Weierstraß-Institut
für Angewandte Analysis und Stochastik

im Forschungsverbund Berlin e.V.

Technical Report ISSN 1618 – 7776

# WPM Package Manager Version 1.0
# - Software Documentation -

Timo Streckenbach

Weierstrass Institute for Applied Analysis and Stochastics

E-Mail: strecken@wias-berlin.de

submitted: 11th March 2010

No. 12
Berlin 2010

*Key words and phrases.* Package manager; software versioning; dependencies; source package; virtual package; compiler flags; database; root-permissions; portability; Mixing FORTRAN, C and C++.

Edited by
Weierstraß-Institut für Angewandte Analysis und Stochastik (WIAS)
Mohrenstraße 39
10117 Berlin
Germany

| | |
|---|---|
| Fax: | + 49 30 2044975 |
| E-Mail: | preprint@wias-berlin.de |
| World Wide Web: | http://www.wias-berlin.de/ |

# Contents

# 1 Introduction

WPM is a command-line tool designed to support build and installation facilities. It is implemented as a collection of script files, written in Bourne Shell syntax. For the sake of portability the code takes care about the common pitfalls of shell programming [s. *autoconf*, Portable Shell Programming].

**NOTE** This software comes with absolutely no warranty!

## 1.1 Main goals

This package manager is mainly designed to assist software developers to handle *source packages*[1] and must consider the following basic requirements.

The package manager itself should

- be easy to install,
- run on common systems,
- work without root-permissions,
- contain a database of build commands and flags for FORTRAN, C and C++,
- be able to handle different versions of a software packages (software versioning),
- consider dependencies of software packages.

## 1.2 Installation of the package manager

The official location of WPM is the URL

`http://www.wias-berlin.de/software/pdelib/download/wpm-1.0.tar.gz`

As an alternative, a checkout from the *subversion* repository is possible with the following command

```
$ svn co -r5091 \
 svn+ssh://hilbert.wias-berlin.de/Home/darcy/svnroot/trunk/wpm wpm
```

Ensure that the user can execute the program `wpm` , `wpm-build` and `wpm-bundle` contained in the wpm folder.

[1]Details about source packages in Chapter 4

## 1.3 Prepare examples

Inside the WPM distribution exists a directory `examples`. To generate the example packages `checklib-1.0-1.wpm`, `check-1.0-1.wpm` and `chksum-1.0-1.wpm`[2], change into this directory and start the setup step (s. Section 4.2 for details) with

```
$ ./wpm_setup.sh
```

To discover the content and dependencies of a package compare Section 3.2.

## 1.4 Package, version and release

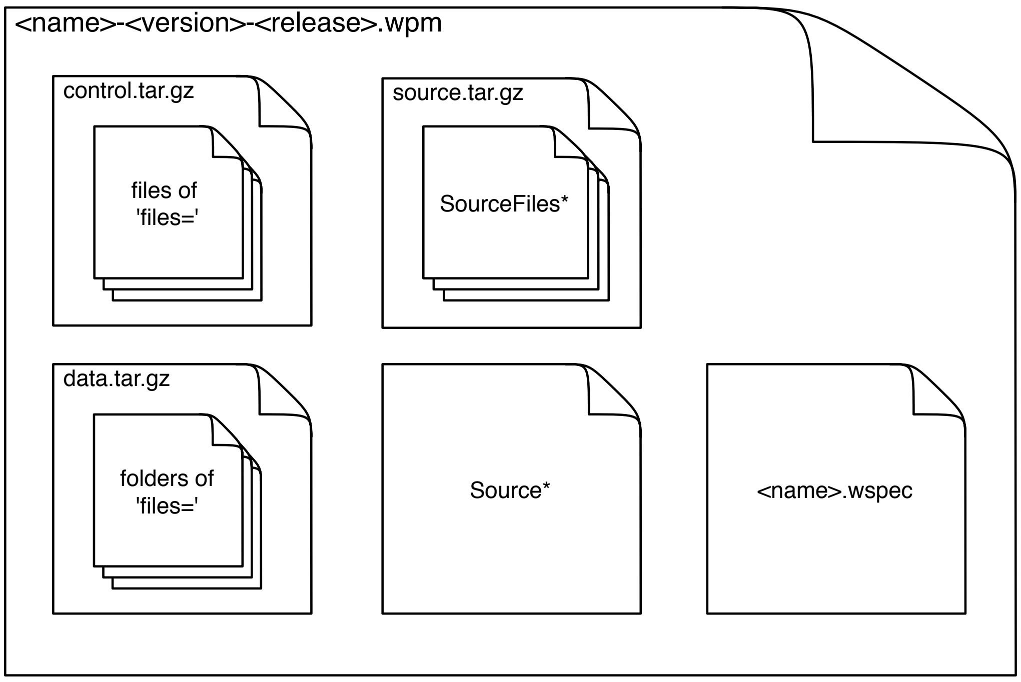
The following terms have a special meaning and are explained here:

| | |
|---|---|
| `name` | Pure name of a package (for instance `checklib`) |
| `version` | Version number of a package (for instance `1.0`) |
| `release` | Release number of a package (for instance `1`) |
| `label` | Composed from `name`, `version`, `release` with separator "`-`" |

The label identifies a package in the database.

The following figure shows the details of the WPM package file format. More details about the `wspec` header variables `Source` and `SourceFiles` can be found in Section 4.3. The WPM package represents a ordinary uncompressed *tar* file created with *tar -c*. And the archives inside the WPM are still compressed
with *tar -c* | *gzip -c*.

[2]This software is also an integrated part of WPM which generates checksums (see 4.5).



* Optional files

## 2 Build source code with `wpm-build`

The program `wpm-build` is designed to collect knowledge about compilers (FORTRAN, C and C++) and build tools to create static libraries and link binaries.

The knowledge database of available vendors (s. 2.2) should be able to compile

- FORTRAN, C and C++ code up to optimization level O3 (without aggressive optimizations which have the potential to alter the semantics of the program);
- FORTRAN, C and C++ code with OpenMP directives and clauses;
- ANSI C code;
- C++ templates including STL;
- C++ with with large file support[3] (`_FILE_OFFSET_BITS=64`);

[3]Relevant on 32 bit systems.

## 2.1 Mixing FORTRAN, C and C++

Integrating C/C++ and FORTRAN into a single executable relies on knowing how to interface the function calls, argument list and global data structures so the symbols match in the object code during linking. The entry point names for some FORTRAN compilers have an underscore appended to the name and case in FORTRAN is NOT preserved and mostly is represented in lower case in the object file.

### 2.1.1 Mangle flag

When using `wpm-build` for instance in configure scripts the following subcommand maybe useful

```
$ wpm-build flags --fmangle
```

Possible results of this call

```
lowercase_nounderscore_noextraunderscore
lowercase_nounderscore_extraunderscore
lowercase_underscore_noextraunderscore
lowercase_underscore_extraunderscore
uppercase_nounderscore_noextraunderscore
uppercase_nounderscore_extraunderscore
uppercase_underscore_noextraunderscore
uppercase_underscore_extraunderscore
```

### 2.1.2 Generate C/C++ macros

The subprogram `wpm-build header` returns C/C++ macros which allow to write portable external function declarations.

Example

```
$ wpm-build header
#define F77_FUNC(name,NAME) name ## _
#define F77_FUNC_(name,NAME) name ## __
```

And for example the declarations in a C++ program of two FORTRAN functions `fortran_func1(i)` and `fortranfunc2(i)` may implemented as follows.

```
extern "C" void F77_FUNC_(fortran_func1,FORTRAN_FUNC1)(int* i);
extern "C" void F77_FUNC(fortranfunc2,FORTRANFUNC2)(int* i);
```

Some compilers will add an extra underscore, if the name of the FORTRAN function contains an underscore.

## 2.2 Available vendors

With the next subcommand a list of available vendors for the current system (`WPM_HOST`) is accessible.

```
$ wpm-build list
WPM_HOST   : powerpc-apple-darwin8.11.0
WPM_VENDOR :
Available vendors : gnu ibm
Currently selected: gnu
```

If the variable `WPM_VENDOR` is empty, `wpm-build` will select the default for this system. The defaults for the systems are defined in `sys/*/default`. User extensions are described in Section 2.8. The following table contains the list of available vendors.

| system | vendor | parallel | comment |
|---|---|---|---|
| ia32 linux | gnu | no | g++, gcc, gfortran 4.1.0 (SUSE Linux) |
| | gnu77comp | no | |
| | gnu77 | no | |
| | intel9 | yes | icc 9.1.038, ifort 9.1.032 |
| x86 64 linux | gnu | no | g++, gcc, gfortran 4.1.0 (SUSE Linux) |
| | intel9e | yes | Intel EMT 64bit with openmp |
| aix | default | yes | multi-threaded executable with xlc++_r, xlc_r, xlf_r; AIX Version 5.3, IBM(R) XL C/C++ Enterprise Edition V8.0 |
| intel darwin | gnuif | no | |
| | gnu42 | yes | g++-4, gcc-4, gfortran 4.2.0 |
| | intel64 | no | |
| powerpc darwin | gnu | no | g++, gcc 4.0.1, g77 3.4.3 |
| | ibm | yes | IBM(R) XL C/C++ Advanced Edition Version 6.0 |
| mingw | gnu | no | |
| mips irix | default | yes | MIPSpro Compilers: Version 7.4 |
| alpha osf | default | yes | Compaq C V6.5-011 on Compaq Tru64 UNIX V5.1B, Compiler Driver V6.5-003; C++ link problems with manual instanciated template members |
| solaris2 | default | | prepared |
| cygwin | gnu | no | neglected |
| ia64 | gnu | no | prepared |
| | intel | | prepared |

## 2.3 Flags

Getting build tools and flags in a granular way from `wpm-build` with

- `wpm-build flags [Options] <Flags>...`

### 2.3.1 Options

| | |
|---|---|
| `...--help` | list of flags |
| `...-opt` | select optimized flags (default and overwrites `WPM_CONFIG`) |
| `...-dbg` | select debug flags (overwrites `WPM_CONFIG`) |
| `...-noopt` | select non optimized flags (overwrites `WPM_CONFIG`) |
| `...-pro` | select profiler flags (overwrites `WPM_CONFIG`) |
| `...-32` | try 32 bit (on some systems the user can choose between 32 and 64 bit binaries) |
| `...-64` | try 64 bit |
| `...--vendor` | query vendor |
| `...--cfg_file` | source of flags |

### 2.3.2 Compiler

| | |
|---|---|
| `...--objext` | extension of object files generated by compiler ("o" or "obj") |
| `...--cxx` | C++ compiler |
| `...--cc` | C compiler |
| `...--f77` | FORTRAN compiler |
| | |
| `...--cxxflags` | accumulated compiler flags: |
| `...--cflags` | basic + warn + incdef + opt/dbg/noopt/pro |
| `...--fflags` | |
| `...--cxxflags_min` | accumulated compiler flags: |
| `...--cflags_min` | basic + incdef + opt/dbg/noopt/pro |
| `...--fflags_min` | |
| | |
| `...--cxxflags_warn` | warning flags |
| | |
| `...--cxxflags_incdef` | include and defines |
| | |
| `...--cxxflags_dbg` | debug without optimization |
| `...--cxxflags_noopt` | no optimization |
| | |
| `...--cxxflags_opt` | optimization (level O3) |
| `...--cxxflags_pro` | optimization (level O3) with profiler flags |
| `...--cxxflags_basic` | basic flags assumed by the flags above |

| | |
|---|---|
| `...--cflags_warn` | analog to the C++ compiler flags |
| `...--cflags_incdef` | |
| | |
| `...--cflags_dbg` | |
| `...--cflags_noopt` | |
| `...--cflags_opt` | |
| `...--cflags_pro` | |
| `...--cflags_basic` | |
| | |
| `...--fflags_warn` | analog to the C++ compiler flags |
| `...--fflags_incdef` | |
| | |
| `...--fflags_dbg` | |
| `...--fflags_noopt` | |
| `...--fflags_opt` | |
| `...--fflags_pro` | |
| `...--fflags_basic` | |
| | |
| `...--fmangle` | FORTRAN name mangling (compare Section 2.1) |
| `...--openmp_flags` | OpenMP flags |

### 2.3.3 Linker

| | |
|---|---|
| `...--exeext` | extension of executable files generated by linker (empty or ".exe" for windows) |
| `...--ldflags` | linker flags |
| `...--libs_start` | some linker supports grouping of statical libs with a unique list. Close group with `--libs_end`. |
| `...--libs_end` | |
| `...--libs` | basic vendor libs (for example: -lg2c) |
| `...--cxxlibs` | |
| `...--clibs` | |
| `...--flibs` | |
| `...--flibs_first` | on some systems this should be a problem |

### 2.3.4 Library

| | |
|---|---|
| `...--ranlib` | ranlib generates an index to the contents of an archive and stores it in the archive |
| `...--ar` | archive program |
| `...--arflags` | flags for creating an archive |

### 2.3.5 System

`...--bits`

## 2.4 Flags *Makefile* example

The following example *Makefile* uses the WPM subcommand `wpm-build flags`.

```
WPM_CFG = -dbg
WPM_FLAGS = wpm-build $(WPM_CFG) flags

CXX = `$(WPM_FLAGS) --cxx`
CXXFLAGS = `$(WPM_FLAGS) --cxxflags`
AR  = `$(WPM_FLAGS) --ar --arflags`
RANLIB  = `$(WPM_FLAGS) --ranlib`
LDFLAGS = `$(WPM_FLAGS) --ldflags`
LIBS = `$(WPM_FLAGS) --libs`

COMPILE = $(CXX) $(CXXFLAGS) -c
LINK = $(CXX) $(CXXFLAGS) $(LDFLAGS) -o

LIB_OBJ = file1.o file2.o
BIN_OBJ = main.o

all: prog.exe

clean:
    -rm -f *.o prog.exe

prog.exe: libfile.a $(BIN_OBJ)
    $(LINK) $@ $(BIN_OBJ) libfile.a $(LIBS)
```

```
libfile.a: $(LIB_OBJ)
    $(AR) $@ $(LIB_OBJ)
    $(RANLIB) $@

.cxx.o:
    $(COMPILE) $<

.PHONY: all clean
.SUFFIXES: .cxx .o
```

This example is part of the WPM distribution. See directory

`wpm/examples/example_flags_makefile`.

## 2.5 Build

The described above usage of `wpm-build flags` is the most flexible way to build programs with a makefile. But alternatively a more comfortable usage is provided by `wpm-build`. In many cases the build subcommands are sufficient:

• `wpm-build compile [extra-flags] *.cxx *.c *f` This subcommand is equal to

```
--cxx [extra_flags] --cxxflags $DEFS $INCLUDES $CXXFLAGS -c *.cxx
--cc [extra_flags] --cflags $DEFS $INCLUDES $CFLAGS -c *.c
--f77 [extra_flags] --fflags $DEFS $INCLUDES $FFLAGS -c *.f
```

The user can specify additional flags with the environment variables `DEFS`, `INCLUDES`, `CXXFLAGS` and with command line extra-flags.
• `wpm-build lib <lib> *.o` Same as `--ar --arflags <lib> *.o`.
• `wpm-build ranlib <lib>` Same as `--ranlib <lib>`.
• `wpm-build cplib <lib> <from-dir> <to-dir>` Installs `<lib>` into a possibly existing target directory. This subcommand is equal to

```
mkdir_p <to-dir>
cp <from-dir>/<lib> <to-dir>
--ranlib <from-dir>/<lib>
```

• `wpm-build link <main> [extra-flags] *.o *.a *.so` The subcommand supports grouping with `--libs_start, --libs_end` described in Section 2.3. The equal command

```
--cxx --cxxflags --ldflags -o <main> [extra-flags] *.o *.a *.so --libs
```

## 2.6 Build *Makefile* example

The following example *Makefile* uses the WPM subcommands `wpm-build compile`, `wpm-build link`, `wpm-build lib` and `wpm-build ranlib`.

```
WPM_CFG = -dbg
WPM_BUILD = wpm-build $(WPM_CFG)

LIB_OBJ = file1.o file2.o
BIN_OBJ = main.o

all: prog.exe

clean:
	-rm -f *.o libfile.a prog.exe

prog.exe: libfile.a $(BIN_OBJ)
	@echo "Linking $@..."
	@rm -f $@
	@$(WPM_BUILD) link $@ $(BIN_OBJ) libfile.a $(LIBS)

libfile.a: $(LIB_OBJ)
	@echo "Creating $@..."
	@$(WPM_BUILD) lib $@ $(LIB_OBJ)
	@$(WPM_BUILD) ranlib $@

.cxx.o:
	@echo "Compiling $< ..."
	@$(WPM_BUILD) compile $<

.PHONY: all clean
.SUFFIXES: .cxx .o
```

This example is part of the WPM distribution, see directory

`wpm/examples/example_build_makefile`,

and is equivalent to the build task as described in the last example of Section 2.4.

## 2.7 Check vendor

The user should always perform a test run with the selected tools of a vendor (s. 2.2). Especially, when the target system is new. The administrator might decide to choose non-standard installation options for the compiler. If this is the case, the user must adapt the desired vendor file.

• `wpm-build check` This subprogram builds simple test-code (.cxx,.c,.f) and tries to link libs with interdependency (test: `--libs_start, --libs_end`); link mixing C++, C and FORTRAN code which tests the macros `F77_FUNC(..), F77_FUNC_(..)` generated from subcommand `wpm-build header`.

## 2.8 Writing user extensions

The user can define an own set of build tools and flags. It is recommend either to start from an existing vendor file (`sys/*/*.sh_inc`) that is close to the target setting or from the commented template file `sys/template.sh_inc`. The template file contains all available variables, which names are the same as the flags but extended with the prefix `arch_`. The arch-files will be included by `wpm` and consequently this files should also contain Bourne Shell compatible code. Testing of the changes is possible with `wpm-build check` (s. 2.7).

# 3 WPM and users

This section describes the user interface of WPM, that is, the set of subprograms available for performing maintenance of packages.

• `wpm initdb` to initially create a database
• `wpm install` to install packages
• `wpm uninstall` to uninstall packages
• `wpm query` to query database
• `wpm-build` to access build commands
• `wpm-bundle` to create Mac OS X application bundles[4]

Both command-line tools `wpm` and `wpm-build` are controllable with switches and variables. To customize the tools with higher flexibility, all variables can be defined on the command line, from the environment or in a resource file (.wpmrc in working directory). The evaluation occurs in following order:

[4]For details see Appendix A.

1. command line
2. environment
3. resource file

There are some general variables which influence the behavior of the subprograms discussed in this section.

`WPM_DBPATH=<dir>` location of the WPM database (default `$HOME/wpm-db`)
`WPM_INSTDIR=<dir>` location of installed packages (default `$HOME/local`)

To become aware of the variable settings in a safe way it is recommended to attempt the following subprogram

```
$ wpm status
```

## 3.1 Initialize database

Before WPM is able to install a package the user must setup a database. This consists of nothing more than a set of ascii files.

• `wpm initdb` If the directory `WPM_DBPATH` is not empty the procedure aborts. Otherwise the subcomand creates the directory and puts a marker inside.

• `wpm initdb --nonfs` Without this flag WPM creates an additional marker which tells `wpm install` to arrange the packages in the `WPM_INSTDIR` and the database entries in a hierarchical level defined by `WPM_HOST`. This feature may be useful on network file systems.

• `wpm initdb --nomulti-pkg` Without this flag WPM creates an additional marker which tells `wpm install` to arrange the packages in the `WPM_INSTDIR[/WPM_HOST]` in a hierarchical level defined by the package. For example if WPM should install different versions of a package this flag is necessary.

• `wpm initdb WPM_INSTDIR=<new-default-dir>` Change the default installation directory to `<new-default-dir>`.

• `wpm initdb --nonfs --nomulti-pkg WPM_INSTDIR=<new-default-dir>` Combination of the above flags is allowed.

There is no way to remove a database with WPM. When the user wants to change options after *wpm initdb*, the recommended way is the complete removal of the directory `WPM_DBPATH` and the restart of `wpm initdb`. In case of a non-empty database the uninstall all packages is necessary.

**Remark:** It is possible to set `WPM_INSTDIR=/usr` and install all packages into the unix directory structure. During installation the subcommand `wpm install` performs a conflict check before files are copied to the target directories.

## 3.2 Getting information about package files

To discover the content and dependencies of a package file, the following command can be used.

```
$ wpm install --simulate <package>.wpm
```

Furthermore, the command extracts the package file into the current working directory and prints the status of directory settings (package file details s. 4.2).

For example

```
$ wpm install --simulate check-1.0-1.wpm
WPM_DBPATH      : /Users/strecken/wpm-db
WPM_HOST        : powerpc-apple-darwin8.11.0
WPM_CONFIG      : opt
WPM_COM         : wpm-build flags (default)
WPM_SOURCE_DIR  : <DIR>/wpm/examples/WPMS/BUILD_check
WPM_BUILD_DIR   : <DIR>/wpm/examples/WPMS/BUILD_check/check
WPM_REQUIRES    : checklib-1.0
WPM_PROVIDES    : check-1.0
WPM_INSTDIR     : <DIR>/local
WPM_INSTDIR_bin : <DIR>/local/bin
```

The variable `WPM_REQUIRES` indicates a dependency from `checklib-1.0`. To be sure that `check-1.0-1.wpm` is the right package, execute the command

```
 wpm install --simulate
```

again and compare the variable `WPM_PROVIDES` with the required package.

## 3.3 Getting information about installed packages

All information about installed packages is stored in the WPM database. This information is important for the package manager itself (especially for the subprogram `wpm uninstall`) and for packages that shall be linked against an installed package. To query the database, WPM provides the subcommand

```
$ wpm query
```

Without additional switches, a list of installed packages will be printed out. To find a package with desired properties, the following options maybe useful

• `wpm query find=opt` returns packages with available option `opt` (empty or not)
• `wpm query find=opt substr=sub` returns packages with `opt` containing `substr`
• `wpm query find=opt equal=eq` returns packages with `opt` equal to `eq`

Beyond these general options, several options concerning a specified package are available.

• `wpm query pkg=pkg-label --check-missing` checks whether the installation is complete. If some files are missing, the list will be stored in `./wpm_check_missing.log`.
• `wpm query pkg=pkg-label files=subdir exe=exe` gets the filename of installed executable. Note: The result is system depending (compare Section 2.3.3).

Example

```
$ wpm query pkg=chksum-1.0-1 files=bin exe=whirlpool
```

The following sections deal with commands that are suboptions from installed database script-files. A more detailed description is given in Sections 4.3.2 and 4.3.4.

### 3.3.1 General

• `wpm query pkg=package-label...`

| | |
|---|---|
| `...-` | filename of database script-file. |
| `...--help` | list of available flags. |
| `...--summary` | summary of the package. |
| `...--name` | pure name of the package (see Section 1.4). |
| `...--version` | version number of the package (see Section 1.4). |
| `...--release` | release number of the package (see Section 1.4). |
| `...--requires` | dependencies. |
| `...--provides` | dependencies. |

### 3.3.2 Build

• `wpm build pkg=package-label...`

| | |
|---|---|
| `...--flags` | flags of specified package for compiling. |
| `...--libs` | libs of specified package for linking. |
| `...--FLAGS` | including dependencies. |
| `...--LIBS` | including dependencies. |

### 3.3.3 History

The following flags provides history information about the installation of a package.

- `wpm query pkg=package-label...`

| | |
|---|---|
| `...--requires_resolved` | used requirements. |
| `...--packagefile` | used package source-file. |
| `...--getfrom` | url of tarball. |
| `...--com` | used built-interface. |
| `...--com_bits` | used 32 or 64 bits. |
| `...--com_config` | used config (`opt|noopt|dbg|pro`). |
| `...--vendor` | used vendor. |
| `...--instdir` | target directory. |

## 3.4 Install packages

The simplest and most suitable way to install a WPM package is

```
$ wpm install <package>.wpm
```

But first ensure with `wpm-build check` that the selected vendor `WPM_VENDOR` works well (compare 2.7). If the software is not included in the package, `wpm install` will invoke `wpm get` with specified options to download and unpack the missing tarball[5]. This subprogram creates a hidden file (`.wpmrc`) during the first run in the current working directory. This file contains a list of locations (URL, local path name) stored in `WPM_GET_DIRS` used by `wpm get`. For a concrete package the user may add locations with `WPM_GET_DIRS_<package-label>` which `wpm get` prefers. The tool *curl* is the default remote get command. Replace this is possible via variable `WPM_GET` and `wpm get` calls this in the following manner

```
 $WPM_GET <dst-file> <src-URL>
```

The next three steps will performed by `wpm get`

[5]Many applications are downloaded in compressed *tar* format, often called a tarball.

1. Download software not included in the package file, but necessary for the build using *curl*.
2. Check signature of tarball (compare Section 4.5).
3. Unpack Software (supported formats: .tgz, .tar.gz and .shar.gz)[6].

If these steps have been successfuly performed, WPM continues with the proper installation procedure. Basically, the installation subprogram executes the following steps

1. conflict check to ensure that no pre-existing files in the target installation directory `WPM_INSTDIR` structure will be overwritten
2. patch (`wspec` section[7])
3. prep (`wspec` section)
4. configure (`wspec` section)
5. clean (`wspec` section)
6. build (`wspec` section)
7. install (`wspec` section)
8. install-check
9. create config-script in `WPM_DBPATH`

The maintainer can use some additional options for package installation.

- `wpm install [Options] <package>.wpm`

The advanced options are basically for package developers and should be avoided by the WPM user.

[6]Uncompress with *gunzip -c*, open archive with *tar -xf* and a shar archive with *sh -xf*.
[7]See 4.3 for details.

### 3.4.1 Basic options

| | |
|---|---|
| `--simulate` | shows variable-state only |
| `--multi-pkg` | extend `WPM_INSTDIR` with package-label (`=<name-vers-release>`) |
| `WPM_COM=<script-file>` | Compiler-script. Default: 'wpm-build flags'. If this has been set, `WPM_VENDOR`, `WPM_BITS` and `WPM_CONFIG` have no influence to `WPM_COM`. |
| `WPM_VENDOR=<vendor>` | selects a vendor; default empty (compare Section 2.2) |
| `WPM_BITS=32\|64` | selects bits; default empty (compare options of `wpm-build flags` in Section 2.3) |
| `WPM_CONFIG=opt\|noopt\|dbg\|pro` | selects optimization level; default opt (compare with options of `wpm-build flags` in Section 2.3) |
| If the setting is | `=noopt`, the package `name` will be extended with `WPM_CONFIG`. |
| | `=dbg`, the section `files_source` (s. a. Section 4.3.4) will be installed automatically. |

### 3.4.2 Advanced options - control execution of `wspec` sections

| | |
|---|---|
| `-bp` | execute `prep configure` |
| `-bc` | execute `prep configure clean build` |
| `-bi` | execute `prep configure clean build install` (default) |

| | |
|---|---|
| `--nopatch` | skip section `patch` |
| `--noprep` | skip section `prep` |
| `--noconfigure` | skip section `configure` |
| `--nobuild` | skip section `build` |
| `--noinstall` | skip section `install` |
| `--noclean` | skip section `clean` |

### 3.4.3 Advanced options - install, uninstall, repair

| | |
|---|---|
| `--uninstall` | remove package installation (without dependency check!) |
| `--check-conflict` | only check whether package-files are existing in install-dir |
| `--repair-script` | overwrites the config-script in `WPM_DBPATH` |
| `--nodeps` | skip check dependencies during install/uninstall |
| `--noinstallcheck` | skip check after `install` |

Warning: The `--uninstall` procedure may set WPM in an unstable state.

### 3.4.4 Advanced options - control installation directories granularly

| | |
|---|---|
| `WPM_INSTDIR_<subdir>=<dir>` | overwrite subdir defaults<br>`WPM_INSTDIR_bin=WPM_INSTDIR/bin`<br>`WPM_INSTDIR_lib=WPM_INSTDIR/lib`<br>... |
| `--force-<subdir>` | force install for subdir (disables conflict check!) for example: `subdir=share` installs files from section `files_share` into `<prefix>/share` |

## 3.5 Erase packages

Remove an installed package is an important part of WPM. After installation a list of all installed files is stored in the database. The package manager removes each of these files explicitly, without calling a possibly existing uninstall procedure from a software. The WPM package itself is not necessary for this task. This way ensures that no commands like `rm -f *.a` are called and the well known unix directory structure for installation can established.

```
$ wpm uninstall pkg=<package>
```

This procedure can not uninstall a package which is required by other installed packages. If the maintainer needs to re-install a package the advanced (and unsafe) install option `--uninstall` is recommended (s. a. Section 3.4).

## 3.6 Examples

### 3.6.1 Install each package as `opt` and `dbg`

Optionally, a database folder might be defined at the first step. For Example with

```
$ export WPM_DBPATH=$HOME/wpm-db_example
```

The next command will initially create a WPM database which only separates the installation directories of each package and sets a new default for the installation base folder; more details in Section 3.1.

```
$ wpm initdb --nonfs WPM_INSTDIR=$HOME/local_example
```

Now we prepared the database for installation of WPM packages. Before we will install the first package, a check of the desired compiler and build tools is recommended (s. a. 2.7).

```
$ wpm-build check
```

The result should produce several output lines after `Run ...` without an error. If this is not the case, the user could first try to choose another available vendor. A detailed description how to do this can be found in Section 2.2.

Perform the installation of all package availabe in directory `examples/WPM` with the following commands as optimized binaries.

```
$ wpm install checklib-1.0-1.wpm
$ wpm install check-1.0-1.wpm
$ wpm install chksum-1.0-1.wpm
```

After these steps the database should contain the following entries:

```
$ wpm query
check-1.0-1
checklib-1.0-1
chksum-1.0-1
```

Now we repeat the same installation commands like above, but we only switch to `dbg` which generates non-optimized binaries with debug information and installs source files of the packages, too.

```
$ wpm install WPM_CONFIG=dbg checklib-1.0-1.wpm
$ wpm install WPM_CONFIG=dbg check-1.0-1.wpm
$ wpm install WPM_CONFIG=dbg chksum-1.0-1.wpm
```

This task will stop after second command, because the package manager detects a conflict. The output appears as follows:

```
 checking dependency: checklib-1.0 ...
 selection failed with WPM_sel_checklib_1_0=<>
    (select one from: checklib-1.0-1 checklib_dbg-1.0-1)
 missing package: Require <checklib-1.0>
```

We installed two packages (`checklib-1.0-1` and `checklib_dbg-1.0-1`) which provide the same software. At this point WPM needs assistance from the user. We make the following decision:

```
$ export WPM_sel_checklib_1_0=checklib_dbg-1.0-1
```

and retry the remaining commands again:

```
$ wpm install WPM_CONFIG=dbg check-1.0-1.wpm
$ wpm install WPM_CONFIG=dbg chksum-1.0-1.wpm
```

The database now contains the following entries:

```
$ wpm query
check-1.0-1
check_dbg-1.0-1
checklib-1.0-1
checklib_dbg-1.0-1
chksum-1.0-1
chksum_dbg-1.0-1
```

And finally you can find all packages which installed source code for debugging with the next command (s. Section 3.3).

```
$ wpm query find=files substr=source
```

### 3.6.2 Install packages built with different compiler vendors

It is highly recommended to create an own WPM database for each compiler vendor. The following citation explains the reason for this decision:

*In general, Fortran compilers are not binary compatible, due to using different runtime libraries. Intel Fortran Compiler is binary compatibl with C-language object files created with either Intel C++ Compiler for Linux or the GNU gcc compiler. The Intel Fortran Compiler documentation has further details on calling C language functions from Fortran.*

*Intel Fortran Compiler for Linux uses a different name-mangling scheme than the GNU Fortran compiler. Intel does not recommend mixing object iles created by the Intel Fortran Compiler and the GNU Fortran compiler.* [White Paper: Intel(R) Compilers for Linux*: Compatibility with GNU Compilers]

For this example we assume that a system *ia32 linux* (s. 2.2) is available. We begin with *gnu* and define a new database location with:

```
$ export WPM_DBPATH=$HOME/wpm-db_example_gnu
```

The next command will initially create a WPM database and sets a new default for the installation base folder[8]; more details in Section 3.1.

```
$ wpm initdb WPM_INSTDIR=$HOME/local_example_gnu
```

Just like in the last example in Section 3.6.1 a check of the desired compiler and build tools is recommended (s. a. Section 2.7):

```
$ wpm-build vendor=gnu check
```

Perform the installation of all package availabe in directory `examples/WPM` with the following commands as optimized binaries.

```
$ export WPM_VENDOR=gnu
$ wpm install checklib-1.0-1.wpm
$ wpm install check-1.0-1.wpm
$ wpm install chksum-1.0-1.wpm
```

Now we repeat the above steps with `_intel9` postfix and select the vendor *intel9*.

```
$ export WPM_DBPATH=$HOME/wpm-db_example_intel9
$ wpm initdb WPM_INSTDIR=$HOME/local_example_intel9
$ wpm-build vendor=intel9 check
$ export WPM_VENDOR=intel9
$ wpm install checklib-1.0-1.wpm
$ wpm install check-1.0-1.wpm
$ wpm install chksum-1.0-1.wpm
```

# 4 WPM and developers

This chapter is a guide to set up packages around existing software. Generally the packages can be divided into the following categories:

[8]A diffrent `WPM_INSTDIR` is not mandatory.

- source packages (build and install)
- binary packages (install)
- virtual packages (refer)

The following sections will basically deal with source and virtual packages. A package which builds source code of the included software during installation time is called source package.

A virtual package only exist logically, not physically; that is why they are called virtual. Get around the problem of pre-existing software by building a virtual package that lists the system libraries installed without WPM in an WPM package.

## 4.1 Guideline

The main tasks in creating WPM packages are:

1. Get the software (download tarball)
2. Patch them and create a reproducible build of the software
3. Outline any dependencies
4. Write the `wspec` file containing the above steps (s. 4.3)
5. Set up the WPM package
6. Test the package

With the first three steps the packager mainly discovers the software. Before trying to make a package, the packager needs to figure out how to build the application or library planned to be packaged.

## 4.2 Build a `wpm` package

First it is recommend to set up a directory structure as follows

| | |
|---|---|
| `SOURCES` | contains all packages in subfolders `<name>-<version>-<release>` |
| `WPMS` | contains the packages generated by `wpm setup` |

The file `<name>.wspec` should reside in the directory

`SOURCES/<name>-<version>-<release>`.

The following command builds a WPM package file with file extension `wpm` in the current working directory[9].

- `wpm setup [options] <name.wspec>`

**Available options**

`files=<file1:subdir1:...>` Specifiy a list of files/directories inside

```
SOURCES/<name>-<version>-<release>
```

which should be added to the package. Only relative path names are allowed. The settings of the header variables `Source` and `SourceFiles` in the `<name>.wspec` are defaults.

`--nosource` Ignore the `Source` and `SourceSubDir` header variables defined in `<name>.wspec`.

It is also possible to put all the command-line options for `wpm setup` into the file `setup.wini`. The next command loads `<package-dir>/setup.wini` and builds a WPM package

- `wpm setup <package-dir>`

The options inside a `setup.wini` file can be specified as shown in the following example

```
# setup.wini
nosource="yes"
files="README"
wspec="chksum.wspec" # this must be specified
```

This file becomes not part of the WPM package.

## 4.3 Write a `wspec` file

The `wspec` file, short for WPM specification file, defines all the actions the `wpm` command should take to build the package, as well as all the actions necessary for the `wpm` command to install the package. Like WPM itself the `wspec` file should contain Bourne Shell syntax compatible code. A header variable is implemented as an Bourne Shell variable and a section is implemented as an Bourne Shell function.

[9]See Section 1.4 for an overview of the WPM package file format.

The naming convention is to name the file with the package name and a `.wspec` filename extension.

```
SOURCE/<name>-<version>-<release>/<name>.wspec
```

The execution of the following sections will be done by `wpm install`. Before this task starts, `wpm install` includes the `wspec` file and defines several WPM variables, which are useable in the sections.

### 4.3.1 Pre-defined WPM environment

Following variables are preseted in `wspec` files:

| | |
|---|---|
| `WPM_SOURCE_DIR` | abs-dir with the wspec-file of a package |
| `WPM_BUILD_DIR` | abs-dir of build |
| `WPM_INSTDIR` | abs-dir for installation |
| `WPM_INSTDIR_<instsub>` | abs-dir for installation |
| `WPM_GET` | |
| `WPM_GET_DIRS` | |
| `WPM_CAT1EXT` | |
| `WPM_CAT3EXT` | |
| `WPM_EXEEXT` | |
| `WPM_OBJEXT` | |
| `WPM_QUERY_<dependency>` | query other installed packages<br>for example:<br>`requires="lua-5.0 dlopen"`<br>`$WPM_QUERY_lua_5_0 --libs`<br>`$WPM_QUERY_dlopen --libs` |
| `WPM_HOST` | current arch (detected with *config.guess*) |
| `WPM_COM` | tools and flags for build.<br>default: `wpm-build flags` |

All directories saved in WPM-variables are absolute and existing directories. Working-dir of all sections are `WPM_BUILD_DIR`.

The following pre-defined functions maybe usefull in the sections:

| | |
|---|---|
| `mkdir_p <dir>` | Create intermediate directories as required. No error will be reported if a directory given as an operand already exists. |
| `wpm_unpack_in_wd <Source>` | Uncompress with *gunzip -c*, open archive with *tar -xf* and a shar archive with *sh -xf* in working directory. |

### 4.3.2 Header variable details

**Summary** Short software description

**Name** Pure name of a package (compare Section 1.4)

**Version** Version number of a package (compare Section 1.4)

**Release** Release number of a package (compare Section 1.4)

**Copyright** Info

**Checksum** Validate tarball (see 4.5)

**Source** tarball name without path or URL. A list of names separated by whitespace is also allowed[10].

**GetDirs** Web locations (URLs) without filename; the software is not included in the package.

**SourceSubDir** Relative build directory; WPM unpacks the tarball into this folder. Default: `<Name>-<Version>-<Release>`

**SourceFiles** Local sources; the software is included in the package.

**requires** Dependency (see Section 4.4)

**provides** Dependency (see Section 4.4)

**URL** Software homepage

**Vendor** Software vendor

**Packager** People who build the package.

**PatchN** Vendor patch tarball name without path or URL (**N**=0,1,...); see also associated section `PatchApplyN` in Section 4.3.3.

**PatchChecksumN** Validate tarball `PatchN`

**PatchSubDirN** WPM unpacks the tarball `PatchN` into this relative folder.

### 4.3.3 Section details (build and install)

The working directory of all sections is `WPM_BUILD_DIR`. This folder will be created in the current working directory and is defined as

[10]This could be useful if the name varies for different locations.

`<current-working-dir>/BUILD_<SourceSubDir>`

The header variable `SourceSubDir` is explained in Section 4.3.2. The execution of the following sections are controlable through the options of `wpm install`; see details in 3.4.2.

**locals** Setting local variables which may helpful inside the `wspec` file. It is recommend to use a safe prefex, for example `local_` for these variables.

**patch** Start with pristine sources; then patch as needed. A patch is an automated set of modifications to the source code. Keep the original sources separate from any patches you need to make the software work in your environment. See also header variable `PatchN` in Section 4.3.2 for details concerning a vendor patch.

**PatchApplyN** Defines the actions how to apply the vendor patch `PatchN`. This section will be called by the WPM function `wpm_patch` which should be specified in the `patch` section.

**prep** The prep section, short for prepare, defines the commands necessary to prepare for the build.

**configure** Perform the configuration of the software. Usually at this place the packager should pass the build tools and flags from `wpm-build` to the software (details in Section 2.3).

**clean** The clean section cleans up the files that the commands in the build sections create. This section should only cleans up binaries generated during build-time in `WPM_BUILD_DIR`.

**build** The build target should perform all the compilation of the package. Usually, this will include just a few commands, since most of the real instructions appear in the *Makefile*.

**install** This section holds the commands necessary to install the newly built software. In most cases your install section should run the make install command. But sometimes the packager needs to install the built software itself. The following list should be a suggestion to program this step in a portable way:

- Install an executable script

  ```
  cp mysoft-config $WPM_INSTDIR_bin
  chmod 755 $WPM_INSTDIR_bin/mysoft-config
  ```

- Install a built binary

  ```
  cp mysoft$WPM_EXEEXT $WPM_INSTDIR_bin
  chmod 755 $WPM_INSTDIR_bin/mysoft$WPM_EXEEXT
  ```

- Install documentation

  ```
  mkdir_p $WPM_INSTDIR_share/doc/mysoft
  (cd $WPM_BUILD_DIR/documentation
   cp *.gif *.html $WPM_INSTDIR_share/doc/mysoft
   chmod 644 $WPM_INSTDIR_share/doc/mysoft/*.*
  )
  ```

- Install a library

  ```
  cp $WPM_BUILD_DIR/libmysoft.a $WPM_INSTDIR_lib
  `$WPM_COM --ranlib` $WPM_INSTDIR_lib/libmysoft.a
  ```

**Remark** Before a section executes code which requires the pre-defined WPM variable `WPM_COM`, the packager should add a check-flag to force WPM to perform a simple check. Example:

```
configure_require_wpm_com="yes"
configure()
{
 CXX=`$WPM_COM --cxx` CXXFLAGS=" `$WPM_COM --cxxflags`"  ./configure
}
```

### 4.3.4 Section details (database)

The previous `wspec` sections perform real actions on file system with the aim of building and installing the software. Finally the following sections result to entries in the WPM database.

The sections concerning files are also important for the subprogram `wpm uninstall`. If it is planned to use the install procedure of the software (e.g. `make install`), the packager should run the install section and determine the list of files which appears in the `WPM_INSTDIR`.

**files**[11] This section should define a list of install-subdirs of `WPM_INSTDIR` (for example lib, bin, include, ...). Remark: If a virtual package is planned, this section will usually be empty.

**files_<install-subdir>** This section must return a list of <install-subdir>-files [12]. It is also allowed to list files in a relative path structure. The directories itself must not be specified and will be created by the package manager.

[11]This section is an exception concerning the syntax. The section `file` must be written as a Bourne Shell variable.

[12]without wildcards

**files_source** As the last section this one holds a list of files in the same manner. But this list should only contain source files and headers which are relevant for degugging this software. This will be installed automatically if the user specifies the `WPM_CONFIG=dbg`; see also 3.4.1.

**package_flags** This section defines the compiler flags concerning the package. Usually this section contains only defines (`-D` switches) and include search paths (`-I` switches) of the installed package. If the software contains only applications (and no libraries), this section must not be specified. This also holds true for the next sections.

**package_libs** This section defines the linker flags concerning the package. Usually this section only contains libraries with absolute paths of the installed package.

**package_FLAGS** This section contains flags for other software to be built against the package.

**package_LIBS** This section contains libraries to be built against the package. This may be important in case of usage of system dependencies detected by the software.

## 4.4 Dependency information

Version clauses should be used rigorously in build-time relationships so that one cannot produce bad or inconsistently configured packages when the relationships are properly satisfied.

It is not necessary to list packages which are required merely because some other package in the list of build-time dependencies depends on them. The reason for this is that dependencies change. What others need is their business.

The header variable `requires` in a `wspec` file specifies a list of packages separated by space (details about `wspec` files in 4.3). Optionally packages are enclosed by square brackets and alternatives are separated by vertical bars. For example, a list of dependencies might appear as:

```
requires="[dlopen] gl math pthread termcap \
 lapackblas-sys|lapackblas-3.0 \
 lua-5.0 \
 gspar-1.0 \
 arpack-96 \
 gms-1.0 \
 metis-4.0 \
 pardiso-1.0 \
 triangle-1.6 \
 tetgen-1.4.1 \
 fltk-1.1.x-r5329 \
 swig-1.3.27"
```

If WPM finds an installed package which provides an item, a dependency will match. Example of several packages

```
provides="lapackblas-3.0"
provides="lua-5.0"
provides="swig"
```

During installation of a package WPM tries to resolve the required package(s) in the following order

1. `Name-Version-Release`
2. `Name-Version`
3. `provides`

These header variables are also part of the `wspec` file; see 4.3.

## 4.5 Signature

The Whirlpool algorithm was developed by Paulo S.L.M. Barreto and Vincent Rijmen [The WHIRLPOOL Hash Function[13]]. With this algorithm the packager generates an unique (hopefully) fingerprint of the tarball. If the package manager downlowds a tarball during an installtion task, WPM will generate a fingerprint from this file and will compare this signatur with the expected one. Not all software providers take care about the versions. However, with this mechanism the user is on the safe side.

The following command gereates an signatur from a file.

```
$ wpm chksum examples/web_location/chksum/chksum.tar.gz
Whirlpool digest: 2ca60aed7498d3c....
```

In order to perform the validation in the installation task, the packager must specify the following `wspec` header variable with the result of `wpm chksum`.

```
Checksum="Whirlpool digest: 2ca60aed7498d3c...."
```

[13]See URL: http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html or ISO/IEC 10118-3:2004

## 4.6 Examples

The examples of this Section are also included in the `example` folder of WPM.

### 4.6.1 Source package of a library

The packager can use the steps described in Section 4.1 as a practical guideline. In this Section we explain the steps on the basis of the package *levmar*[14].

**Write the `wspec` file**

First we download the tarball `levmar-2.1.3.tgz` and generate a signature of this file as described in Section 4.5.

```
$ wpm chksum levmar-2.1.3.tgz
Whirlpool digest: f1d388f2ce7aeb27....
```

Now we copy the `template_web_source.wspec` to `levmar-2.1.3-1/levmar.wspec` and modify the header of them as follows

```
Summary="levmar - Levenberg-Marquardt nonlinear least squares algorithms"
Name="levmar"
Version="2.1.3"
Release="1"
Copyright="GNU General Public License"
Source="levmar-2.1.3.tgz"
Checksum="Whirlpool digest: f1d388f2ce7aeb27...."
GetDirs="http://www.ics.forth.gr/~lourakis/levmar"
SourceFiles=""
SourceSubDir="levmar-2.1.3"
requires=""
provides="levmar-2.1.3"
URL="http://www.ics.forth.gr/~lourakis/levmar"
Distribution=""
Vendor="M.I.A. Lourakis"
Packager="Timo Streckenbach <strecken@wias-berlin.de>"
```

The `wspec` section of the copy should be removed. We only want to package the library of *levmar* and not a executable program. That is why the header variable `requires` leaved empty (details in Section 4.4).

[14]Levenberg-Marquardt nonlinear least squares algorithms in C/C++, URL: http://www.ics.forth.gr/ lourakis/levmar

In this example the library should only contain the compilation of the following source files

```
 lm.c Axb.c misc.c lmlec.c lmbc.c
```

And we plan to use an external *lapack* library as dependency. This requires the definition of the preprocessor variable `HAVE_LAPACK` during compile time of the source files.

In the next we make a test compilation of the desired source files. Ensure that your compiler vendor will work with `wpm-build check` (details in Section 2.7). Now the first try of

```
$ wpm-build flags --cc -opt --cflags
gcc -ansi -pedantic -Wall -O3
$ gcc -ansi -pedantic -Wall -O3 -DHAVE_LAPACK -c \
 lm.c Axb.c misc.c lmlec.c lmbc.c
```

will produce an compilation error. This indicates that the C source code is not written in clean ANSI standard. Here we decide to reduce this restriction with

```
$ wpm-build flags --cc -opt --cflags_min
gcc -O3
$ gcc -O3 -DHAVE_LAPACK -c lm.c Axb.c misc.c lmlec.c lmbc.c
```

This was the critical part. If the build of a software is more complicated, the packager often can not avoid to use the software build system. For this example the sections `locals`, `prep`, `configure` and `patch` are not required. The packager must fill the body of those with a `:` in an own line.

Now we can write the `build` and `clean` sections as follows

```
build_require_wpm_com="yes"
build()
{
 `$WPM_COM --cc -opt --cflags_min` -DHAVE_LAPACK -c \
                                       lm.c Axb.c misc.c lmlec.c lmbc.c
 `$WPM_COM --ar --arflags` liblevmar.a lm.o Axb.o misc.o lmlec.o lmbc.o
 `$WPM_COM --ranlib` liblevmar.a
}
clean()
{
 rm -f lm.o Axb.o misc.o lmlec.o lmbc.o
}
```

The compilation line beginning with `WPM_COM` is equivalent to the manual command-line build. And the archive will be created in the standard way as described in the example in Section 2.4. From the point of the build system this `wspec` section is portable. But maybe the source code itself not. To ensure this, the packager must repeat the last build steps on all desired systems and vendors.

The following section `install` installs the built library `liblevmar.a` and two header files

```
install()
{
 cp $WPM_BUILD_DIR/liblevmar.a $WPM_INSTDIR_lib
 `$WPM_COM --ranlib` $WPM_INSTDIR_lib/liblevmar.a

 mkdir_p $WPM_INSTDIR_include/levmar-2.1.3
 cp $WPM_BUILD_DIR/lm.h $WPM_INSTDIR_include/levmar-2.1.3/lm.h
 cp $WPM_BUILD_DIR/misc.h $WPM_INSTDIR_include/levmar-2.1.3/misc.h
}
```

Only a copy of the library is not enough. For example on Apple Mac OS X the indexing with *ranlib* must happen after the copy command. The WPM shell function `mkdir_p` forces the existence of the destination subdirectory `levmar-2.1.3` for the header files.

The following part of the `wspec` file tells WPM which files will be installed. WPM defines the variables `WPM_INSTDIR_lib` and `WPM_INSTDIR_include` and creates these folders depending from section `files`. Furthermore depending from `files` the sections `files_lib` and `files_include` must be specified. WPM uses this sections to perform a conflict check before files are copied to the target directories and adds this into the WPM database.

```
files="lib include"
files_lib()
{
cat <<EOF
liblevmar.a
EOF
}
files_include()
{
cat <<EOF
levmar-2.1.3/lm.h
levmar-2.1.3/misc.h
EOF
}
```

**Remark:** This sections must be written carefully. They must exactly match the files installed by the `install` section. If the packager forgets one file, the conflict check leaks and the subprogram `wpm uninstall` can not clean-up.

At the end we define the sections which act as the user build interface via the `wpm query` subcommand.

```
package_flags()
{
 echo "-I$WPM_INSTDIR_include"
}
package_FLAGS()
{
 package_flags
}
package_libs()
{
 echo "$WPM_INSTDIR_lib/liblevmar.a"
}
package_LIBS()
{
 package_libs
}
```

### Build the `wpm` package

Now mainly steps from Section 4.2 are important. First we create the file `levmar-2.1.3-1/setup.wini` with the following content

```
nosource="yes"
wspec="levmar.wspec"
```

This instructs `wpm setup` that the software source is not inside the WPM package. Perform the set up step with

```
$ wpm setup ./levmar-2.1.3-1/
```

### Test the `wpm` package

We prepare a test database only for this purpose with

```
$ export WPM_DBPATH=$HOME/wpm-levmar-test/db
$ wpm initdb WPM_INSTDIR=$HOME/wpm-levmar-test/local  --nonfs
```

```
nfs       : no
multi-pkg : yes
```

WPM performs the build and installation of *levmar* with the following command

```
$ wpm install levmar-2.1.3-1.wpm
```

The target directory `~/wpm-levmar-test/local/levmar-2.1.3-1` contains the files

```
 include/levmar-2.1.3/lm.h
 include/levmar-2.1.3/misc.h
 lib/liblevmar.a
```

If this is the case, the `wspec` sections `build`, `install` and `files` seems to be ok. Now we check `files_lib` and `files_include` with the subcommand `wpm uninstall` as follows

```
$ wpm uninstall pkg=levmar-2.1.3-1
```

This step must end up with an empty but existing directory

`~/wpm-levmar-test/local/levmar-2.1.3-1`.

### 4.6.2 Source package of an application

This example mainly deals with difficulties of patch and configure of the software *Lua*[15]. It is recommend to read the last example in Section 4.6.1 first.

**Write the `wspec` file**

First we download the tarball `lua-5.0.tar.gz` and generate a signature of this file. The header may look as follows

```
Summary="Lua - light-weight programming language"
Name="lua"
Version="5.0"
Release="1"
Copyright="1994-2006 Lua.org, PUC-Rio"
Source="lua-5.0.tar.gz"
Checksum="Whirlpool digest: 23bf42ed99cd ...."
GetDirs="http://www.lua.org/ftp"
SourceFiles=""
requires="[dlopen] math"
provides="lua-5.0"
```

[15]Lua - light-weight programming language, URL: http://www.lua.org

```
URL="http://www.lua.org"
Distribution=""
Vendor="Tecgraf PUC-Rio"
Packager="Timo Streckenbach <strecken@wias-berlin.de>"
```

We want to package the library and the executable program of *Lua*. The executable requires a math library `-lm` which is defined in the prototype file `lua-5.0/config` of the software. And furthermore *Lua* supports loading of dynamic libraries optionally (requires `-ldl`).

To satisfy these dependencies in a portable way, we defined the header varible `requires` above. If the dependency `dlopen` is available, the package should pass this to the software and `math` is not an optional dependency[16].

A suitable way to build this software with WPM is the following

1. copy the prototype file `lua-5.0/config` into the `WPM_SOURCE_DIR`
2. replace the prototype setting with *Makefile* varibales `WPM_<name>`
3. prepend the definitions to this file during configure time and replace the prototype `lua-5.0/config`
4. start the compilation with *make* in `WPM_BUILD_DIR`

Before we will write the `wspec` file it is recommend to make a test compilation of the desired source files. Ensure that your compiler vendor will work with `wpm-build check` (details in Section 2.7). Retrieve the build tools and flags with `wpm-build flags` as described in the last example of Section 4.6.1 and put it inside the prototype file `lua-5.0/config`. For example replace the value of variable `CC` as follows

```
 CC = ‘wpm-build flags --cc‘
```

Then try to compile the software with

```
$ cd lua-5.0
$ make
```

We now write the the `wspec` section `configure` as follows, which realizes the third step

```
configure_require_wpm_com="yes"
configure()
{
 local_PCFG="$WPM_BUILD_DIR/config"
```

[16]The next example in Section 4.6.3 deals with this dependencies.

```
 echo "### BEGIN generated from WPM" > $local_PCFG

cat >>$local_PCFG <<EOF
WPM_ranlib  = `$WPM_COM --ranlib`
WPM_ar      = `$WPM_COM --ar --arflags`
WPM_cc      = `$WPM_COM --cc`
WPM_cflags  = `$WPM_COM --cflags`
WPM_LDFLAGS = `$WPM_COM --ldflags`
WPM_warn    =
EOF

case $WPM_HOST in
 *cygwin*|*mingw*)
  echo "WPM_loadlib = -DUSE_DLL" >> $local_PCFG
  echo "WPM_libs    = `$WPM_QUERY_math --libs`" >> $local_PCFG
  ;;
 *)
  if test -x "$WPM_QUERY_dlopen" ; then
   echo "WPM_loadlib = -DUSE_DLOPEN" >> $local_PCFG
   echo "WPM_libs    = `$WPM_QUERY_dlopen --libs` \\" >> $local_PCFG
   echo "              `$WPM_QUERY_math --libs`" >> $local_PCFG
  else
   echo "WPM_libs    = `$WPM_QUERY_math --libs`" >> $local_PCFG
  fi
  ;;
esac

cat >>$local_PCFG <<EOF
WPM_HOME   = $WPM_SOURCE_DIR
WPM_BINDIR = $WPM_INSTDIR_bin
WPM_LIBDIR = $WPM_INSTDIR_lib
WPM_INCDIR = $WPM_INSTDIR_include
WPM_MANDIR = $WPM_INSTDIR_man
EOF

 cat $WPM_SOURCE_DIR/config_opt >> $local_PCFG
}
```

The first part sets the WPM standard build tools and flags and the last part prepares the installation. And the mid part handles the requirements. The math dependency is always presented with `WPM_QUERY_math`, but the support of dynamic libraries is optionally and depends from the system indicated by `WPM_HOST`. Basically the packager can figure out the existence of a dependency in a `wspec` section in the following way

```
  if test -x "$WPM_QUERY_dlopen" ; then
   ... available ...
  else
   ... missing ...
  fi
```

The following sections are more or less standard usage of *make*. Only the command `make install` needs some postprocessing.

```
build()
{
 make all
}
clean()
{
 make clean
}
install()
{
 make install
 `$WPM_COM --ranlib` $WPM_INSTDIR_lib/liblua*.a
 chmod a+x $WPM_INSTDIR_bin/lua$WPM_EXEEXT
 chmod a+x $WPM_INSTDIR_bin/luac$WPM_EXEEXT

 (cd etc; make bin2c)
 mv $WPM_BUILD_DIR/etc/bin2c$WPM_EXEEXT $WPM_INSTDIR_bin
 chmod +x $WPM_INSTDIR_bin/bin2c$WPM_EXEEXT
}
```

Like the example in the last Section the following part of the wspec file tells WPM which files will be installed. WPM defines the variables `WPM_INSTDIR_include`, `WPM_INSTDIR_man`, `WPM_INSTDIR_lib`, `WPM_INSTDIR_bin` and `WPM_INSTDIR_include` and creates these folders depending from section files. Furthermore depending from `files` the sections `files_include`, `files_man`, `files_lib`, `files_bin` and `files_include` must be specified. WPM uses this sections to perform a conflict check before files are copied to the target directories and adds this into the WPM database.

```
 files="include man lib bin"
 files_include()
 {
 cat <<EOF
 lua/lua.h
 lua/lualib.h
 lua/lauxlib.h
 EOF
 }
 files_man()
 {
 cat <<EOF
 man1/lua.1
 man1/luac.1
 EOF
 }
```

```
files_lib()
{
cat <<EOF
liblua.a
liblualib.a
EOF
}
files_bin()
{
cat <<EOF
lua$WPM_EXEEXT
luac$WPM_EXEEXT
bin2c$WPM_EXEEXT
EOF
}
```

The following section `files_source` is only relevant for degugging this software and will be installed automatically if the user specifies the `WPM_CONFIG=dbg`; see also 3.4.1.

```
files_source()
{
cat <<EOF
src/lapi.h
src/lcode.h
...
src/lapi.c
src/lcode.c
...
EOF
}
```

**Remark:** This sections must be written carefully. They must exactly match the files installed by the `install` section. If the packager forgets one file, the conflict check leaks and the subprogram `wpm uninstall` can not clean-up.

At the end we define the sections which act as the user build interface via the `wpm query` subcommand.

```
package_flags()
{
 echo "-I$WPM_INSTDIR_include -I$WPM_INSTDIR_include/lua"
}
package_FLAGS()
{
 package_flags
}
package_libs()
{
 echo "$WPM_INSTDIR_lib/liblua.a $WPM_INSTDIR_lib/liblualib.a"
}
package_LIBS()
{
 if test -x "$WPM_QUERY_dlopen" ; then
  echo "‘package_libs‘ ‘$WPM_QUERY_dlopen --libs‘ \
                     ‘$WPM_QUERY_math --libs‘"
 else
  echo "‘package_libs‘ ‘$WPM_QUERY_math --libs‘"
 fi
}
```

### Build the `wpm` package

Like described is the last example we now first create the file `lua-5.0-1/setup.wini` with the following content

```
nosource="yes"
files="config_opt:README"
wspec="lua.wspec"
```

This instructs `wpm setup` that the software source is not inside the WPM package, but we put two additional files `config_opt` and `README` into the WPM package. Perform the set up step with

```
$ wpm setup ./lua-5.0-1/
```

### Test the `wpm` package

Repeat the same steps as described in the last example (s. Section 4.6.1). On some systems the linker will properly fail because of missing `CFLAGS` in the *Makefile*'s in `lua-5.0/src/lua` and `lua-5.0/src/luac`. A possible workaround is to patch the desired files as follows

```
patch()
{
  local_patch_file="$WPM_BUILD_DIR/src/lua/Makefile"
  if test ! -f ${local_patch_file}.patch ; then
   cp $local_patch_file ${local_patch_file}.patch
   cat ${local_patch_file}.patch | \
     sed -e "s/\$(CC) -o/\$(CC) \$(CFLAGS) -o/g" > $local_patch_file
   echo "patch installed (for $local_patch_file)"
  fi

  local_patch_file="$WPM_BUILD_DIR/src/luac/Makefile"
  if test ! -f ${local_patch_file}.patch ; then
   cp $local_patch_file ${local_patch_file}.patch
   cat ${local_patch_file}.patch | \
     sed -e "s/\$(CC) -o/\$(CC) \$(CFLAGS) -o/g" > $local_patch_file
   echo "patch installed (for $local_patch_file)"
  fi
}
```

This `wspec` section will backup both files to `Makefil.patch` and edits the original files with *sed*[17].

### 4.6.3 Virtual package

As mentioned in Section 4 a virtual package doesn't build and install software. This package tries to package the following system software and make this accessable with `wpm query`.

| provides | description |
|---|---|
| `pthread` | POSIX Threads Programming |
| `gl` | OpenGL (Open Graphics Library) and Utility Library (GLU) |
| `dlopen` | gain access to an executable object file |
| `math` | ISO C standard mathematical library |
| `termcap` | text user interfaces (TUI) |
| `lapackblas` | linear algebra package (lapack) and basic linear algebra subprograms (blas) |

Here we deside to specify flags and libraries for each system software. During package installation a *configure* script tries to link against this independently. If *configure* fails on a system package, it will be ignored and the script continues with the next one. All special options of the system packages are defined in the file `pkg_config/arch_defaults.sh_inc`. The file `pkg_config.m4` contains macros which are required by the *autoconf* source file `configure.ac`. If the packager makes

[17]Unix Stream EDitor.

changes in one of these files, then a regeneration of the *configure* script should be done by calling

```
$ cd ./vsys-1.0-1/pkg_config
$ ./bootstrap.sh
```

**Write the `wspec` file**

The header may look as follows

```
Summary="virtual system-package"
Name="vsys"
Version="1.0"
Release="1"
Copyright=""
Source="README"
GetDirs=""
SourceFiles="pkg_config"
SourceSubDir="pkg_config"
requires=""
provides="vsys-1.0"
URL=""
Distribution=""
Vendor="Timo Streckenbach"
Packager="Timo Streckenbach <strecken@wias-berlin.de>"
```

The next `wspec` sections becomes more important.

```
configure_require_wpm_com="yes"
configure()
{
 export WPM_COM
 export WPM_HOST
 export WPM_VENDOR

 SH_LIBDIR="$wpm_abs_bindir/`basename $ld_libdir`" ; export SH_LIBDIR

 if ./configure ; then : ; else exit 1 ; fi

 # post-set
 WPM_REQUIRES="`$WPM_BUILD_DIR/pkg_config --check_ok`"
 if test -z "$WPM_REQUIRES" ; then
  echo "error: no packages found!"
  exit 1
 fi
 for local_configure_rq in $WPM_REQUIRES ; do
  wpm_dependency_built_with_packages=\
   "$wpm_dependency_built_with_packages $local_configure_rq-sys"
 done
}
```

This section makes some WPM variables public in *configure* and starts the script. The script will perform checking the desired system packages and generates WPM database files. Finally *configure* generates an executable script `pkg_config` which contains the list of successful detected system software. The rest of the section beginning with the comment `post-set` checks the result and modifies a WPM internal dependency variable for filling the flag `--requires_resolved`; see also Section 3.3.3.

The last non empty section will be the following, which simply puts the generated WPM database files in the database folder.

```
install()
{
 cp wpm-*.sh $WPM_DBPATH
}
```

### Build the `wpm` package

We don't need a `setup.wini` for this package. All information to package is included in the `wspec` file. Hence we call

```
$ wpm setup ./vsys-1.0-1/virtual.wspec
```

### Test the `wpm` package

WPM performs the installation of the virtual package in the specified `WPM_DBPATH` with the following command.

```
$ wpm install vsys-1.0-1.wpm
```

After success the packager should get a list as follows

```
$ wpm query
dlopen-sys
gl-sys
lapackblas-sys
math-sys
pthread-sys
```

```
termcap-sys
vsys-1.0-1
```

The system packages named with a trailing `-sys` are required by `vsys-1.0-1`. So an attempt to uninstall one should give the following result

```
$ wpm uninstall pkg=gl-sys
name    : gl
version : sys
release : 1
provides: gl
uninstall failed: depending package(s) <vsys-1.0-1>
```

The packager must first uninstall `vsys-1.0-1` and after that an uninstall of each system package is possible.

# 5 How do I . . .

## 5.1 install a `wpm` package?

This Section is mainly a short overview of Section 3. Also a good getting started is the example Section 3.6, which contains more details.

If you start from zero you first need to initialize a WPM database as described in Section 3.1. Call the default

```
$ wpm initdb
WPM_DBPATH: <HOME>/wpm-db
nfs       : yes
multi-pkg : yes
```

WPM creates only a database if the directory `~/wpm-db` doesn't exist. Now it is recommend to check the default vendor of your system with

```
$ wpm-build check
```

If this step fails, you may try a different vendor; see Section 2.7 for more details. The next command performs an installation of a package

```
$ wpm install <package>.wpm
```

The default target directory is `~/loacl`.

## 5.2 create a `wpm` package from a software?

The Section 4 deals with this question. Especially if you read the guidline at the beginning (Section 4.1), you will see the general steps. Depending from the question which type of package should be created, the examples in Section 4.6 are a good getting started. It is recommend to start with the example in Section 4.6.1.

## 5.3 add a new compiler to WPM?

The Section 2.8 describes the the way how to write your own vendor file. If you prefer testing of your own extension is a *Makefile*, the example in Section 2.4 is a good choice.

## 5.4 add a new system to WPM?

The supported systems are represented by subdirectories lacated in the WPM software

```
$ ls wpm/sys
aix cygwin mingw ...
```

Change to your desired system an call the following command to determine the system string

```
$ cd wpm
$ ./config.guess
powerpc-apple-darwin8.11.0
```

Then you must map your system string or a part of them to the new subdirectory name. This must be specified in the Bourne Shell function `wpm_dispatcher_arch2dir` of the file `wpm/sys/arch_dispatcher.sh_inc`. We got the system string

```
 powerpc-apple-darwin8.11.0
```

here, which matches the following line

```
 powerpc*-darwin*) echo "powerpc_darwin" ;;
```

For example if you need to separate these special version, you must insert a line before the more general one.

```
 powerpc*-darwin*8.11.*) echo "powerpc_darwin_811" ;;
 powerpc*-darwin*) echo "powerpc_darwin" ;;
```

This expects the system subdirectory `powerpc_darwin_811`. Now you may follow the Section 2.8 to write a new vendor file or simply copy one from other existing system directories and modify them. After this don't forget to create the file `default` in your system subdirectory. The must contain a valid vendor name.

### 5.5 change the vendor default?

Figure out the current system subdirectory with the command `wpm-build flags`. For example

```
$ wpm-build flags --cfg_file
... wpm/sys/powerpc_darwin/gnu40gfortran.sh_inc
```

Now change the valid vendor name to another one in the file

```
 ... wpm/sys/powerpc_darwin/default
```

The following command returns a valid list of valid vendor names

```
$ wpm-build list
```

# A Apple bundle with `wpm-bundle`

The program `wpm-bundle` is a command-line tool designed to create Mac OS X application bundles. Bundles are directory hierarchies in the file system and contains real files that can be manipulated by all file-based services [developer.apple.com, Bundle Programming Guide].

The command `wpm-bundle [Flags] <main-exe>` creates an application bundle directory with extension `.app` and copies the specified file `main-exe` into the folder

```
 <bundle>.app/Contents/MacOS
```

Furthermore the *information property list file* will be created automatically at

```
<bundle>.app/Contents/Info.plist
```

and the bundle resource files will be stored in the folder

```
<bundle>.app/Contents/Resources
```

Details of the command-line tool are following now:

- `wpm-bundle [options] <main-exe>...`

**Available options**

| | |
|---|---|
| `...--help` | list of flags |
| `...-bundle <name>` | creates a bundle `<name>.app` and defines `CFBundleName` in `Info.plist` (default: `<main-exe>.app`). |
| `...-info <string>` | sets info string by defining `CFBundleGetInfoString` in `Info.plist`. |
| `...-Vs <vers_s>` | sets short version by defining `CFBundleShortVersionString` in `Info.plist` (default: `1.0`). |
| `...-Vb <vers_b>` | sets bundle version by defining `CFBundleShortVersionString` in `Info.plist` (default: `--vers_s`) |
| `...-icon <file>` | copy icons file (extension `.icns`) into bundle and defines `CFBundleIconFile` in `Info.plist`. |
| `...-res <dir>` | copy content of directory `dir` into bundle resource directory. If `dir` ends with the subfolder `Resources`, then the hole bundle resource directory will be replaced by `dir`. Multiple usage of this flag is allowed. The bundle directory will be cleaned form `.svn` and `*~` by this flag. |
| `...-rescp <file>` | copy `file` into bundle resource directory. |