

# A proposal for scsh packages

Michel Schinz

August 20, 2005

## 1 Introduction

The aim of the following proposal is to define a standard for the packaging, distribution, installation, use and removal of libraries for scsh. Such packaged libraries are called *scsh packages* or simply *packages* below.

This proposal attempts to cover both libraries containing only Scheme code and libraries containing additional C code. It does not try to cover applications written in scsh, which are currently considered to be outside of its scope.

### 1.1 Package identification and naming

Packages are identified by a globally-unique name. This name should start with an ASCII letter (a-z or A-Z) and should consist only of ASCII letters, digits or hyphens '-'. Package names are case-sensitive, but there should not be two packages with names which differ only by their capitalisation.

**Rationale** This restriction on package names ensures that they can be used to name directories on current operating systems.

Several versions of a given package can exist. A version is identified by a sequence of non-negative integers. Versions are ordered lexicographically.

A version has a printed representation which is obtained by separating (the printed representation of) its components by dots. For example, the printed representation of a version composed of the integer 1 followed by the integer 2 is the string 1 . 2. Below, versions are usually represented using their printed representation for simplicity, but it is important to keep in mind that versions are sequences of integers, not strings.

A specific version of a package is therefore identified by a name and a version. The *full name* of a version of a package is obtained by concatenating:

- the name of the package,
- a hyphen '-',

- the printed representation of the version.

In what follows, the term *package* is often used to designate a specific version of a package, but this should be clear from the context.

## 2 Distributing packages

Packages are usually distributed as a stand-alone archive containing the code and documentation to install, as well as a file describing the installation procedure. However, it is also possible to distribute the scsh-specific installation procedure separately from the main code and documentation. This is typically useful to turn general Scheme libraries into scsh packages.

### 2.1 Stand-alone packages

Stand-alone packages are distributed in a single `tar` archive, which can optionally be compressed by `gzip` or `bzip2`. The name of the archive is composed by appending:

- the full name of the package,
- the string `.tar` indicating that it's a `tar` archive,
- either the string `.gz` if the archive is compressed using `gzip`, or the string `.bz2` if the archive is compressed using `bzip2`, or nothing if the archive is not compressed.

The archive of a stand-alone package is organised so that it contains one top-level directory whose name is the full name of the package. This directory is called the *package unpacking directory*. All the files belonging to the package are stored below it.

The unpacking directory contains at least the following files:

**pkg-def.scm** a Scheme file containing the installation procedure for the package (see § 5),

**README** a textual file containing a short description of the package,

**COPYING** a textual file containing the license of the package.

### 2.2 Split packages

Split packages are distributed as two separate archives. The first one, called the *upstream archive*, contains the code and documentation of the package. The second one, called the *downstream archive*, contains the scsh-specific installation procedure for the package.

Ideally, the upstream archive should follow the rules laid out in the previous section for stand-alone archives. Most of the time, however, the author of the upstream package

has no reason to be aware of the current policy, or simply to adhere to it. The name and layout of the downstream archive should therefore mirror the one of the upstream archive, as follows:

1. the name of the downstream archive is composed by appending:
  - the string `pkg-` (the three letters ‘p’, ‘k’ and ‘g’ followed by an underscore character),
  - the name of the upstream archive, including its version but excluding extensions indicating the archive type and/or compression method,
  - an underscore character,
  - a single number indicating the version of the downstream archive,
  - the string `.tar` indicating that it’s a `tar` archive,
  - either the string `.gz` if the archive is compressed using `gzip`, or the string `.bz2` if the archive is compressed using `bzip2`, or nothing if the archive is not compressed.
2. the downstream archive should contain at least the file `pkg-def.scm` placed in a directory with the same name as the “main” directory of the upstream archive.

These conventions ensure that split packages behave almost like stand-alone packages after the two archives have been unpacked at the same location.

## 3 Downloading and installing packages

A package can be installed on a target machine by downloading its archive, expanding it and finally running the installation script located in the unpacking directory.

### 3.1 Layouts

The installation script installs files according to some given *layout*. A layout maps abstract *locations* to concrete directories on the target machine. For example, a layout could map the abstract location `doc`, where documentation is stored, to the directory `/usr/local/share/doc/my_package`.

Currently, the following abstract locations are defined:

**base** The “base” location of a package, where the package loading script `load.scm` resides.

**active** Location containing a symbolic link, with the same name as the package (excluding the version), pointing to the base location of the package. This link is used to designate the *active* version of a package—the one to load when a package is requested by giving only its name, without an explicit version.



### 3.1.1 Predefined layouts

Every installation script comes with a set of predefined layouts which serve different aims. They are described below

The directories to which these layouts map locations often have a name which includes the current version of `scsh` and/or the full name of the package. In what follows, the notation `<version>` represents the printed representation of the first two components of `scsh`'s version (e.g. `0.6` for `scsh v0.6.x`). The notation `<pkg_fname>` represents the *full* name of the package being installed.

**The `scsh` layout** The `scsh` layout is the default layout. It maps all locations to sub-directories of a single directory, called the package installation directory, which contains nothing but the files of the package being installed. Its name is simply the full name of the package in question, and it resides in the `prefix` directory.

The `scsh` layout maps locations as given in the following table:

Location	Directory (relative to prefix)
base	<code>&lt;version&gt;/&lt;pkg_fname&gt;</code>
active	<code>&lt;version&gt;</code>
scheme	<code>&lt;version&gt;/&lt;pkg_fname&gt;/scheme</code>
lib	<code>&lt;version&gt;/&lt;pkg_fname&gt;/lib</code>
doc	<code>&lt;version&gt;/&lt;pkg_fname&gt;/doc</code>
misc-shared	<code>&lt;version&gt;/&lt;pkg_fname&gt;</code>

This layout is well suited for installations performed without the assistance of an additional package manager, because it makes many common operations easy. For example, finding to which package a file belongs is trivial, as is the removal of an installed package.

**The `fhs` layout** The `fhs` layout maps locations according to the File Hierarchy Standard (FHS, see <http://www.pathname.com/fhs/>), as follows:

Location	Directory (relative to prefix)
base	<code>share/scsh-&lt;version&gt;/modules/&lt;pkg_fname&gt;</code>
active	<code>share/scsh-&lt;version&gt;/modules</code>
scheme	<code>share/scsh-&lt;version&gt;/modules/&lt;pkg_fname&gt;/scheme</code>
lib	<code>lib/scsh-&lt;version&gt;/modules/&lt;pkg_fname&gt;</code>
doc	<code>share/doc/scsh-&lt;version&gt;/&lt;pkg_fname&gt;</code>
misc-shared	<code>share/scsh-&lt;version&gt;/modules/&lt;pkg_fname&gt;</code>

The main advantage of this layout is that it adheres to the FHS standard, and is therefore compatible with several packaging policies, like Debian's, Fink's and others. Its main drawback is that files belonging to a given package are scattered, and therefore hard to find when removing or upgrading a package. Its use should therefore be considered only if third-party tools are available to track files belonging to a package.

## 3.2 Installation procedure

Packages are installed using the `scsh-install-pkg` script, which is part of the installation library. This script must be given the name of the prefix using the `--prefix` option. It also accepts the following options:

<code>--layout name</code>	Specifies the layout to use (see § 3.1.1).
<code>--verbose</code>	Print messages about what is being done.
<code>--dry-run</code>	Print what actions would be performed to install the package, but do not perform them.
<code>--inactive</code>	Do not activate package after installing it.
<code>--non-shared-only</code>	Only install platform-dependent files, if any.
<code>--force</code>	Overwrite existing files during installation.
<code>--no-user-defaults</code>	Don't read user defaults in <code>.scsh-pkg-defaults.scm</code> (see § 3.2.1).

A few, more advanced options exist and are documented in section 6.

### 3.2.1 User preferences

Users can store default values for the options passed to the installation script by storing them in a file called `.scsh-pkg-defaults.scm` residing in their home directory. This file must contain exactly one Scheme expression whose value is an association list. The keys of this list, which must be symbols, identify options and the values specify the default value for these options. The contents of this file is implicitly quasi-quoted.

The values stored in this file override the default values of the options, but they are in turn overridden by the values specified on the command line of the installation script. Furthermore, it is possible to ask for this file to be completely ignored by passing the `--no-user-defaults` option to the installation script.

**Example A** `.scsh-pkg-defaults.scm` file containing the following:

```
;; Default values for scsh packages installation
((layout . "fhs")
 (prefix . "/usr/local/share/scsh/modules")
 (verbose . #t))
```

specifies default values for the `--layout`, `--prefix` and `--verbose` options.

## 4 Using packages

To use a package, its *loading script* must be loaded in Scheme 48's `exec` package. The loading script for a package is a file written in the Scheme 48 `exec` language, whose name is `load.scm` and which resides in the base location.

To load this file, one typically uses `scsh's -lel` option along with a properly defined `SCSH_LIB_DIRS` environment variable.

`Scsh` has a list of directories, called the library directories, in which it looks for files to load when the options `-ll` or `-lel` are used. This list can be given a default value during `scsh's` configuration, and this value can be overridden by setting the environment variable `SCSH_LIB_DIRS` before running `scsh`.

In order for `scsh` to find the package loading scripts, one must make sure that `scsh's` library search path contains the names of all `active` locations which containing packages.

The names of these directories should not end with a slash `'/'`, as this forces `scsh` to search them recursively. This could *drastically* slow down `scsh` when looking for packages.

**Example** Let's imagine a machine on which the system administrator installs `scsh` packages according to the `fhs` layout in prefix directory `/usr/local`. The `active` location for these packages corresponds to the directory `/usr/local/share/scsh-0.6/modules`, according to section 3.1.1.

Let's also imagine that there is a user called `john` on this machine, who installs additional `scsh` packages for himself in his home directory, using `/home/john/scsh` as a prefix. To ease their management, he uses the `scsh` layout. The `active` location for these packages corresponds to the directory `/home/john/scsh/0.6`, according to section 3.1.1.

In order to be able to use `scsh` packages installed both by the administrator and by himself, user `john` needs to put both `active` directories in his `SCSH_LIB_DIRS` environment variable. The value of this variable will therefore be:

```
"/usr/local/share/scsh-0.6/modules" "/home/john/scsh/0.6"
```

Now, in order to use packages `foo` and `bar` in one of his script, user `john` just needs to load their loading script using the `-lel` option when invoking `scsh`, as follows:

```
-lel foo/load.scm -lel bar/load.scm
```

## 5 Authoring packages

Once the Scheme and/or C code for a package has been written, the last step in turning it into a standard package as defined by this proposal is to write the installation script.

This script could be written fully by the package author, but in order to simplify this task a small `scsh` installation framework is provided. This framework must be present on the host system before a `scsh` package can be installed.

As explained above, when the `scsh-install-pkg` script is invoked, it launches `scsh` on the main function of the installation library, which does the following:

1. parse the command line arguments (e.g the `--prefix` option),
2. load the package definition file, a (Scheme) file called `pkg-def.scm`, which is supplied by the package author and which contains one or several package definition statements, and
3. install the packages which were defined in the previous step.

Most package definition files should contain a single package definition, but the ability to define several packages in one file can sometimes be useful.

The main job of the package author is therefore to write the package definition file, `pkg-def.scm`. This file is mostly composed of a package definition statement, which specifies the name, version and installation code for the package. The package definition statement is expressed using the `define-package` form, documented in the next section.

## 5.1 Installation library

### 5.1.1 Package definition

`(define-package name version extension body ...)` *(syntax)*

Define a package to be installed. *Name* (a string) is the package name, *version* (a list of integers) is its version, *extensions* is a list of extensions (see below), and *body* is the list of statements to be evaluated in order to install the package.

The installation statements typically use functions of the installation library in order to install files in their target location. The available functions are presented below.

*Extensions* consists in a list of lists, each one starting with a symbol identifying the extension, possibly followed by extension-specific parameters. It is used to specify various parameters, which are usually optional. Currently, the following extensions are defined:

**install-lib-version** specifies the version of the installation library that this package definition requires. The version is specified as a list composed of *exactly two* integers, giving the major and minor version number of the library. Before installing a package, this version requirement is checked and installation aborts if the installation library does not satisfy it.<sup>1</sup>It is strongly recommended that package authors provide this information, as it makes it possible to provide helpful error messages to users.

**options** enables the script to define additional command-line options. It accepts nine parameters in total, with the last three being optional. The description of these parameters follows, in the order in which they should appear:

---

<sup>1</sup>Version  $(i_1 i_2)$  of the installation library satisfies a requirement  $(r_1 r_2)$  if and only if both major numbers are equal, and the minor number of the installation library is greater or equal to the minor requirement. In other words, iff  $i_1 = r_1$  and  $i_2 \geq r_2$ .



*name* (a symbol) is the name of the option, without the initial double hyphen (--),  
*help-text* (a string) describes the option for the user,  
*arg-help-text* (a string) describes the option's argument (if any) for the user,  
*required-arg?* (a boolean) says whether this option requires an argument or not,  
*optional-arg?* (a boolean) says whether this option's argument can be omitted or not,  
*default* (anything) is the default value for the option,  
*parser* (a function from string to anything) parses the option, i.e. turns its string representation into its internal value,  
*unparser* (a function from anything to string) turns the internal representation of the option into a string,  
*transformer* is a function taking the current value of the option, the value given by the user and returning its new value.

By default, *parser* and *unparser* are the identity function, and *transformer* is a function which takes two arguments and returns the second (i.e. the current value of the option is simply replaced by the one given).

### 5.1.2 Content installation

(install-file *file location [target-dir]*) (procedure)

Install the given *file* in the sub-directory *target-dir* (which must be a relative directory) of the given *location*. *Target-dir* is `.` by default.

If the directory in which the file is about to be installed does not exist, it is created along with all its parents, as needed. If *file* is a string, then the installed file will have the same name as the original one. If *file* is a pair, then its first element specifies the name of the source file, and its second element the name it will have once installed. The second element must be a simple file name, without any directory part.

(install-files *file-list location [target-dir]*) (procedure)

Like `install-file` but for several files, which are specified as a list. Each element in the list can be either a simple string or a pair, as explained above.

(install-directory *directory location [target-dir]*) (procedure)

Install the given *directory* and all its contents, including sub-directories, in sub-directory *target-dir* of *location*. This is similar to what `install-file` does, but for complete hierarchies.

Notice that *directory* will be installed as a sub-directory of *target-dir*.

(install-directories *dir-list location [target-dir]*) (procedure)

Install several directories in one go.

(install-directory-contents *directory location [target-dir]*) (procedure)

Install the contents of the given *directory* in sub-directory *target* of *location*.

(install-string *string location [target-dir]*) (procedure)

Install the contents of *string* in sub-directory *target-dir* of *location*.

### 5.1.3 Queries

(get-directory *location install?*) (procedure)

Get the absolute name of the directory to which the current layout maps the abstract *location*. If *install?* is true, the directory is the one valid during installation; If it is false, the directory is the one valid after installation, that is when the package is later used.

The distinction between installation-time and usage-time directories is necessary to support staged installation, as performed by package managers like Debian's APT.

(get-option-value *option*) (procedure)

Return the value of the given command-line *option* (a symbol). This can be used to get the value of predefined options (like `--dry-run`) or package-specific options.

(phase-active? *phase*) (procedure)

Return true iff the given *phase* is active, that is if the steps associated with it should be performed. *Phase* should be one of the following symbols: `build`, `build-clean` or `install`, designating the corresponding phase (see section 6).

### 5.1.4 Load script generation

(with-output-to-load-script\* *thunk*) (procedure)

Evaluate *thunk* with the current output opened on the loading script of the current package. If this script was already existing, its previous contents is deleted.

(with-output-to-load-script *body ...*) (syntax)

Syntactic sugar for `with-output-to-load-script*`.

(write-to-load-script *s-expression*) (procedure)

Pretty-print the *s-expression* to the loading script of the current package. If this script was already existing, its previous contents is deleted.

**Example** A typical package definition file for a simple package called `pkg` whose version is 1.2 could look like this:

```
(define-package "pkg" (1 2) ())
```

```
(install-file "load.scm" 'base)
(install-directory-contents "scheme" 'scheme)
(install-file ("LICENSE" . "COPYING") 'doc)
(install-directory-contents "doc" 'doc)
```

With such a definition, invoking the installation script with `/usr/local/` as prefix and `fhs` as layout has the following effects:

1. The base directory `/usr/local/share/scsh/modules/pkg-1.2` is created and file `load.scm` is copied to it.
2. All the contents of the directory called `scheme` is copied to directory `/usr/local/share/scsh/modules/pkg-1.2/scheme` which is created before, if needed.
3. File `LICENSE` is copied to directory `/usr/local/share/doc/pkg-1.2/` with name `COPYING`.
4. All the contents of the directory called `doc` is copied to directory `/usr/local/share/doc/pkg-1.2/`
5. The package is activated by creating a symbolic link with name `/usr/local/share/scsh/modules/pkg` pointing to `./pkg-1.2`

### 5.1.5 Miscellaneous

A few functions which are not specifically related to installation are provided, as they can sometimes be useful when writing installation scripts. They are documented below.

`(parent-directory dir)` *(procedure)*

Returns the parent directory of *dir*. Notice that a trailing slash is always ignored by this function, so that the parent directory of both `/tmp/dir` and `/tmp/dir/` is `/tmp/`.

`(create-directory&parents dir)` *(procedure)*

Similar to `scsh's create-directory`, but also create all parent directories which do not exist yet.

`(relative-file-name fname [dir])` *(procedure)*

Return the name of file *fname* relative to *dir*, which defaults to the current directory.

`(paths->file-name path ...)` *(procedure)*

Similar to `scsh's path-list->file-name` except that all arguments are taken to form the final path. This function has no equivalent for `path-list->file-name's` optional argument.

## 5.2 Packages containing C code (for shared libraries)

Packages containing C code are more challenging to write, since all the problems related to C's portability and incompatibilities between the APIs of the various platforms have to be accounted for. Fortunately, the GNU Autoconf system simplifies the management of these problems, and authors of scsh packages containing C code are strongly encouraged to use it.

## 6 Packaging packages

Most important Unix systems today have one (or several) package management systems which ease the installation of packages on a system. In order to avoid confusion between these packages and the scsh packages discussed above, they will be called *system packages* in what follows.

It makes perfect sense to provide system packages for scsh packages. System packages should as much as possible try to use the standard installation script described above to install scsh packages. In order to make this as easy as possible, the installation script recognizes the following additional options:

```
--dest-dir dir    Specify a prefix for all installed files
--phases phases   Only perform some phases of installation.
```

The `--dest-dir` option plays the same role as the `DESTDIR` variable which is typically given to make `install`. Its value, which is the empty string by default, is prepended to all installation paths. This is used to perform staged installations, during which the package is first installed into a temporary directory and then moved to its final location by some external tool.

The `--phases` option is used to perform only some steps of the whole installation process. Currently, three phases are defined: `build`, `build-clean` and `install`.

The `build` phase is the one during which the operations required to build the package are performed. For scsh packages containing only Scheme code, this phase usually does nothing. For scsh packages containing some C code, this phase is the one during which the C code is compiled.

The `build-clean` phase is the one during which all the files created during the `build` phase are removed.

The `install` phase is the one during which the files are copied to their final location.

Using the `--phases` option, each phase can happen during a separate run of the installation script. This makes it possible, for example, to perform the `build` phase as one user, and the `install` phase as another one (e.g. `root`). In order for this to work, the installation script must be invoked as follows:

1. first with the `--phases build` option and maybe other options,

2. then with the *exact same options* as in the first step, except that `install` has to be given to `--phases` instead of `build`.

The behaviour of the installation script is not defined if these conditions are violated.

## 7 Acknowledgments

Discussions with Andreas Bernauer, Anthony Carrico, David Frese, Friedrich Delgado Friedrichs, Martin Gasbichler, Eric Knauel, Daniel Kobras, Lionel Elie Mamane, and Jorgen Schäfer greatly helped the design of this proposal. Mark Sapa started everything by asking for a Fink package for `sunet` and `sunterlib`.