next section.

### 2.2.2 Bootstrapping from source

You will need a working C compiler and the GNU-make utility version 3.81 or later. If you have extracted the robotpkg archive into the standard `/opt/openrobots/robotpkg` location, installing the bootstrap kit from source should then be as simple as:

```
% cd /opt/openrobots/robotpkg/bootstrap
% ./bootstrap
```

This will install various utilities into `/opt/openrobots/sbin`.

Should you prefer another installation path, you could use the `--prefix` option to change the default installation prefix. For instance, configuring robotpkg to install programs into the openrobots directory in your home directory can be done like this:

```
% cd robotpkg/bootstrap
% ./bootstrap --prefix=${HOME}/openrobots
```

**After the bootstrap script has run, a message indicating the success should be displayed. If you choosed a non-standard installation path, read this message carefuly**, as it contains instructions that you have to follow in order to setup your shell environment correctly. These instructions are described in the next section.

### Configuring your environment

If you configured robotpkg, during the bootstrap phase, to install to some other location than `/opt/openrobots`, you have to setup manually your shell environment so that it contains a few variables holding the installation path. Assuming you invoked bootstrap with `-prefix=/path/to/openrobots`, you have two options that are compatible with each other:

- Add the directory `/path/to/openrobots/sbin` to your `PATH` variable. robotpkg will then be able to find its administrative tools automatically and from that recover other configuration information. This is the preferred method.
- Create the environment variable `ROBOTPKG_BASE` and set its value to `/path/to/openrobots`. robotpkg will look for this variable first, so it takes precedence over the first method. This is the method you have to choose if you have configured several instances of robotpkg in your system. This is ony useful in some circumstances and is not normally needed.

If you don't know how to setup environment variables permanently in your system, please refer to your shell's manual or contact your local system administrator.

### The bootstrap script usage

The `bootstrap` script will by default install the package administrative tools in `/opt/openrobots/sbin`, use `gcc` as the C compiler and `make` as the GNU-make program. This behaviour can be fine-tuned by using the following options:

`--prefix <path>` will select the prefix location where programs will be installed in.

`--sysconfdir <path>` defaults to `<prefix>/etc`. This is the path to the robotpkg configuration file. Other packages configuration files (if any) will also be stored in this directory.

`--pkgdbdir <path>` defaults to `<prefix>/var/db/pkg`. This is the path to the package database directory where robotpkg will do its internal bookkeeping.

`--compiler <program>` defaults to `gcc`. Use this option if you want to use a different C compiler.

`--make <program>` defaults to `make`. Use this option if you want to use a different make program. This program should be compatible with GNU-make.

`--help` displays the `bootstrap` usage. The comprehensive list of recognized options will be displayed.

## 2.3 Using robotpkg

After obtaining `robotpkg` , the `robotpkg` directory now contains a set of packages, organized into categories. You can browse the online index of packages, or run `make index` from the `robotpkg` directory to build local `index.html` files for all packages, viewable with any web browser such as `lynx` or `firefox`.

`robotpkg` is essentially based on the `make(1)` program. All actions are triggered by invoking `make` with the proper target. The following sections document the most useful ones and section 2.3.7, Available `make` targets, recaps a more comprehensive list.

### 2.3.1 Building packages from source

The first step for building a package is downloading the *distfiles* (i.e. the unmodified source). If they have not yet been downloaded, `robotpkg` will fetch them automatically and place them in the `robotpkg/distfiles` directory.

Once the software has been downloaded, any patches will be applied and the package will be compiled for you. This may take some time depending on your computer, and how many other packages the software depends on and their compile time.

For example, type the following commands at the shell prompt to build the robotpkg documentation package:

```
% cd /opt/openrobots/robotpkg
% cd doc/robotpkg
% make
```

The next stage is to actually install the newly compiled package onto your system. While you are still in the directory for whatever package you are installing, you can do this by entering:

```
% make install
```

Installing the package on your system does not require you to be root (except for a few specific packages). However, if you bootstraped with a prefix for which you don't have writing permissions, `robotpkg` has a just-in-time-sudo feature, which allows you to become `root` for the actual installation step.

That's it, the software should now be installed under the prefix of the packages tree — `/opt/openrobots` by default — and setup for use.

You can now enter:

```
% make clean
```

to remove the compiled files in the work directory, as you shouldn't need them any more. If other packages were also added to your system (dependencies) to allow your program to compile, you can also tidy these up with the command:

```
% make clean-depends
```

Since the three tasks of building, installing and cleaning correspond to the typical usage of `robotpkg` , a helper target doing all these tasks exists and is called `update`. Thus, to intall a package with a single command, you can simply run:

```
% make update
```

In addition, `make update` will also recompile all the installed packages that were depending on the package that you are updating. This can be quite time consuming if you are updating a low-level package. Also, note that all packages that depend on the package you are updating will be deinstalled first and unavailable in your system until all packages are recompiled and reinstalled.

Occasionally, people want to "look under the covers" to see what is going on when a package is building or being installed. This may be for debugging purposes, or out of simple curiosity. A number of utility values have been added to help with this.

1. If you invoke the `make` command with `PKG_DEBUG_LEVEL=1`, then a huge amount of information will be displayed. For example,

   ```
   % make patch PKG_DEBUG_LEVEL=1
   ```

   will show all the commands that are invoked, up to and including the "patch" stage. Using `PKG_DEBUG_LEVEL=2` will give you even more details.

2. If you want to know the value of a certain `make` definition, then the `VARNAME` variable should be used, in conjunction with the `show-var` target. e.g. to show the expansion of the `make` variable `LOCALBASE`:

   ```
   % make show-var VARNAME=LOCALBASE
   ```

### 2.3.2 Building packages from a repository checkout

Before building a package, `robotpkg` fetches the sources from the official(s) download location(s), as instructed by the `MASTER_SITES` variable. This is the standard and expected behaviour when you work with stable packages.

Occasionally, though, it is useful to fetch a snapshot of the sources from a development repository. For instance, one might want to quickly test a release candidate of a package, or fix a simple bug and create a patch from the fix. Whenever a package defines the `MASTER_REPOSITORY` variable, `robotpkg` is able to temporarily work with the repository defined in this variable. At the moment, `cvs`, `svn` and `git` repositories are supported.

To enable this feature for a given package, you have to first instruct `robotpkg` to work from a '`checkout`' (instead of the stable releases) by doing '`make checkout`' in the package directory. For instance:

```
% cd robotpkg/foo/bar
% make checkout
```

This sets a permanent flag in the *working* directory of the package and the *checkout* configuration option will be retained until the next '`make clean`'. After a '`make clean`', the configuration option is set back to its default and `robotpkg` will work again with stable releases. This option is set on a *per* package basis only: configuring one package to work with checkouts does not affect the behaviour of other packages.

After a '`make checkout`' (and until a '`make clean`'), the package has a regular checkout in its *working* subdirectory. You can thus manually edit, commit, switch branches, etc. in the package sources, like in any other repository, by first `cd`ing into the working directory, then using the usual repository commands (`cvs`, `svn` or `git`).

Of course, the individual `robotpkg` targets are still available from the package entry in the robotpkg hierarchy. You can for instance `'make patch'`, `'configure'`, `'build'`, `'install'` or `'update'` as usual. Note that `robotpkg` is not exactly stateless, and this is most visible when working with checkouts: for instance, after a successful `'make build'`, you have to do `'make rebuild'` to force rebuilding if you have modified the sources. The same holds for `'configure'` (do `'reconfigure'`) or `'install'` (do 'reinstall', but since you cannot install a package twice, you normally have to use `'make replace'` in the particular case of reinstalling a package).

The `'clean'` target is special, in that it removes the checkout configuration option and all checkouted sources, including locally modified sources. In order to prevent accidental deletion of precious files, you have to confirm the cleanign with `'clean confirm'`, as in:

```
% make clean confirm
```

A final remark: we *STRONGLY DISCOURAGE* the use of robotpkg as a development tool (i.e. using the `'checkout'` feature on a *regular* basis), for at least two reasons:

- `robotpkg` is not designed for this: it will not really help you in your daily development work, compared to the manual configuration installation of the software. It will sometimes create even more trouble, by ensuring that all the software depending on the checkouted software is up-to-date, which is not necessarily something you want to do every time you compile.

- A checkout breaks the notion of 'release' and you loose all the benefits from working with packages. In particular, you have no clear state of what is installed: you cannot easily reproduce the situation of time T at time T+n and don't know precisely who requires which version of what. It is much more efficient and robust to release frequently a software in a development phase, than using a *rolling release* approach.

In our opinion, the `'checkout'` target use should be limited to testing a release candidate or quickly fix a bug and create a patch from the fix, that you commit upstream and put in the patches/ directory until the next release.

### 2.3.3 Installing binary packages

At the moment, installing binary packages is not documented.

### 2.3.4 Removing packages

To deinstall a package, it does not matter whether it was installed from source code or from a binary package. The `robotpkg_delete` command does not know it anyway. To delete a package, you can just run `robotpkg_delete <package-name>`. The package name can be given with or without version number. Wildcards can also be used to deinstall a set of packages, for example `*genom*` all packages whose name contain the word `genom`. Be sure to include them in quotes, so that the shell does not expand them before `robotpkg_delete` sees them.

The `-r` option is very powerful: it removes all the packages that require the package in question and then removes the package itself. For example:

```
% robotpkg_delete -r genom
```

will remove genom and all the packages that used it; this allows upgrading the `genom` package.

### 2.3.5 Getting information about installed packages

The `robotpkg_info` shows information about installed packages or binary package files.

### 2.3.6 Other administrative functions

The `robotpkg_admin` executes various administrative functions on the package system.

### 2.3.7 Available `make` targets

The following targets are available in a package directory. They can be invoked by running `make <target>` after `cd`ing into some `robotpkg/category/package`.

#### Source manipulation

`fetch` Download the `${DISTFILES}`.

`extract` Extract the contents of `${DISTFILES}` into the work directory `${WRKDIR}`.

`patch` Apply local patches available in `${PATCHDIR}` (usually the `patches` directory in the package).

`checkout` Extract the sources in `${WRKDIR}` from `${MASTER_REPOSITORY}` instead of `${MASTER_SITES}`. This can be used to fetch a not yet released version instead of the latest release. This is mutually exclusive with the `fetch` and `extract` targets. See section 2.3.2, Building packages from a repository checkout, for details.

`configure` Perform the necessary actions to configure the sources. This may for instance involve running `configure` or `cmake`. If no configuration is required, this step simply does nothing.

`build` Or just `make`, the default target. It compiles the package locally in `${WRKDIR}`.

`install` Install the package into `${PREFIX}`. The package is then available to the rest of the system. If an older version of the package is installed and required by other packages, this target requires confirmation. Otherwise, any older version of the package is first deinstalled.

`replace` Same as `install`, but does not remove packages that depend on the replaced package. This saves some time, since already installed package are not touched, but if the replaced package is incompatible with the older version, you will run into trouble. Use with care and when you know what you are doing.

`clean` Tidy the work directory and removes `${WRKDIR}`. If the package was extracted using `checkout`, this target requires confirmation as it may delete locally modified files that will be lost.

`update` This is a shortcut target for `fetch`, `extract`, `configure`, `build`, `install` and `clean`. If the package is already installed and up-to-date, the target asks for confirmation.

### Introspection

`show-options` Display the list of available alternatives (see section 2.4.2, Selecting build alternatives) and build options (see section 2.4.1, Selecting build options).

`show-depends` Recursively display all the required dependencies of a package. The results are splitted between system and `robotpkg` dependencies, and missing dependencies are indicated.

`show-var` Display the contents of a variable. This must be invoked as `make show-var VARNAME=foo`, where `foo` is the name of the variable to be displayed.

### Package sets

`fetch-depends`, `replace-depends`, `update-depends`, `clean-depends` This runs the same action as `fetch`, `replace`, `update` or `clean` (respectively), but on all dependencies of the package, including the package itself. Useful to update a meta-packages, for instance.

`fetch-<set>`, `replace-<set>`, `update-<set>`, `clean-<set>` This runs the same action as `fetch`, `replace`, `update` or `clean` (respectively), but on all members of the package set named `<set>`. See section 2.4.3, Defining collections of packages, for an explanation of package sets and how to configure them.

## 2.4 Configuring robotpkg

The whole `robotpkg` system is configured *via* a single, centralized file, called `robotpkg.conf` and placed in the `/opt/openrobots/etc` directory by default. This location might be redefined during the bootstrap phase, see Section 2.2, Bootstrapping robotpkg. During the bootstrap, an initial configuration file is created with the settings you provided to `bootstrap`.

The format of the configuration file is that of the usual GNU style `Makefile`s. The whole `robotpkg` configuration is done by setting variables in this file. Note that you can define all kinds of variables, and no special error checking (for example for spelling mistakes) takes place, so you have to try it out to see if it works.

### 2.4.1 Selecting build options

Some packages have build time options, usually to select between different dependencies, enable optional support for big dependencies or enable experimental features.

To see which options, if any, a package supports, and which options are mutually exclusive, run `make show-options`. For example:

```
Any of the following general options may be selected:
   api                  Generate module API only
   debug                Produce debugging information for binary programs
*  openprs              Generate OpenPRS client code
*  python               Enable Python client code
*d tcl                  Generate TCL client code
*  tclserv_client       Generate C tclServ client code
   xenomai              Enable Xenomai support
```

This indicates that the package supports a number of options (`api`, `debug`, `openprs` ...). The currently enabled options are indicated by a star (`*`) and the default options are shown by the small letter `d` in front of each option (here, only the `tcl` is enabled by default).

The following variables can be defined in `robotpkg.conf` to select which options to enable for a package:

`PKG_DEFAULT_OPTIONS` can be used to select or disable options for all packages that support them,

`PKG_OPTIONS.<pkgbase>` can be used to select or disable options specifically for package `<pkgbase>`. Options listed in these variables are selected, options prefixed by `-` are disabled (e.g. `-tcl` would disable the `tcl` option).

A few examples:

```
PKG_DEFAULT_OPTIONS=    debug
PKG_OPTIONS.genom=      doc -tcl
```

It is important to note that options that were specifically suggested by the package maintainer must be explicitely removed if you do not wish to include the option. If you are unsure you can view the current state with `make show-options`.

The following settings are consulted in the order given, and the last setting that selects or disables an option is used:

1. the default options as suggested by the package maintainer,
2. `PKG_DEFAULT_OPTIONS`,
3. `PKG_OPTIONS.<pkgbase>`

For groups of mutually exclusive options, the last option selected is used, all others are automatically disabled. If an option of the group is explicitly disabled, the previously selected option, if any, is used. It is an error if no option from a required group of options is selected, and building the package will fail.

### 2.4.2 Selecting build alternatives

Some packages have alternative dependencies, usually to select between equivalent components or versions of components. This is similar to options but the configuration is done globally for all packages that use the same alternatives (this is to ensure consistency between packages).

To see which alternatives, if any, a package uses, run `make show-options`. For example:

```
Available c-compiler alternatives (PREFER_ALTERNATIVE.c-compiler):
*1 gcc                  Use the GNU C compiler
 2 clang                Use the LLVM C compiler
   ccache-gcc           Use ccache and the GNU C compiler
   ccache-clang         Use ccache and the LLVM C compiler
```

This indicates that the package supports a `c-compiler` alternative, for which `gcc`, `clang`, `ccache-gcc` and `ccache-clang` can be used. The currently selected alternative is shown by the star (`*`), and the user preferences

(or the default if the user has not set explicit preferences) are indicated by the integer in front of the alternative item (here `gcc` is the preferred alternative, then `clang` should be used if `gcc` is not available. `ccache` should not be used).

The following variables can be defined in `robotpkg.conf` to select which alternative to use:

`PREFER_ALTERNATIVE.<alt>` Alternatives are selected by setting the variable corresponding to the alternative (`PREFER_ALTERNATIVE.c-compiler` in the example above) to a space separated list, sorted by order of preference, containing one or several of the items shown by `make show-options`.

### 2.4.3 Defining collections of packages

Instead of installing, removing or updating packages one-by-one, you can define collections of packages in your `robotpkg.conf`. Once one or more collections are defined, they enable special targets that work on all the packages of a collection.

To define a collection, you have to give it a name and list the set of packages forming the collection in the special `PKGSET` variable in `robotpkg.conf`. The syntax is the following:

```
PKGSET.<name> = <list>
PKGSET_DESCR.<name> = short, optional description of the collection
```

where `<name>` is the name of the collection (any string is valid) and `<list>` is the list of packages in the collection, in the form `<category>/<name>`. For instance,

```
PKGSET.myset = architecture/genom middleware/pocolibs
PKGSET_DESCR.myset = an awesome duo
```

defines a collection named `myset` that contains the two packages `genom` and `pocolibs` and describes itself with a rather doubtful sentence.

For each collection `<name>` defined in `robotpkg.conf`, the following targets are available: `clean-<name>`, `fetch-<name>`, `extract-<name>`, `install-<name>`, `replace-<name>`, `update-<name>` and `deinstall-<name>`. They perform the same action as their respective counterpart without `-<name>` suffix, expect that they work on all packages of the set. In addition, for the `replace`, `update` and `deinstall` targets, they sort the packages in the order of their dependencies so that the job is done a sensible order.

For the user convenience, two special targets are provided. The "`installed`" collection is always defined and represents all currently installed packages. Invoking, for instance, the `update-installed` target will therefore update

all currently installed packages. The "`depends`" collection is available only when the current working directory is inside a package. It merely defines the current package and all of its dependencies as the sole elements of the collection. Invoking, for instance, the `update-depends` target will update all dependencies of the package in the current directory.

Two `robotpkg.conf` variables affect the default behaviour of `robotpkg` regarding packages sets:

**PKGSET_FAILSAFE** When working on a set, and this variable is set to yes, robotpkg will continue with further packages instead of stopping on an error. If set to 'no', stop on first error. Default: no.

**PKGSET_STRICT** Specify if package sets should be considered as 'strict' or include dependencies of packages defined in the set. If set to 'yes', only package strictly defined in sets are considered. If set to 'no', dependencies of packages listed in sets are added to their respective sets. Default: no.

Each of these variables can be defined on a per-collection basis, by adding the .<name> suffix to the variable name, where <name> is the name of the collection to be configured.

### 2.4.4 Package specific configuration variables

In this section, you can find variables that apply to one specific package. Each variable is suffixed by `.<pkg>`, where `<pkg>` is the actual package name to which the variable should apply.

**REPOSITORY.<pkg>** locally overrides the default `MASTER_REPOSITORY` defined for a package. This is useful if you want to work with an alternative, perhaps local, repository when doing a `make checkout`.

**CHECKOUT_VCS_OPTS.<pkg>** is a list of options used when fetching a package via a `make checkout` command. The options are passed to the "cvs checkout", "git clone" or "svn checkout" command that extract the source archive.

### 2.4.5 General configuration variables

In this section, you can find some variables that apply to all `robotpkg` packages.

**ACCEPTABLE_LICENSES** List of acceptable licenses. Whenever you try to build a package whose license is not in this list, you will get an error message that includes instructions on how to change this variable.

**DISTDIR** Where to store the downloaded copies of the original source distributions used for building `robotpkg` packages. The default is `$ROBOTPKG_DIR/distfiles`.

**PACKAGES** The top level directory for the binary packages. The default is `$ROBOTPKG_DIR/packages`.

**MASTER_SITE_BACKUP** List of backup locations for distribution files if not found locally or in `$MASTER_SITES`. The default is `http://softs.laas.fr/openrobots/robotpkg/distfiles/`.

**PKG_DEBUG_LEVEL** The level of debugging output which is displayed whilst making and installing the package. The default value for this is 0, which will not display the commands as they are executed (normal, default, quiet operation); the value 1 will display all shell commands before their invocation, and the value 2 will display both the shell commands before their invocation, and their actual execution progress with `set -x`.

### 2.4.6 Variables affecting the build process

**WRKOBJDIR** The top level directory where, if defined, the separate working directories will get created. This is useful for building packages on a different filesystem than the `robotpkg` sources.

**OBJHOSTNAME** If set to yes (the default), use hostname-specific working directories, e.g. work.cactus, work.localhost. `OBJHOSTNAME` takes precedence over `OBJMACHINE` (see below).

**OBJMACHINE** If set to yes (the default) use machine-specific working directories, e.g. work.Linux-i386.

**DEPENDS_TARGET** By default, dependencies are only installed, and no binary package is created for them. You can set this variable to `package` to automatically create binary packages after installing dependencies.

**LOCALBASE** Where packages will be installed. The default value is `/opt/openrobots`. Do not mix binary packages with different values of `LOCALBASE`s!

**MAKE_JOBS** When defined, specifies the maximum number of jobs that are run in parallel when building packages with the default action. `MAKE_JOBS` only affects the "build" target. `MAKE_JOBS` can be set to any positive integer; useful values are around the number of processors on the machine.

### 2.4.7 Additional flags to the compiler

If you wish to set compiler variables such as `CFLAGS`, `CXXFLAGS`, `FFLAGS` ... please make sure to use the `+=` operator instead of the `=` operator:

```
CFLAGS+= -your -flags
```

Using `CFLAGS=` (i.e. without the "`+`") may lead to problems with packages that need to add their own flags.

If you want to pass flags to the linker, both in the configure step and the build step, you can do this in two ways. Either set `LDFLAGS` or `LIBS`. The difference between the two is that `LIBS` will be appended to the command line, while `LDFLAGS` come earlier. `LDFLAGS` is pre-loaded with rpath settings for machines that support it. As with `CFLAGS` you should use the `+=` operator:

```
LDFLAGS+= -your -linkerflags
```

## 2.5 Creating binary packages for everything

There are two ways of getting a set of binary packages: manually building the packages you need, or using `robotpkg` "bulk build" infrastructure.

Bulk builds can also be used to test that packages compile and install cleanly, and `robotpkg` provides reporting tools that can summarize the results of a "bulk build".

### 2.5.1 Initial setup

The required setup for running bulk build merely consists in properly setting up `robotpkg` itself. Details can be found in sections 2.2, Bootstrapping robotpkg, and 2.4, Configuring robotpkg.

For instance, setup `robotpkg` in the `/local/robotpkg` directory:

```
% mkdir -p /local/var/lib
% cd /local/var/lib
% git clone git://git.openrobots.org/git/robots/robotpkg
% cd robotpkg/bootstrap
% ./bootstrap --prefix=/local/robotpkg
```

You should install at least `pkgtools/pkg_install`, `pkgtools/digest` and `pkgtools/tnftp`. Optionally, you can install `pkgtools/rbulkit` that can generate pretty HTML reports (section 2.5.3, Generating pretty reports).

```
% cd /local/var/lib/robotpkg
% cd pkgtools/rbulkit
% make update
```

You must configure the prefix directory where binary packages are built. This is important: since binary package are not relocatable, this directory will be the installation directory of all generated packages. However, if you use bulk builds only as a way to test the build of your packages, any directory can be configured. The following variables can be customized in `robotpkg.conf`:

**BULKBASE?= /opt/openrobots** The installation prefix of binary packages. This **must** be different from the `${LOCALBASE}` directory where regular robotpkg packages are installed. The default is `/opt/openrobots`.

**BULK_LOGDIR?= ${LOCALBASE}/var/log/bulk** The directory where log files are kept. The default is to put log files in the regular installation prefix of robotpkg (`/local/robotpkg/var/log/bulk` in the example setup above).

**BULK_TAG** A name (alphanumeric characters) for that bulk session. The default name is based on the machine operating system and version. Giving a different name can be used to distinguish between different runs, but in this case it is probably easier to pass that variable definition on the command line.

Finally, you must define at least one package set (see section 2.4.3, Defining collections of packages) containing the list of packages to build (running bulk build on a single package is also supported, but has only limited interest). For instance, create a "`huge`" and a "`tiny`" set by placing the following definitions in your `robotpkg.conf` file:

```
PKGSET_DESCR.huge= Huge bulk set: everything!
PKGSET.huge=  */*:*

PKGSET_DESCR.tiny= Tiny bulk set: just one package + dependencies
PKGSET.tiny=  shell/eltclsh
PKGSET_STRICT.tiny= no
```

### 2.5.2 Running bulk builds

The target for running bulk builds is called `bulk`. You can run a bulk build for one package by just running `make bulk` in the package directory. Running `make bulk-depends` would run the bulk target on the package and all of its dependencies. More useful though is to use some predefined sets of packages as explained in the previous section:

```
% cd /local/var/lib/robotpkg
% make bulk-tiny
```

This would run the `bulk` target on each package of the `tiny` set (see section 2.4.3, Defining collections of packages, for an explanation of the -<set> targets).

This should run for a while and enventually populate `${BULK_LOGDIR}` with lots of log files. You can then examine them manually, or generate some reports with `rbulkit`.

### 2.5.3 Generating pretty reports

If you installed the `pkgtools/rbulkit` package, you can use the `rbulk-report` program (installed in `<prefix>/sbin`) to generate:

- an `sqlite` database containing the bulk results
- an HTML report

With the sample setup used throughout this chapter, the `sqlite` database can be populated like so:

```
% rbulk-report log2db /local/robotpkg/var/log/bulk sqlite.db
```

Note that the command will *append* the results to a pre-existing `sqlite.db`. Only results using the same `BULK_TAG` will be overwritten.

The HTML report can then be updated like so:

```
% rbulk-report db2html sqlite.db /tmp/bulk-report/
```

The report can then be viewed by pointing a web browser to `/tmp/bulk-report/index.html`.

To go further, please read the `rbulk-report(1)` manual for available commands and options.

### 2.5.4 Automated bulk builds

The `pkgtools/rbulkit` package contains sample scripts and programs that can be used to automate bulk builds. First, there is the `rbulk-build` script, that does essentially all the steps described in the previous sections. See `rbulk-build(1)` for more information. It relies on a properly setup `robotpkg` and `robotpkg.conf`.

There are also the `rbulk-watchd` and `rbulk-notify` programs, than can respectively wait for and signal notifications over TCP. This can be used to trigger a bulk build in a commit hook, for instance. See `rbulk-watchd(1)` and `rbulk-notify(1)`.

Finally, `rbulk-dispatch` and `rbulk-dispatchd` are able to parallelize jobs on a group of machines according to user defined priorities. See `rbulk--dispatchd(1)`.

# 3

# The robotpkg developer's guide

This part of the documentation deals with creating and modifying packages.

## 3.1 Package files, directories and contents

Whenever you're preparing a package, there are a number of files involved which are described in the following sections.

### 3.1.1 Makefile

Building, installation and creation of a package are all controlled by the package's Makefile. The Makefile describes various things about a package, for example from where to get it, how to configure, build, and install it.

A package Makefile contains several sections that describe the package.

In the first section there are the following variables, which should appear exactly in the order given here. The order and grouping of the variables is mostly historical and has no further meaning.

**PKGREVISION** Defines the `robotpkg` revision number of the package. This *should not be set* for a new package. See Section 3.2.4, Incrementing versions when fixing an existing package, for details.

**MASTER_SITES** In simple cases, `MASTER_SITES` defines all URLs from where the distfile, whose name is derived from the `DISTNAME` variable, is fetched.

When actually fetching the distfiles, each item from `MASTER_SITES` gets the name of each distfile appended to it, without an intermediate slash. Therefore, all site values have to end with a slash or other separator character. This allows for example to set `MASTER_SITES` to a URL of a CGI script that gets the name of the distfile as a parameter. In this case, the definition would look like:

```
MASTER_SITES= http://www.example.com/download.cgi?file=
```

There are some predefined values for `MASTER_SITES`, which can be used in packages. The names of the variables should speak for themselves.

```
${MASTER_SITE_SOURCEFORGE}
${MASTER_SITE_GNU}
${MASTER_SITE_OPENROBOTS}
```

If you choose one of these predefined sites, you may want to specify a subdirectory of that site. Since these macros may expand to more than one actual site, *you must* use the following construct to specify a subdirectory:

```
MASTER_SITES= ${MASTER_SITE_SOURCEFORGE:=project_name/}
```

Note the trailing slash after the subdirectory name.

**FETCH_METHOD** This is the method used to download the distfile from `MASTER_SITES`. It defaults to '`archive`' which corresponds to the normal situation where distfile is an archive available from `MASTER_SITES`, so it normally needs not to be set.

However, it can happen that a software provider does not provide any archive available for download but has only a public repository. In this case, `FETCH_METHOD` can be set to `cvs`, `git` or `svn` according to the kind of repository available. `MASTER_SITES` is then interpreted as a repository of the form `url[@revision[+module]]`, where the bits between square brackets are optional and refer to a particular revision and module in the repository located at `url`. `url` can take any form supported by the underlying fetch tool (`cvs`, `git` or `svn`). It is *strongly* advised to define at least a specific revision to be checked out, so that the package can be reproducibly installed in a known state.

**MASTER_REPOSITORY** defines a VCS repository from where a "`make checkout`" will download the latest revision of a software. `MASTER_REPOSITORY`

is a list of 2 or 3 elements. The first element is the VCS tool to be used: it must be one of `cvs`, `git` or `svn`. The second element is the location of the repository. It must be written in a syntax understood by the actual VCS tool. The third optional element is a list of specific elements to be checked out instead of the default (the whole repository).

**CHECKOUT_VCS_OPTS** is a list of options used when fetching a package via a `make checkout` command. The options are passed to the "cvs checkout", "git clone" or "svn checkout" command that extract the source archive.

The second section contains information about separately downloaded patches, if any.

**PATCHFILES** Name(s) of additional files that contain distribution patches distributed by the author or other maintainers. There is no default. robotpkg will look for them at `PATCH_SITES`. They will automatically be uncompressed before patching if the names end with .gz or .Z.

**PATCH_SITES** Primary location(s) for distribution patch files (see `PATCHFILES` above) if not found locally.

The third section contains the following variables.

**MAINTAINER** is the email address of the person who feels responsible for this package, and who is most likely to look at problems or questions regarding this package. Other developers may contact the `MAINTAINER` before making changes to the package, but are not required to do so. When packaging a new program, set `MAINTAINER` to yourself. If you really can't maintain the package for future updates, set it to `<robotpkg@laas.fr>`.

**HOMEPAGE** is a URL where users can find more information about the package.

**COMMENT** is a one-line description of the package (should not include the package name).

**LICENSE** Denoting that a package may be installed and used according to a particular license is done by placing the license in `robotpkg/licenses` and setting the LICENSE variable to a string identifying the license file, e.g. in `shell/eltclsh`:

LICENSE= 2-clause-bsd

The license tag mechanism is intended to address copyright-related issues surrounding building, installing and using a package, and not to address redistribution issues (see RESTRICTED and NO_PUBLIC_SRC, etc.). Packages with redistribution restrictions should set these tags.

Other variables affecting the build process may be gathered in their own section.

**PKG_SUPPORTED_OPTIONS** is the list of build options supported by the package. A number of related variables are used in combination with `PKG_SUPPORTED_OPTIONS`. See Section 3.2.1, Adding build options to a package, for details.

### 3.1.2 distinfo

The distinfo file contains the message digest, or checksum, of each distfile needed for the package. This ensures that the distfiles retrieved from the Internet have not been corrupted during transfer or altered by a malign force to introduce a security hole. Due to recent rumor about weaknesses of digest algorithms, all distfiles are protected using both SHA1 and RMD160 message digests, as well as the file size.

The distinfo file also contains the checksums for all the patches found in the patches directory (see Section 3.1.4, patches/*).

To regenerate the distinfo file, use the `make distinfo` or `make mdi` command.

### 3.1.3 PLIST

This file governs the files that are installed on your system: all the binaries, manual pages, etc. There are other directives which may be entered in this file, to control the creation and deletion of directories, and the location of inserted files.

The names used in the PLIST are relative to the installation prefix (`${PREFIX}`), which means that it cannot register files outside this directory (absolute path names are not allowed). As a general sanity rule, robotpkg must not alter any files outside `${PREFIX}` anyway and, in particular, not modify automatically existing configuration files. If a package needs to install files outside `${PREFIX}`, the best option is to install them with robotpkg inside `${PREFIX}` (e.g. `${PREFIX}/etc` or `${PREFIX}/var`) and create a `MESSAGE` file that will instruct the user to manually link or copy the files in question to their final location. See the package `hardware/ieee1394-kmod` for an example of such package.

In order to create or update a `PLIST`, you can use the `make print-PLIST` command to output a PLIST that matches any new installed files since the package was extracted. This command will generate a `PLIST.guess` file

which you must move manually to PLIST after reviewing the result of the semi-automatic generation. In order to fine tune the PLIST or its semi-automatic generation, specific variables documented in section 3.2.2, Customizing the PLIST, and section 3.2.3, Customizing the semi-automatic PLIST generation, may be used.

### 3.1.4 patches/*

Some packages may not work out-of-the box with robotpkg. Therefore, a number of custom patch files may be needed to make the package work. These patch files are found in the patches/ directory. If you want to share patches between multiple packages in robotpkg, e.g. because they use the same distfiles, set PATCHDIR to the path where the patch files can be found, e.g.:

PATCHDIR= ../../devel/boost/patches

The file names of the patch files must be of the form patch-*, and they are usually named patch-[a-z][a-z]. In the *patch* phase, these patches are automatically applied to the files in ${WRKSRC} directory after extracting them, in alphabetic order.

The patch-* files should be in diff -bu format, and apply without a fuzz to avoid problems. (To force patches to apply with fuzz you can set PATCH_FUZZ_FACTOR=-F2 in a package's Makefile).

Each patch file should be commented so that any developer who knows the code of the application can make some use of the patch. Special care should be taken for the upstream developers, since we generally want that they accept robotpkg patches, so there is less work in the future. When adding a patch that corrects a problem in the distfile (rather than e.g. enforcing robotpkg's view of where man pages should go), send the patch as a bug report to the maintainer. This benefits non-robotpkg users of the package, and usually makes it possible to remove the patch in future version.

When you add or modify existing patch files, remember to generate the checksums for the patch files by using the make mdi command, see Section 3.1.2, distinfo.

## 3.2 General operation

### 3.2.1 Adding build options to a package

Build options (see section 2.4.1, Selecting build options, for details) can be defined in a package by properly configuring the following variables.

**PKG_SUPPORTED_OPTIONS** This is a list of build options supported by the package. This variable should be set in a package Makefile. E.g.,

```
PKG_SUPPORTED_OPTIONS= ipv6 ssl
```

If this variable is not defined, `PKG_OPTIONS` is set to the empty list and the package is otherwise treated as not using the options framework.

**PKG_OPTION_DESCR.<opt>** This is the textual description of the option <opt> which is displayed by the `make show-options` target. E.g.,

```
PKG_OPTION_DESCR.bar= Enable the bar option.
```

**PKG_OPTION_SET.<opt> (resp. PKG_OPTION_UNSET.<opt>)** This is a makefile fragment that is evaluated when the option <opt> is set (resp unset) for the package. E.g.,

```
PKG_OPTION_SET.bar= CFLAGS+=-DBAR
PKG_OPTION_UNSET.bar= CFLAGS+=-DNO_BAR
```

Complex (multiline) `_SET` or `_UNSET` actions can be defined with the `define` command of GNU-make. It is for instance possible to add additional dependencies: see the example below.

**PKG_OPTIONS_OPTIONAL_GROUPS** This is a list of names of groups of mutually exclusive options. The options in each group are listed in `PKG_OPTIONS_GROUP.<groupname>`. The most specific setting of any option from the group takes precedence over all other options in the group. Options from the groups will be automatically added to `PKG_SUPPORTED_OPTIONS`.

**PKG_OPTIONS_REQUIRED_GROUPS** Like `PKG_OPTIONS_OPTIONAL_GROUPS`, but building the packages will fail if no option from the group is selected.

**PKG_OPTIONS_NONEMPTY_SETS** This is a list of names of sets of options. At least one option from each set must be selected. The options in each set are listed in `PKG_OPTIONS_SET.<setname>`. Options from the sets will be automatically added to `PKG_SUPPORTED_OPTIONS`.

**PKG_OPTIONS_SUFFIX** The suffix in `PKG_OPTIONS.suffix` variable the user can set to enable or disable options specifically for this package. Defaults to `${PKGBASE}`.

**PKG_SUGGESTED_OPTIONS** This is a list of build options which are enabled by default. This defaults to the empty list.

Here is an example Makefile fragment for a 'wibble' package. This fragment should be included in the 'wibble' package Makefile.

```
PKG_OPTIONS_SUFFIX=  wibble # this is the default
PKG_SUPPORTED_OPTIONS=  foo bar
PKG_OPTIONS_OPTIONAL_GROUPS= robot
PKG_OPTIONS_GROUP.robot= lama hrp2
PKG_SUGGESTED_OPTIONS=  foo

PKG_OPTION_DESCR.foo=  Enable the foo option.
PKG_OPTION_DESCR.bar=  Build with the bar package.
PKG_OPTION_DESCR.lama=  Build for the lama robot.
PKG_OPTION_DESCR.hrp2=  Build for the hrp2 robot.

define PKG_OPTION_SET.bar
 CFLAGS+=-DNO_BAR
 include ../../pkg/bar/depend.mk
endef
PKG_OPTION_UNSET.bar=  CFLAGS+=-DNO_BAR
```

### 3.2.2 Customizing the PLIST

The packing list of a package is usually computed from the `PLIST` file located in the package directory. The following variables determine how the final packing list is setup:

**PLIST_SRC** The source file(s) for the final packing list. By default, its value is constructed from the PLIST.* files within the package directory:

```
PLIST_SRC+= ${PKGDIR}/PLIST.${OS_KERNEL}
PLIST_SRC+= ${PKGDIR}/PLIST.${OPSYS}
PLIST_SRC+= ${PKGDIR}/PLIST.${MACHINE_ARCH}
PLIST_SRC+= ${PKGDIR}/PLIST.${OPSYS}-${MACHINE_ARCH}
PLIST_SRC+= ${PKGDIR}/PLIST
```

If a Makefile sets `PLIST_SRC`, the defaults are not used.

**DYNAMIC_PLIST_DIRS** A list of directories, absolute or relative to the installation `${PREFIX}`, whose contents are dynamically added to the final packing list. This is useful to handle non-deterministic packing lists, most notably those generated by `doxygen`. This should be used with care, since `DYNAMIC_PLIST_DIRS` somewhat defeats the purpose of the packing list.

**PLIST_SUBST** The PLIST file(s) of a package may also contain variable references (in the ${VAR} form) that are expanded at intallation time. The following variables are supported by default:

```
PKGBASE
PKGNAME
PKGVERSION

OPSYS
OS_VERSION
OS_KERNEL
OS_KERNEL_VERSION

PKGMANDIR
PKGINFODIR

PLIST.<opt> # for all supported options
PLIST.no<opt>
```

PLIST.<opt> is special: one such variable is defined for each supported build option of the package. It can be used to dynamically enable an entry of the packing list, depending on the build options configuration. A ${PLIST.<opt>} variable may only be present only at the beginning of a line. Technically, ${PLIST.<opt>} expands to a packing list comment @comment when the option <opt> is not enabled, and to the empty string otherwise.

PLIST.no<opt> is similar to PLIST.<opt>, but it enables a PLIST entry only if the corresponding option is not enabled.

Other substitutions may be added by adding definitions to the PLIST_SUBST variable. For instance, a package may define the FOO variable substitution like so:

```
PLIST_SUBST+= FOO=${FOO}
```

This would instruct the packing list generator to replace all occurences of ${FOO} by the value of the ${FOO} variable in the Makefile.

**GENERATE_PLIST** A sequence of commands, terminating in a semicolon, that outputs contents for a PLIST to stdout and is appended to the contents of ${PLIST_SRC}. The default works for almost all packages, and it is usually not needed to define a custom command.

**PLIST_FILTER** A sequence of commands, each starting with a pipe, that filter out the packing list. This is to be used only in rare situations, and a standard package should not need to customize this.

### 3.2.3 Customizing the semi-automatic PLIST generation

The semi-automatic initial PLIST generation does not handle package options. If the list of installed files depends on the package build options, `${PLIST.<opt>}` variable references, detailed in section 3.2.2, Customizing the PLIST, must be manually added to the result of `make print-PLIST`.

### 3.2.4 Incrementing versions when fixing an existing package

When making fixes to an existing package it can be useful to change the version number in `PKGNAME`. To avoid conflicting with future versions by the original author, a "r1", "r2", ... suffix can be used on package versions by setting `PKGREVISION=1` (2, ...) in the package Makefile. E.g.

```
DISTNAME= foo-17.42
PKGREVISION= 9
```

will result in a `PKGNAME` of "foo-17.42r9". The "r" is treated like a "." by the package tools.

`PKGREVISION` should be incremented for any non-trivial change in the resulting binary package. Without a `PKGREVISION` bump, someone with the previous version installed has no way of knowing that their package is out of date. Thus, changes without increasing `PKGREVISION` are essentially labeled "this is so trivial that no reasonable person would want to upgrade", and this is the rough test for when increasing `PKGREVISION` is appropriate. Examples of changes that do not merit increasing `PKGREVISION` are:

- Changing `HOMEPAGE`, `MAINTAINER` or comments in Makefile.
- Changing build variables if the resulting binary package is the same.
- Changing `DESCR`.
- Adding `PKG_OPTIONS` if the default options don't change.

Examples of changes that do merit an increase to `PKGREVISION` include:

- Security fixes
- Changes or additions to a patch file
- Changes to the `PLIST`
- A dependency is changed or renamed.

`PKGREVISION` must also be incremented when dependencies have ABI changes.

When a new release of the package is released, the `PKGREVISION` must be removed.

### 3.2.5 Substituting variable text in the package files

When you want to replace the same text in multiple files or when the replacement text varies, patches alone cannot help. This is where the SUBST framework comes in. It provides an easy-to-use interface for replacing text in files. Example:

```
SUBST_CLASSES+=           fix-paths
SUBST_STAGE.fix-paths=    pre-configure
SUBST_MESSAGE.fix-paths=  Fixing absolute paths.
SUBST_FILES.fix-paths=    src/*.c
SUBST_SED.fix-paths=      -e 's,"/usr/local,"${PREFIX},g'
```

`SUBST_CLASSES` is a list of identifiers that are used to identify the different `SUBST` blocks that are defined. The `SUBST` framework is used by `robotpkg` , so it is important to always use the `+=` operator with this variable. Otherwise some substitutions may be skipped.

The remaining variables of each `SUBST` block are parameterized with the identifier from the first line (`fix-paths` in this case).

`SUBST_STAGE.*` specifies the stage at which the replacement will take place. All combinations of pre-, do- and post- together with a phase name are possible, though only few are actually used. Most commonly used are post-patch and pre-configure. Of these two, pre-configure should be preferred because then it is possible to run `make patch` and have the state after applying the patches but before making any other changes. This is especially useful when you are debugging a package in order to create new patches for it. Similarly, post-build is preferred over pre-install, because the install phase should generally be kept as simple as possible. When you use post-build, you have the same files in the working directory that will be installed later, so you can check if the substitution has succeeded.

`SUBST_MESSAGE.*` is an optional text that is printed just before the substitution is done.

`SUBST_FILES.*` is the list of shell globbing patterns that specifies the files in which the substitution will take place. The patterns are interpreted relatively to the `WRKSRC` directory.

`SUBST_SED.*` is a list of arguments to `sed(1)` that specify the actual substitution. Every sed command should be prefixed with -e, so that all `SUBST` blocks look uniform.

There are some more variables, but they are so seldomly used that they are only documented in the mk/internal/subst.mk file.

## 3.3 The build phase

For building a package, a rough equivalent of the following code is executed.

```
for d in ${BUILD_DIRS}; do                \
   cd ${WRKSRC}                           \
   && cd ${d}                             \
   && env ${MAKE_ENV}                     \
      ${MAKE_PROGRAM} ${BUILD_MAKE_FLAGS} \
         -f ${MAKE_FILE}                  \
         ${BUILD_TARGET}
done
```

The following variables affecting the build process of a package may be defined in a package `Makefile`:

**NO_BUILD** (default: unset). If there is no build step at all, set `NO_BUILD` to "yes".

**MAKE_PROGRAM** (default: `MAKE`) is the program used to perform the actual build in a package.

**BUILD_DIRS** (default: ".") is a list of pathnames relative to `WRKSRC`. In each of these directories, `MAKE_PROGRAM` is run with the environment `MAKE_ENV` and arguments `BUILD_MAKE_FLAGS`.

**BUILD_TARGET** (default: "all") is the default `make` target for building the package.

**MAKE_JOBS_SAFE** Whether the package supports parallel builds. If set to yes, at most `MAKE_JOBS` jobs are carried out in parallel. The default value is "yes", and packages that don't support it must explicitly set it to "no".