Dorodango Manual

Andreas Rottmann

This manual is for dorodango, version 0.0.1.

Table of Contents

Dorodango is a package manager written in and for R6RS Scheme.

From Wikipedia:

> Dorodango is a Japanese art form in which earth and water are molded to create a delicate shiny sphere, resembling a marble or billiard ball.

Using dorodango, you can easily install and distribute collections of R6RS libraries as well as programs using these libraries. It can handle dependencies, so when a program requires several libraries, and each of those has further dependencies, dorodango allows you to install all the prerequisites for that program in one go.

1 Getting Started

1.1 Installation

Dorodango comes in a tarball, which includes all its dependencies and an installation script. The installation script will probe for a supported Scheme implementation (currently Ikarus and Ypsilon), and uses dorodango to install itself and all dependencies into a selectable destination (see Section 2.2 [Destinations], page 5 for details).

It is recommended to install dorodango outside of the location(s) managed by it. There are several ways to achieve this:

1. Install dorodango in '`/usr/local/`', and use dorodango's default configuration, which will install packages into '`~/.local/`'.
2. Install dorodango in some other place other than '`~/.local/`', and symlink the `doro` script to some directory in your `PATH`.
3. Install dorodango into the default location, '`~/.local/`', and configure it to use some other location by default.

For option 1, installation works like this:

```
tar -xjf dorodango-full-0.0.1.tar.bz2
cd dorodango-0.0.1/dorodango
./setup --prefix /usr/local
```

The R6RS implementation is probed for automatically in the above example, but you can explicitly choose using the '`-i`' or '`--implementation`' option:

```
./setup --implementation ikarus --prefix /usr/local
```

Note that '`--implementation`' must go before other arguments.

After running `setup` like above, you should now have '`/usr/local/bin/doro`', and as '`/usr/local/bin`' is ordinarily in `PATH`, running `doro --help` should display a help message.

Besides the `doro` script, the installation process also creates a helper program `r6rs-script` in the same directory (i.e. '`/usr/local/bin`' in the running example). This script is specific to the implementation chosen when installing dorodango, and is used for running Scheme programs installed by dorodango.

Option 2 works nearly the same, except that you use a different argument after '`--prefix`', and afterwards make `doro` available in your PATH.

For option 3, you don't need to specify '`--prefix`', but you should configure a different default destination for dorodango to use. See Section 3.3 [Configuration File], page 9, for more information on configuring dorodango.

1.2 Quickstart

For the unpatient, this section presents the minium you need to know to use dorodango for installing software.

Configuration

So, you've successfully installed dorodango, and were able to get the help message via `doro --help`? Then it's time to tell dorodango where it can find software (see Section 2.1 [Packages], page 5) to install. Create the file '`~/.config/dorodango/config.scm`' and add this line:

```
(repository experimental "http://rotty.yi.org/doro/experimental")
```

This tells dorodango the location of the author's experimental repository, and gives it the name '`experimental`'. You could add further repositories with different names and locations, and dorodango will consider them all when installing packages.

Updating the package database

This is all configuration that is needed; if you now run `doro update`, you should see something like the following:

```
Fetching http://rotty.yi.org/doro/experimental/available.scm
Loading available information for repository ‘experimental’
```

Now dorodango has obtained the information of available packages from the repository. You can verify that by running `doro list --all`, which should produce output resembling the following:

```
u conjure                      0-20091204
u dorodango                    0-20091204
u fidfw                        0-20091204
...
```

The rightmost columns indicates the package state ('`u`' means "uninstalled"), the other columns are the package name and version.

Installing software

You can now install any of the listed packages, using `doro install` *`package-name`*:

```
% doro install spells
The following NEW packages will be installed:
  spells srfi{a}
Do you want to continue? [Y/n]
Fetching http://rotty.yi.org/doro/experimental/srfi_0-20091204.zip
Installing srfi (0-20091204) ...
Fetching http://rotty.yi.org/doro/experimental/spells_0-20091204.zip
Installing spells (0-20091204) ...
```

As demonstrated in the above verbatim, dorodango knows that the package '`spells`' depends on '`srfi`', and will automatically install that package as well.

Other important commands

Now you you know how to achieve the primary task of dorodango: installing software. There are a few other things you probably want to do at times:

`doro upgrade`
: Attempts to upgrade each package to the newest available version.

`doro remove`
: Allows you to remove packages from your system.

Getting help

For each command, you can invoke `doro` *`command`* `--help`, and it will show you what options and argument that command requires:

```
% doro remove --help
Usage: doro remove PACKAGE...
  Remove packages.

Options:
  --no-depends  ignore dependencies
  --help        show this help and exit
```

2 Overview

Dorodango is used via the `doro` command-line program, which allows you to automatically download, install, remove and upgrade R6RS library collections and programs that might be included with them.

A library collection, possibly including programs and documentation, together with some metadata, which, for example, describes the dependencies on other software, is called a package. Packages are distributed in ZIP files called ; each bundle may contain one or more packages.

If you already are familiar with other package managers, such as Debian's APT, having more than one package bundled in the same file might seem unusual to you, but don't worry: bundles are mostly transparent to the user. Most of the time, you will deal with packages, and bundles are of concern mostly when using dorodango to package your or other people's software.

2.1 Anatomy of a Package

A package is the "unit of software" dorodango works with. It has a name, and a version, which may be used to form dependency relationships among packages. It also may have other attributes, like a description and a homepage URL. Besides the metadata, a package also contains files, which are grouped into categories. Each category of a package conceptionally contains a set of files, along with their desired installation locations. The categories currently handled by dorodango are:

'`libraries`'
: R6RS libraries, and files required by them (either at runtime or at expand-time).

'`programs`'
: R6RS programs.

'`documentation`'
: README files, HTML documentation, etc.

2.2 Destinations

Now the files contained in these categories must be installed somewhere, and usually into different locations. The rules that describe where software is installed into are provided by a *destination*. You can select the destination by invoking the `doro` command line tool with the '`--prefix`' option, see Section 3.1 [Global Options], page 7. For each destination, dorodango maintains a database of installed and available packages.

Currently, all destinations have the same rules which should be suitable for POSIXish platforms, and especially for FHS platforms:

'`libraries`'
: Installed into '*PREFIX*`/share/r6rs-libs`'.

'`programs`'
: Installed into '*PREFIX*`/share/libr6rs-`*PACKAGE-NAME*`/programs`', and a shell wrapper in '*PREFIX*`/bin`' is created which starts the Scheme program via '`r6rs-script`', which is created automatically when dorodango initializes a destination.

‘`documentation`’
: Installed into *PREFIX*‘`/share/doc/libr6rs-PACKAGE-NAME`’.

2.3 Repositories

The bundles in which the packages are installed from are fetched from repositories. A repository is accessed via HTTP and is essentially a directory that contains bundles along with a file listing their locations and the packages within them.

3 Reference

`doro` is a command-line interface for downloading, installing and inspecting packages containing R6RS libraries and programs.

`doro` accepts, besides global options, a command and command-specific options. Each command is geared to a particuliar task, for example installing, removing or upgrading packages.

The following subsections document the available commands, grouped by related tasks.

3.1 Global Options

'`--no-config`'
: Do not read any configuration file. This option can be used to ensure that `doro` will just use built-in defaults.

'`--config=`*file*'
'`-c`'
: Use configuration file *file*, instead of the default one. See Section 3.3 [Configuration File], page 9, for configuration file syntax.

'`--prefix=`*prefix*'
: Use an FHS destination located below the directory *prefix*. This option has the same effect as adding an FHS destination to the configuration file and using it for this invocation of `doro`.

3.2 Command Reference

Note that all commands take a '`--help`' option showing brief usage information, although that option is not explicitly listed below.

3.2.1 Querying

The following commands gather information; either from the package database, uninstalled bundles, or about the configuration.

`list` [Command]
: Produces a list of packages, along with their installation status and version on standard output.

  '`--all`'
  '`-a`'
  : Show all packages, including uninstalled, but available ones. By default only installed packages are listed.

  '`--bundle=`*bundle*'
  '`-b` *bundle*'
  : Temporarily adds *bundle*'s contents to the package database.

`show` *package ...* [Command]
: Shows information about one or more packages. This command lists package, name, version and dependencies in RFC822-like style on standard output.

‘`--bundle=`*`bundle`*’
‘`-b` *`bundle`*’
: Temporarily adds *bundle*’s contents to the package database.

`show-bundle` *bundle ...* [Command]
: Shows the contents of one or more bundles on standard output. The content listing consist of each package’s information, as shown by the the `show` command, plus the package’s the list of files in each category. See Section 2.1 [Packages], page 5.

`config` [Command]
: Shows the current configuration in YAML-like style.

3.2.2 Package managment

`update` [Command]
: Download information about available packages from all repositories of the selected destination.

`install` *package ...* [Command]
: Install the listed *package*s. Each *package* argument can be a package name, in which case the newest available version is installed. If the package in question is already installed, it will be upgraded. One may also explicitly specifying a specific version to be installed using the syntax ‘*`package-name=version`*’.

‘`--bundle=`*`bundle`*’
‘`-b` *`bundle`*’
: Temporarily adds *bundle*’s contents to the package database.

‘`--no-depends`’
: Disable dependency resolution. This option allows for installing packages with unresolved dependencies.

`remove` *package ...* [Command]
: Remove the listed *package*s from the system.

‘`--no-depends`’
: Disable dependency resolution. This option allows for installing packages with unresolved dependencies.

`upgrade` [Command]
: Upgrade all packages to the newest available version.

`init` [Command]
: This command can be used to explicitly initialize a destination for use with a particuliar Scheme implementation. The initialization is otherwise done implicitly upon first opening the database for that destination, and uses the `default-implementation` configuration clause (see [default-implementation], page 10).

‘`--implementation=`*`impl`*’
‘`-i` *`impl`*’
: Select the implementation to use for that destination.

3.2.3 Development

The following commands are of use if you want to create your own packages and repositories.

`create-bundle` *directory ...* [Command]

Create a bundle from the directories given as arguments.

'`--output=`*`filename`*'
'`-o` *`filename`*'
: Output the bundle to *filename*. When this option is not given, dorodango will try to name the bundle based on the package contained in it. Should the bundle contain multiple packages, this option is mandatory.

'`--directory=`*`directory`*'
'`-d` *`directory`*'
: Output directory for the created bundle file. This option only has an effect when '`--output`' is *not* provided.

'`--append-version=`*`version`*'
: Rewrite the versions of all packages in the created bundle by appending *version*. This is useful, for e.g. creating "snapshot" bundles from a VCS, where one could append the current date to the upstream version.

`scan-bundles` *directory ...* [Command]

Search the directories passed as arguments for bundles and produce an "available file" containing information about found bundles on standard output.

'`--output=`*`filename`*'
'`-o` *`filename`*'

`symlink-bundle` *bundle-directory target-directory* [Command]

Create a symbolic link tree in *target-directory*, using the bundle at *bundle-directory*.

'`--force`' Allow the command to operate even when *target-directory* already exists.

'`--deep`' Create a symbolic link for every file. Without this option, `doro` will create symbolic links to directories when this doesn't change the created symlink tree.

'`--include=`*`packages`*'
: Create symbolic links just for the packages listed in the comma- or space-seperated list *packages*.

'`--exclude=`*`packages`*'
: Create symbolic links for all *but* the packages listed in the comma- or space-seperated list *packages*.

3.3 Configuration File

The configuration file stores permanent settings for dorodango, and can be selected with the '`--config`' option, see Section 3.1 [Global Options], page 7. It's syntax is S-expression-based clauses. In the following, we will dissect an example configuration file; note however, that for most users, a much simpler configuration will suffice (see Section 1.2 [Quickstart], page 2). Also the '`--prefix`' global option can be used to work with multiple destinations

without explicitly setting them up in the configuration file. Anyway, without further ado, here's a configuration that uses all possible clauses:

```
(repository experimental "http://rotty.yi.org/doro/experimental")
(repository unstable "http://rotty.yi.org/doro/unstable")
(destination unstable
  (fhs "/home/alice/scheme")
  (repositories unstable))
(destination experimental
  (fhs "/home/alice/scheme-experiments")
  (database "/home/alice/scheme-experiments/db"))
(default-destination experimental)
(default-implementation ypsilon)
```

3.3.1 Repositories

A repository clause defines a repository, which may be located on an HTTP server or a local file system. The repository's is given a name, and a location is specified as an URI:

```
(repository <name> <location-uri>)
```

In the running example, <*name*> is `experimental`, and <*location-uri*> name is the string `"http://rotty.yi.org/doro/experimental"`, denoting an HTTP repository at the apparent location.

3.3.2 Destinations

Destinations are where a package's files are installed to; they have an associated package database that keeps track of installed packages. In principle, destinations come in "flavors", but at the time of writing, there's only a single flavor: `fhs`, which puts the files in subdirectories of the specified prefix directory that are (at least roughly) in line with FHS. This means one can use an `fhs` destination to install to '`/usr/local`', and have files ending up in familiar locations.

In the configuration file, destinations are given a name so they can be referred to by `doro`'s '`--destination`' option. See Section 3.1 [Global Options], page 7.

Unless specified otherwise via the `repositories` sub-clause, all repositories listed up to the point of the destination's declaration will be used with this destination. A repository mus be declared before being referenced in a destination's `repositories` clause.

The `database` sub-clause allows to define the location of the package database on disk; if it is left out, dorodango will use a default location, based on the destination's prefix.

3.3.3 Defaults

The `default-implementation` clause specifies the Scheme implementation to use by default when setting up new destinations, affecting the implementation that R6RS programs installed into the location will use. One can initialize a destination using the `doro init` command, thus overriding the default.

If `default-implementation` is not specified, dorodango will use a built-in default (`ikarus` at the time of writing).

Using the `default-destination` one can specify which configured destination will be used when none is explicitly specified via the '`--destination`' global option. If there is no `default-implementation` clause, the first destination specified is considered the default.

3.3.4 Formal Grammar

A complete BNF-style grammar for the configuration file:

```
<configuration> -> <clause>*
<clause> ->  <repository> | <destination>
   | <default-destination> | <default-implementation>

<repository> -> (repository <name> <uri>)

<destination> -> (destination <name> <destination-spec> <option>*)
<option> -> (database <directory>)
   | (repositories <name>*)
<destination-spec> -> (fhs <directory>)

<default-destination> -> (default-destination <name>)
<default-implementation> -> (default-implementation <implementation>)
<implementation> = ikarus | ypsilon

<directory> -> <string>
<name> -> <symbol>
<uri> -> <string>
```

Index