Common Architecture for Building Applications and Libraries

User's Guide

Table of Contents

The *Cabal* aims to simplify the distribution of Haskell [http://www.haskell.org/] software. It does this by specifying a number of interfaces between package authors, builders and users, as well as providing a library implementing these interfaces.

1. Packages

A *package* is the unit of distribution for the Cabal. Its purpose, when installed, is to make available either or both of:

- A library, exposing a number of Haskell modules. A library may also contain *hidden* modules, which are used internally but not available to clients.[1]
- One or more Haskell programs.

However having both a library and executables in a package does not work very well; if the executables depend on the library, they must explicitly list all the modules they directly or indirectly import from

[1] Hugs doesn't support module hiding.

that library.

Internally, the package may consist of much more than a bunch of Haskell modules: it may also have C source code and header files, source code meant for preprocessing, documentation, test cases, auxiliary tools etc.

A package is identified by a globally-unique *package name*, which consists of one or more alphanumeric words separated by hyphens. To avoid ambiguity, each of these words should contain at least one letter. Chaos will result if two distinct packages with the same name are installed on the same system, but there is not yet a mechanism for allocating these names. A particular version of the package is distinguished by a *version number*, consisting of a sequence of one or more integers separated by dots. These can be combined to form a single text string called the *package ID*, using a hyphen to separate the name from the version, e.g. "`HUnit-1.1`".

Note

Packages are not part of the Haskell language; they simply populate the hierarchical space of module names. It is still the case that all the modules of a program must have distinct module names, regardless of the package they come from, and whether they are exposed or hidden. This also means that although some implementations (i.e. GHC) may allow several versions of a package to be installed at the same time, a program cannot use two packages, P and Q that depend on different versions of the same underlying package R.

2. Creating a package

Suppose you have a directory hierarchy containing the source files that make up your package. You will need to add two more files to the root directory of the package:

`.cab` *`package`*`al`
: a text file containing a package description (for details of the syntax of this file, see Section 2.1, "Package descriptions"), and

`Setup.hs` or `Setup.lhs`
: a single-module Haskell program to perform various setup tasks (with the interface described in Section 3, "Building and installing a package"). This module should import only modules that will be present in all Haskell implementations, including modules of the Cabal library. In most cases it will be trivial, calling on the Cabal library to do most of the work.

Once you have these, you can create a source bundle of this directory for distribution. Building of the package is discussed in Section 3, "Building and installing a package".

Example 1. A package containing a simple library

The HUnit package contains a file `HUnit.cabal` containing:

```
Name:           HUnit
Version:        1.1
License:        BSD3
Author:         Dean Herington
Homepage:       http://hunit.sourceforge.net/
Category:       Testing
Build-Depends:  base
Synopsis:       Unit testing framework for Haskell
Exposed-modules:
        Test.HUnit, Test.HUnit.Base, Test.HUnit.Lang,
```

```
        Test.HUnit.Terminal, Test.HUnit.Text
Extensions:     CPP
```

and the following `Setup.hs`:

```
import Distribution.Simple
main = defaultMain
```

Example 2. A package containing executable programs

```
Name:           TestPackage
Version:        0.0
License:        BSD3
Author:         Angela Author
Synopsis:       Small package with two programs
Build-Depends:  HUnit

Executable:     program1
Main-Is:        Main.hs
Hs-Source-Dirs: prog1

Executable:     program2
Main-Is:        Main.hs
Hs-Source-Dirs: prog2
Other-Modules:  Utils
```

with `Setup.hs` the same as above.

Example 3. A package containing a library and executable programs

```
Name:            TestPackage
Version:         0.0
License:         BSD3
Author:          Angela Author
Synopsis:        Package with library and two programs
Build-Depends:   HUnit
Exposed-Modules: A, B, C

Executable:      program1
Main-Is:         Main.hs
Hs-Source-Dirs:  prog1
Other-Modules:   A, B

Executable:      program2
Main-Is:         Main.hs
Hs-Source-Dirs:  prog2
Other-Modules:   A, C, Utils
```

with `Setup.hs` the same as above. Note that any library modules required (directly or indirectly) by an executable must be listed again.

The trivial setup script used in these examples uses the *simple build infrastructure* provided by the Cabal library (see Distribution.Simple [../libraries/Cabal/Distribution-Simple.html]). The simplicity lies in its interface rather that its implementation. It automatically handles preprocessing with standard preprocessors, and builds packages for all the Haskell implementations (except nhc98, for now).

The simple build infrastructure can also handle packages where building is governed by system-dependent parameters, if you specify a little more (see Section 2.3, “System-dependent parameters”). A few packages require more elaborate solutions (see Section 2.4, “More complex packages”).

2.1. Package descriptions

The package description file should have a name ending in “`.cabal`”. There must be exactly one such file in the directory. The first part of the name is immaterial, but it is conventional to use the package name.

In the package description file, lines beginning with “`--`” are treated as comments and ignored.

This file should contain one or more *stanzas* separated by blank lines:

- The first stanza describes the package as a whole (see Section 2.1.1, “Package properties”), as well as an optional library (see Section 2.1.2, “Library”) and relevant build information (see Section 2.1.4, “Build information”).
- Each subsequent stanza (if any) describes an executable program (see Section 2.1.3, “Executables”) and relevant build information (see Section 2.1.4, “Build information”).

Each stanza consists of a number of field/value pairs, with a syntax like mail message headers.

- case is not significant in field names
- to continue a field value, indent the next line
- to get a blank line in a field value, use an indented “`.`”

The syntax of the value depends on the field. Field types include:

`token`, *`filename`*, *`directory`*	Either a sequence of one or more non-space non-comma characters, or a quoted string in Haskell 98 lexical syntax. Unless otherwise stated, relative filenames and directories are interpreted from the package root directory.
`freeform`, *`URL`*, *`address`*	An arbitrary, uninterpreted string.
`identifier`	A letter followed by zero or more alphanumerics or underscores.

Modules and preprocessors

Haskell module names listed in the `exposed-modules` and `other-modules` fields may correspond to Haskell source files, i.e. with names ending in “`.hs`” or “`.lhs`”, or to inputs for various Haskell preprocessors. The simple build infrastructure understands the extensions “`.gc`” (**haddock** [http://www.haskell.org/greencard/]), “`.chs`” (**c2hs** [http://www.cse.unsw.edu.au/~chak/haskell/c2hs/]), “`.hsc`” (**hsc2hs**), “`.y`” and “`.ly`” (**happy** [http://www.haskell.org/happy/]), “`.x`” (**alex** [http://www.haskell.org/alex/]) and “`.cpphs`” (**cpphs** [http://www.haskell.org/cpphs/]). When building, Cabal will automatically run the appropriate preprocessor and compile the Haskell module it produces.

Some fields take lists of values, which are optionally separated by commas, except for the `build-depends` field, where the commas are mandatory.

Some fields are marked as required. All others are optional, and unless otherwise specified have empty default values.

2.1.1. Package properties

These fields may occur in the first stanza, and describe the package as a whole:

`name:` *`package-name`* (required)
: The unique name of the package (see Section 1, “Packages”), without the version number.

`version:` *`numbers`* (required)
: The package version number, usually consisting of a sequence of natural numbers separated by dots.

`cabal-version:` *`>, <=, etc. & numbers`*
: The version of Cabal required for this package. Use ONLY if this package requires a particular version of Cabal, since unfortunitely, early versions of Cabal do not recognize this field. List the field early in your .cabal file so that it will appear as a syntax error before any others.

`license:` *`identifier`* (default: `AllRightsReserved`)
: The type of license under which this package is distributed. License names are the constants of the `License` [../libraries/Cabal/Distribution-License.html#t:License] type.

`license-file:` *`filename`*
: The name of a file containing the precise license for this package.

`copyright:` *`freeform`*
: The content of a copyright notice, typically the name of the holder of the copyright on the package and the year(s) from which copyright is claimed.

`author:` *`freeform`*
: The original author of the package.

`maintainer:` *`address`*
: The current maintainer or maintainers of the package. This is an e-mail address to which users should send bug reports, feature requests and patches.

`stability:` *`freeform`*
: The stability level of the package, e.g. `alpha`, `experimental`, `provisional`, `stable`.

`homepage:` *`URL`*
: The package homepage.

`package-url:` *`URL`*
: The location of a source bundle for the package. The distribution should be a Cabal package.

`synopsis:` *`freeform`*
: A very short description of the package, for use in a table of packages. This is your headline, so keep it short (one line) but as informative as possible. Save space by not including the package name or saying it's written in Haskell.

`description:` *`freeform`*
: Description of the package. This may be several paragraphs, and should be aimed at a Haskell programmer who has never heard of your package before.

`category:` *`freeform`*
: A classification category for future use by the package catalogue *Hackage*. These categories have not yet been specified, but the upper levels of the module hierarchy make a good start.

`tested-with:` *`compiler list`*
: A list of compilers and versions against which the package has been tested (or at least built).

`build-depends:` *package list*
: A list of packages, possibly annotated with versions, needed to build this one, e.g. `foo > 1.2, bar`. If no version constraint is specified, any version is assumed to be acceptable.

`data-files:` *filename list*
: A list of files to be installed for run-time use by the package. This is useful for packages that use a large amount of static data, such as tables of values or code templates. For details on how to find these files at run-time, see Section 2.2, “Accessing data files from package code”.

`extra-source-files:` *filename list*
: A list of additional files to be included in source distributions built with **setup sdist** (see Section 3.10, “setup sdist”).

`extra-tmp-files:` *filename list*
: A list of additional files or directories to be removed by **setup clean** (see Section 3.8, “setup clean”). These would typically be additional files created by additional hooks, such as the scheme described in Section 2.3, “System-dependent parameters”.

2.1.2. Library

If the package contains a library, the first stanza should also contain the following field:

`exposed-modules:` *identifier list* (required if this package contains a library)
: A list of modules added by this package.

The first stanza may also contain build information fields (see Section 2.1.4, “Build information”) relating to the library.

2.1.3. Executables

Subsequent stanzas (if present) describe executable programs contained in the package, using the following fields, as well as build information fields (see Section 2.1.4, “Build information”).

`executable:` *freeform* (required)
: The name of the executable program.

`main-is:` *filename* (required)
: The name of the source file containing the `Main` module, relative to one of the directories listed in `hs-source-dirs`.

These stanzas may also contain build information fields (see Section 2.1.4, “Build information”) relating to the executable.

2.1.4. Build information

The following fields may be optionally present in any stanza, and give information for the building of the corresponding library or executable. See also Section 2.3, “System-dependent parameters” for a way to supply system-dependent values for these fields.

`buildable:` *Boolean* (default: `True`)
: Is the component buildable? Like some of the other fields below, this field is more useful with the slightly more elaborate form of the simple build infrastructure described in Section 2.3, “System-dependent parameters”.

`other-modules:` *identifier list*
: A list of modules used by the component but not exposed to users. For a library component, these would be hidden modules of the library. For an executable, these would be auxiliary modules to be linked with the file named in the

`main-is` field.

`hs-source-dirs:` *`directory list`* (default: "`.`")
: Root directories for the module hierarchy.

  For backwards compatibility, the old variant `hs-source-dir` is also recognized.

`extensions:` *`identifier list`*
: A list of Haskell extensions used by every module. Extension names are the constructors of the `Extension` [../libraries/Cabal/Language-Haskell-Extension.html#t:Extension] type. These determine corresponding compiler options. In particular, `CPP` specifies that Haskell source files are to be preprocessed with a C preprocessor.

  Extensions used only by one module may be specified by placing a `LANGUAGE` pragma in the source file affected, e.g.:

```
{-# LANGUAGE CPP, MultiParamTypeClasses #-}
```

Note

GHC doesn't support the `LANGUAGE` pragma yet.

`ghc-options:` *`token list`*
: Additional options for GHC. You can often achieve the same effect using the `extensions` field, which is preferred.

  Options required only by one module may be specified by placing an `OPTIONS_GHC` pragma in the source file affected.

`ghc-prof-options:` *`token list`*
: Additional options for GHC when the package is built with profiling enabled.

`hugs-options:` *`token list`*
: Additional options for Hugs. You can often achieve the same effect using the `extensions` field, which is preferred.

  Options required only by one module may be specified by placing an `OPTIONS_HUGS` pragma in the source file affected.

`nhc-options:` *`token list`*
: Additional options for nhc98. You can often achieve the same effect using the `extensions` field, which is preferred.

  Options required only by one module may be specified by placing an `OPTIONS_NHC` pragma in the source file affected.

`includes:` *`filename list`*
: A list of header files from standard include directories or those listed in `include-dirs`, to be included in any compilations via C. These files typically contain function prototypes for foreign imports used by the package.

`include-dirs:` *`directory list`*
: A list of directories to search for header files, both when using a C preprocessor and when compiling via C.

`c-sources:` *`filename list`*
: A list of C source files to be compiled and linked with the Haskell files.

  If you use this field, you should also name the C files in `CFILES` pragmas in the Haskell source files that use them, e.g.:

```
{-# CFILES dir/file1.c dir/file2.c #-}
```

These are ignored by the compilers, but needed by Hugs.

`raries:` *token list*	A list of extra libraries to link with.
`extra-lib-dirs:` *directory list*	A list of directories to search for libraries.
`cc-options:` *token list*	Command-line arguments to be passed to the C compiler. Since the arguments are compiler-dependent, this field is more useful with the setup described in Section 2.3, "System-dependent parameters".
`ld-options:` *token list*	Command-line arguments to be passed to the linker. Since the arguments are compiler-dependent, this field is more useful with the setup described in Section 2.3, "System-dependent parameters".
`frameworks:` *token list*	On Darwin/MacOS X, a list of frameworks to link to. See Apple's developer documentation for more details on frameworks. This entry is ignored on all other platforms.

2.2. Accessing data files from package code

The placement on the target system of files listed in the `data-files` field varies between systems, and in some cases one can even move packages around after installation (see Section 3.1.2.2, "Prefix-independence"). To enable packages to find these files in a portable way, Cabal generates a module called `Paths_`*`pkgname`* (with any hyphens in *`pkgname`* replaced by underscores) during building, so that it may be imported by modules of the package. This module defines a function

```
getDataFileName :: FilePath -> IO FilePath
```

If the argument is a filename listed in the `data-files` field, the result is the name of the corresponding file on the system on which the program is running.

2.3. System-dependent parameters

For some packages, especially those interfacing with C libraries, implementation details and the build procedure depend on the build environment. The simple build infrastructure can handle many such situations using a slightly longer `Setup.hs`:

```
import Distribution.Simple
main = defaultMainWithHooks defaultUserHooks
```

This program differs from `defaultMain` in two ways:

1. If the package root directory contains a file called `configure`, the configure step will run that. This `configure` program may be a script produced by the **autoconf** [http://www.gnu.org/software/autoconf/] system, or may be hand-written. This program typically discovers information about the system and records it for later steps, e.g. by generating system-dependent header files for inclusion in C source files and preprocessed Haskell source files. (Clearly this won't work for Windows without MSYS or Cygwin: other ideas are needed.)

2. If the package root directory contains a file called *`package`*`.buildinfo` after the configuration step, subsequent steps will read it to obtain additional settings for build information fields (see Section 2.1.4, "Build information"), to be merged with the ones given in the `.cabal` file. In particular, this file may be generated by the `configure` script mentioned above, allowing these settings to vary depending on the build environment.

   The build information file should have the following structure:

```
buildinfo

executable: name
buildinfo

executable: name
buildinfo

...
```

where each *buildinfo* consists of settings of fields listed in Section 2.1.4, “Build information”. The first one (if present) relates to the library, while each of the others relate to the named executable. (The names must match the package description, but you don't have to have entries for all of them.)

Neither of these files is required. If they are absent, this setup script is equivalent to `defaultMain`.

Example 4. Using autoconf

(This example is for people familiar with the **autoconf** [http://www.gnu.org/software/autoconf/] tools.)

In the X11 package, the file `configure.ac` contains:

```
AC_INIT([Haskell X11 package], [1.1], [libraries@haskell.org], [X11])

# Safety check: Ensure that we are in the correct source directory.
AC_CONFIG_SRCDIR([X11.cabal])

# Header file to place defines in
AC_CONFIG_HEADERS([include/HsX11Config.h])

# Check for X11 include paths and libraries
AC_PATH_XTRA
AC_TRY_CPP([#include <X11/Xlib.h>],,[no_x=yes])

# Build the package if we found X11 stuff
if test "$no_x" = yes
then BUILD_PACKAGE_BOOL=False
else BUILD_PACKAGE_BOOL=True
fi
AC_SUBST([BUILD_PACKAGE_BOOL])

AC_CONFIG_FILES([X11.buildinfo])
AC_OUTPUT
```

Then the setup script will run the `configure` script, which checks for the presence of the X11 libraries and substitutes for variables in the file `X11.buildinfo.in`:

```
buildable: @BUILD_PACKAGE_BOOL@
cc-options: @X_CFLAGS@
ld-options: @X_LIBS@
```

This generates a file `X11.buildinfo` supplying the parameters needed by later stages:

```
buildable: True
cc-options:  -I/usr/X11R6/include
ld-options:  -L/usr/X11R6/lib
```

The `configure` script also generates a header file `include/HsX11Config.h` containing C preprocessor defines recording the results of various tests. This file may be included by C source files and preprocessed Haskell source files in the package.

Note

Packages using these features will also need to list additional files such as `configure`, templates for `.buildinfo` files, files named only in `.buildinfo` files, header files and so on in the `extra-source-files` field, to ensure that they are included in source distributions. They should also list files and directories generated by **configure** in the `extra-tmp-files` field to ensure that they are removed by **setup clean**.

2.4. More complex packages

For packages that don't fit the simple schemes described above, you have a few options:

- You can customize the simple build infrastructure using *hooks*. These allow you to perform additional actions before and after each command is run, and also to specify additional preprocessors. See `UserHooks` in Distribution.Simple [../libraries/Cabal/Distribution-Simple.html] for the details, but note that this interface is experimental, and likely to change in future releases.

- You could delegate all the work to **make**, though this is unlikely to be very portable. Cabal supports this with a trivial setup library Distribution.Make [../libraries/Cabal/Distribution-Make.html], which simply parses the command line arguments and invokes **make**. Here `Setup.hs` looks like

  ```
  import Distribution.Make
  main = defaultMain
  ```

  The root directory of the package should contain a `configure` script, and, after that has run, a `Makefile` with a default target that builds the package, plus targets `install`, `register`, `unregister`, `clean`, `dist` and `docs`. Some options to commands are passed through as follows:

  - The `--with-hc`, `--with-hc-pkg`, `--prefix`, `--bindir`, `--libdir`, `--datadir` and `--libexecdir` options to the `configure` command are passed on to the `configure` script.

  - the `--destdir` option to the `copy` command becomes a setting of a `destdir` variable on the invocation of `make copy`. The supplied `Makefile` should provide a `copy` target, which will probably look like this:

    ```
    copy :
            $(MAKE) install prefix=$(destdir)/$(prefix) \
                            bindir=$(destdir)/$(bindir) \
                            libdir=$(destdir)/$(libdir) \
                            datadir=$(destdir)/$(datadir) \
                            libexecdir=$(destdir)/$(libexecdir)
    ```

- You can write your own setup script conforming to the interface of Section 3, “Building and installing a package”, possibly using the Cabal library for part of the work. One option is to copy the source of `Distribution.Simple`, and alter it for your needs. Good luck.

3. Building and installing a package

After you've unpacked a Cabal package, you can build it by moving into the root directory of the package and using the `Setup.hs` or `Setup.lhs` script there:
`runhaskell Setup.hs` [*command*] [*option*...]

where `runhaskell` might be **runhugs**, **runghc** or **runnhc**. The *command* argument selects a particular step in the build/install process. You can also get a summary of the command syntax with
`runhaskell Setup.hs --help`

Example 5. Building and installing a system package

```
runhaskell Setup.hs configure --ghc
runhaskell Setup.hs build
runhaskell Setup.hs install
```

The first line readies the system to build the tool using GHC; for example, it checks that GHC exists on the system. The second line performs the actual building, while the last both copies the build results to some permanent place and registers the package with GHC.

Example 6. Building and installing a user package

```
runhaskell Setup.hs configure --ghc --user --prefix=$HOME
runhaskell Setup.hs build
runhaskell Setup.hs install
```

The package may use packages from the user's package database as well as the global one (`--user`), is installed under the user's home directory (`--prefix`), and is registered in the user's package database (`--user`).

Example 7. Creating a binary package

When creating binary packages (e.g. for RedHat or Debian) one needs to create a tarball that can be sent to another system for unpacking in the root directory:

```
runhaskell Setup.hs configure --ghc --prefix=/usr
runhaskell Setup.hs build
runhaskell Setup.hs copy --destdir=/tmp/mypkg
(cd /tmp/mypkg; tar cf - .) | gzip -9 >mypkg.tar.gz
```

If the package contains a library, you need two additional steps:

```
runhaskell Setup.hs register --gen-script
runhaskell Setup.hs unregister --gen-script
```

This creates shell scripts `register.sh` and `unregister.sh`, which must also be sent to the target system. After unpacking there, the package must be registered by running the `register.sh` script. The `unregister.sh` script would be used in the uninstall procedure of the package. Similar steps may be used for creating binary packages for Windows.

The following options are understood by all commands:

`--help`, `-h` or `-?`	List the available options for the command.
`--verbose=`*n* or `-v`*n*	Set the verbosity level (0-5). The normal level is 1; a missing *n* defaults to 3.

The various commands and the additional options they support are described below. In the simple build infrastructure, any other options will be reported as errors, except in the case of the `configure` command.

3.1. setup configure

Prepare to build the package. Typically, this step checks that the target platform is capable of building the package, and discovers platform-specific features that are needed during the build.

The user may also adjust the behaviour of later stages using the options listed in the following subsections. In the simple build infrastructure, the values supplied via these options are recorded in a private file read by later stages.

If a user-supplied `configure` script is run (see Section 2.3, “System-dependent parameters” or Section 2.4, “More complex packages”), it is passed the `--with-hc`, `--with-hc-pkg`, `--prefix`, `--bindir`, `--libdir`, `--datadir` and `--libexecdir` options, plus any unrecognized options.

3.1.1. Programs used for building

The following options govern the programs used to process the source files of a package:

`--ghc` or `-g`, `--nhc` or `-n`, `--hugs`	Specify which Haskell implementation to use to build the package. At most one of these flags may be given. If none is given, the implementation under which the setup script was compiled or interpreted is used.

`path` or `-wpath`	Specify the path to a particular compiler. If given, this must match the implementation selected above. The default is to search for the usual name of the selected implementation.
pa `--with-hc-pkg=`*th*	Specify the path to the package tool, e.g. **ghc-pkg**.
p *a*	Specify the path to **haddock** [http://www.haskell.org/haddock/].
pat `--with-happy=`*h* `--with-haddock=`*h*	Specify the path to **happy** [http://www.haskell.org/happy/].
`--with-alex=`*path*	Specify the path to **alex** [http://www.haskell.org/alex/].
pa `--with-hsc2hs=`*th*	Specify the path to **hsc2hs**.
`--with-c2hs=`*path*	Specify the path to **c2hs** [http://www.cse.unsw.edu.au/~chak/haskell/c2hs/].

`=`*`path`*	Specify the path to **haddock** [http://www.haskell.org/greencard/].
`pat` `--with-cpphs=`*`h`*	Specify the path to **cpphs** [http://www.haskell.org/cpphs/].

3.1.2. Installation paths

The following options govern the location of installed files from a package:

`--prefix=`*`dir`*	The root of the installation, for example `/usr/local` on a Unix system, or `C:\Program Files` on a Windows system. The other installation paths are usually subdirectories of *`prefix`*, but they don't have to be.
`--bindir=`*`dir`*	Executables that the user might invoke are installed here.
`--libdir=`*`dir`*	Object-code libraries are installed here.

dir	A subdirectory of *libdir* in which libraries are actually installed. For example, in the simple build system on Unix, the default *libdir* is `/usr/local/lib`, and *libsubdir* contains the package identifier and compiler, e.g. `mypkg-0.2/ghc-6.4`, so libraries would be installed in `/usr/local/lib/mypkg-0.2/ghc-6.4`. Not all build systems make use of *libsubdir*, in particular the Distribution.Make [../libraries/Cabal/Distribution-Make.html] system does not.
di `--datadir=`*r*	Architecture-independent data files are installed here.

=*dir* — A subdirectory of *datadir* in which data files are actually installed. This option is similar to `--libsubdir` in that not all build systems make use of it.

`=dir`	Executables that are not expected to be invoked directly by the user are installed here.

3.1.2.1. Paths in the simple build system

For the simple build system, the following defaults apply:

Option	**Windows Default**	**Unix Default**
`--prefix`	`C:\Program Files`	`/usr/local`
`--bindir`	`$prefix\$pkgid`	`$prefix/bin`
`--libdir`	`$prefix\Haskell`	`$prefix/lib`
`--libsubdir` (Hugs)	`hugs\packages\$pkg`	`hugs/packages/$pkg`
`--libsubdir` (others)	`$pkgid\$compiler`	`$pkgid/$compiler`
`--datadir` (executable)	`$prefix`	`$prefix/share`
`--datadir` (library)	`C:\Program Files\Common Files`	`$prefix/share`
`--datasubdir`	`$pkgid`	`$pkgid`
`--libexecdir`	`$prefix\$pkgid`	`$prefix/libexec`

The following strings are substituted into directory names:

`$prefix`	The value of *`prefix`*
`$pkgid`	The full package identifier, e.g. `pkg-0.1`
`$compiler`	The compiler and version, e.g. `ghc-6.4.1`
`$pkg`	The name of the package only
`$version`	The version of the package

3.1.2.2. Prefix-independence

On Windows (and perhaps other OSs), it is possible to query the pathname of the running binary. This means that we can construct an installable executable package that is independent of its absolute install location. The executable can find its auxiliary files by finding its own path and knowing the location of the other files relative to *`bindir`*. Prefix-independence is particularly useful: it means the user can choose the install location (i.e. the value of *`prefix`*) at install-time, rather than having to bake the path into the binary when it is built.

In order to achieve this, we require that for an executable on Windows, all of *`bindir`*, *`libdir`*, *`datadir`* and *`libexecdir`* begin with `$prefix`. If this is not the case then the compiled executable will have baked in all absolute paths.

The application need do nothing special to achieve prefix-independence. If it finds any files using `getDataFileName` and the other functions provided for the purpose (see Section 2.2, “Accessing data files from package code”), the files will be accessed relative to the location of the current executable.

A library cannot (currently) be prefix-independent, because it will be linked into an executable whose filesystem location bears no relation to the library package.

3.1.3. Miscellaneous options

`--user`
: Allow dependencies to be satisfied by the user package database, in addition to the global database.

  This also implies a default of `--user` for any subsequent `install` command, as packages registered in the global database should not depend on packages registered in a user's database.

`--global`
: (default) Dependencies must be satisfied by the global package database.

`--enable-library-profiling` or `-p`
: Request that an additional version of the library with profiling features enabled be built and installed (only for implementations that support profiling).

`--disable-library-profiling`
: (default) Do not generate an additional profiling version of the library.

`--enable-executable-profiling`
: Any executables generated should have profiling enabled (only for implementations that support profiling). For this to work, all libraries used by these executables must also have been built with profiling support.

`--disable-executable-profiling`
: (default) Do not enable profiling in generated executables.

In the simple build infrastructure, an additional option is recognized:

=*dir* or -b*dir* Specify the directory into which the package will be built (default: `dist/build`).

3.2. setup build

Perform any preprocessing or compilation needed to make this package ready for installation.

3.3. setup haddock

Build the interface documentation for a library using **haddock** [http://www.haskell.org/haddock/].

3.4. setup install

Copy the files into the install locations and (for library packages) register the package with the compiler, i.e. make the modules it contains available to programs.

The install locations are determined by options to `setup configure` (see Section 3.1.2, “Installation paths”).

This command takes the following options:

`--global` Register this package in the system-wide database. (This is the default, unless the `--user` option was supplied to the `configure` command.)

`--user` Register this package in the user's local package database. (This is the default if the `--user` option was supplied to the `configure` command.)

3.5. setup copy

Copy the files without registering them. This command is mainly of use to those creating binary packages.

This command takes the following option:

`=path` Specify the directory under which to place installed files. If this is not given, then the root directory is assumed.

3.6. setup register

Register this package with the compiler, i.e. make the modules it contains available to programs. This only makes sense for library packages. Note that the `install` command incorporates this action. The main use of this separate command is in the post-installation step for a binary package.

This command takes the following options:

`--global` Register this package in the system-wide database. (This is the default.)

`--user` Register this package in the user's local package database.

`--gen-script` Instead of registering the package, generate a script containing commands to perform the registration. On Unix, this file is called `register.sh`, on Windows, `register.bat`. This script might be included in a binary bundle, to be run after the bundle is unpacked on the target system.

3.7. setup unregister

Deregister this package with the compiler.

This command takes the following options:

`--global` Deregister this package in the system-wide database. (This is the default.)

`--user` Deregister this package in the user's local package database.

`--gen-script` Instead of deregistering the package, generate a script containing commands to perform the deregistration. On Unix, this file is called `unregister.sh`, on Windows, `unregister.bat`. This script might be included in a binary bundle, to be run on the target system.

3.8. setup clean

Remove any local files created during the `configure`, `build`, `haddock`, `register` or `unregister` steps, and also any files and directories listed in the `extra-tmp-files` field.

3.9. setup test

Run the test suite specified by the `runTests` field of `Distribution.Simple.UserHooks`. See Distribution.Simple [../libraries/Cabal/Distribution-Simple.html] for information about creating hooks and using `defaultMainWithHooks`.

3.10. setup sdist

Create a system- and compiler-independent source distribution in a file *package-version*`.tar.gz` in the `dist` subdirectory, for distribution to package builders. When unpacked, the commands listed in this section will be available.

The files placed in this distribution are the package description file, the setup script, the sources of the modules named in the package description file, and files named in the `license-file`, `main-is`, `c-sources`, `data-files` and `extra-source-files` fields.

This command takes the following option:

`--snapshot` Append today's date (in *YYYYMMDD* form) to the version number for the generated source package. The original package is unaffected.

4. Known bugs and deficiencies

All these should be fixed in future versions:

- The scheme described in Section 2.3, "System-dependent parameters" will not work on Windows without MSYS or Cygwin.
- Cabal has some limitations both running under Hugs and building packages for it:
  - Cabal requires the latest release (Mar 2005).
  - It doesn't work with Windows.
  - There is no `hugs-pkg` tool.
- Though the library runs under Nhc98, it cannot build packages for Nhc98.

Please report any other flaws to <`libraries@haskell.org`>.