The l3build package
Checking and building packages

The LaTeX3 Project*

Released 2018-03-26

Contents

1 The l3build system

1.1 Introduction

The l3build system is a Lua script for building TeX packages, with particular emphasis on regression testing. It is written in cross-platform Lua code, so can be used by any modern TeX distribution with the `texlua` interpreter. A package for building with l3build can be written in any TeX dialect; its defaults are set up for LaTeX packages written in the DocStrip style. (Caveat: minimal testing has yet been performed for non-LaTeX packages.)

*E-mail: latex-team@latex-project.org

Test files are written as standalone TEX documents using the `regression-test.tex` setup file; documentation on writing these tests is discussed in Section 2.

Each package will define its own `build.lua` configuration file which both sets variables (such as the name of the package) and may also provide custom functions.

A standard package layout might look something like the following:

```
abc/
    abc.dtx
    abc.ins
    build.lua
    README.md
    support/
    testfiles/
```

Most of this should look fairly self-explanatory. The top level `support/` directory (optional) would contain any necessary files for compiling documentation, running regression tests, and so on.

The l3build system is also capable of building and checking *bundles* of packages. To avoid confusion, we refer to either a standalone package or a package within a bundle as a *module*.

For example, within the LATEX3 project we have the l3packages bundle which contains the xparse, xtemplate, etc., modules. These are all built and distributed as one bundle for installation, distribution *via* CTAN and so forth.

Each module in a bundle will have its own build script, and a bundle build script brings them all together. A standard bundle layout would contain the following structure.

```
mybundle/
        build.lua
        support/
        yyy/                    zoo/
            build.lua               build.lua
            README.md               README.md
            testfiles/              testfiles/
            yyy.dtx                 zoo.dtx
            yyy.ins                 zoo.ins
```

All modules within a bundle must use the same build script name.

In a small number of cases, the name used by CTAN for a module or bundle is different from that used in the installation tree. For example, the LATEX 2$_\varepsilon$ kernel is called latex-base by CTAN but is located inside ⟨*texmf*⟩`/tex/latex/base`. This can be handled by using `ctanpkg` for the name required by CTAN to override the standard value.

The `testfiles/` folder is local to each module, and its layout consists of a series of regression tests with their outputs.

```
testfiles/
         test1.lvt
         test1.tlg
         ...
         support/
                 my-test.cls
```

Again, the `support/` directory contains any files necessary to run some or all of these tests.

When the build system runs, it creates a directory `build/` for various unpacking, compilation, and testing purposes. For a module, this build folder can be in the main directory of the package itself, but for a bundle it should be common for the bundle itself and for all modules within that bundle. A `build/` folder can be safety deleted; all material within is re-generated for each command of the l3build system.

1.2 Main build commands

In the working directory of a bundle or module, the following commands can be executed:

- `check`
- `check` ⟨*name(s)*⟩
- `cmdcheck`
- `clean`
- `ctan`
- `doc` ⟨*name(s)*⟩
- `install`
- `save` ⟨*name(s)*⟩
- `tag`
- `unpack`

These commands are described below.

As well as these commands, the system recognises the options

- `--config` (`-c`) Configuration(s) to use for testing
- `--date` Date to use when tagging data
- `--dry-run` Runs the `install` target but does not copy any files: simply lists those that would be installed
- `--engine` (`-e`) Sets the engine to use for testing
- `--epoch` Sets the epoch for typesetting and testing
- `--first` Name of the first test to run
- `--force` (`-f`) Force checks to run even if sanity checks fail, *e.g.* when `--engine` is not given in `{"pdftex", "xetex", "luatex"}`
- `--halt-on-error` (`-H`) Specifies that checks should stop as soon as possible, rather than running all requested tests; the difference file is printed in the terminal directly in the case of failure
- `--last` Name of the last test to run
- `--pdf` (`-p`) Test PDF file against a reference version rather than using a log comparison
- `--quiet` (`-q`) Suppresses output from unpacking
- `--rerun` Run tests without unpacking/set up

- `--shuffle` Shuffle the order in whichs tests run
- `--texmfhome` Sets the location of the user tree for installing

`$ texlua build.lua check`

The `check` command runs the entire test suite. This involves iterating through each `.lvt` file in the test directory (specified by the `testfiledir` variable), compiling each test in a "sandbox" (a directory specified by `testdir`), and comparing the output against each matching predefined `.tlg` file.

If changes to the package or the typesetting environment have affected the results, the check for that file fails. A `diff` of the expected to actual output should then be inspected to determine the cause of the error; it is located in the `testdir` directory (default `builddir .. "/test"`).

On Windows, the `diff` program is not available and so `fc` is used instead (generating an `.fc` file). Setting the environmental variables `diffexe` and `diffext` can be used to adjust the choice of comparison made: the standard values are

Windows `diffext = fc`, `diffexe = fc /n`

*nix `diffext = diff`, `diffexe = diff -c --strip-trailing-cr`

The following files are moved into the "sandbox" for the `check` process:

- all `installfiles` after unpacking;
- all `checkfiles` after unpacking;
- any files in the directory `testsuppdir`;
- any files that match `checksuppfiles` in the `supportdir`.

This range of possibilities allow sensible defaults but significant flexibility for defining your own test setups.

Checking can be performed with any or all of the 'engines' `pdftex`, `xetex`, and `luatex`. By default, each test is executed with all three, being compared against the `.tlg` file produced from the `pdftex` engine (these defaults are controlled by the `checkengines` and `stdengine` variable respectively). The format used for tests can be altered by setting `checkformat`: the default setting `latex` means that tests are run using *e.g.* `pdflatex`, whereas setting to `plain` will run tests using *e.g.* `pdftex`. (Currently, this should be one of `latex` or `plain`.) To perform the check, the engine typesets each test up to `checkruns` times. More detail on this in the documentation on `save`. Options passed to the binary are defined in the variable `checkopts`.

By default, `texmf` trees are searched for input files when checking. This can be disabled by setting `checksearch` to `false`: isolation provides confidence that the tests cannot accidentally be running with incorrect files installed in the main distribution or `hometexmf`.

`$ texlua build.lua check ⟨name(s)⟩`

Checks only the test ⟨*name(s)*⟩`.lvt`. All engines specified by `checkengines` are tested unless the command line option `--engine` (or `-e`) has been given to limit testing to a single engine. Normally testing is preceded by unpacking source files and copying the

result plus any additional support to the test directory: this may be skipped using the `-s` option.

```
$ texlua build.lua check -p
```

Rather than the log-based checking carried out by the standard `check` target, running with the `-p` option carries out a binary comparison of the PDF files produced by typesetting against those saved in `testfiledir`.

This functionality requires TEX Live 2016 or later as it needs support from the engines not available in earlier releases.

```
$ texlua build.lua clean
```

This command removes all temporary files used for package bundling and regression testing. In the standard layout, these are all files within the directories defined by `localdir`, `testdir`, `typesetdir` and `unpackdir`, as well as all files defined in the `cleanfiles` variable in the same directory as the script. The defaults are `.pdf` files from typesetting (`doc`) and `.zip` files from bundling (`ctan`).

```
$ texlua build.lua ctan
```

Creates an archive of the package and its documentation, suitable for uploading to CTAN The archive is compiled in `distribdir`, and if the results are successful the resultant `.zip` file is moved into the same directory as the build script. If `packtdszip` is set true then the building process includes a `.tds.zip` file containing the 'TEX Directory Structure' layout of the package or bundle. The archive therefore may contain two 'views' of the package:

```
abc.zip/
        abc/
            abc.dtx
            abc.ins
            abc.pdf
            README.md
        abc.tds.zip/
                    doc/latex/abc/
                                    abc.pdf
                                    README.md
                  source/latex/abc/
                                    abc.dtx
                                    abc.ins
                    tex/latex/abc/
                                    abc.sty
```

The files copied into the archive are controlled by a number of variables. The 'root' of the TDS structure is defined by `tdsroot`, which is `"latex"` by default. Plain users would redefine this to `"plain"` (or perhaps `"generic"`), for example. The build process for a `.tds.zip` file currently assumes a 'standard' structure in which all extracted files should be placed inside the `tex` tree in a single directory, as shown above. If the module includes any BIBTEX or MakeIndex styles these will be placed in the appropriate subtrees.

The `doc` tree is constructed from:

- all files matched by `demofiles`,
- all files matched by `docfiles`,
- all files matched by `typesetfiles` with their extension replaced with `.pdf`,
- all files matched by `textfiles`,
- all files matched by `bibfiles`.

The `source` tree is constructed from all files matched by `typesetfiles` and `sourcefiles`. The `tex` tree from all files matched by `installfiles`.

Files that should always be excluded from the archive are matched against the `excludefiles` variable; by default this is `{"*~"}`, which match Emacs' autosave files.

Binary files should be specified with the `binaryfiles` variable (default `{"*.pdf", "*.zip"}`); these are added to the zip archive without normalising line endings (text files are automatically converted to Unix-style line endings).

To create the archive, by default the binary `zipexe` is used (`"zip"`) with options `zipopts` (`-v -r -X`). The intermediate build directories `ctandir` and `tdsdir` are used to construct the archive.

```
$ texlua build.lua doc
```

Compiles documentation files in the `typesetdir` directory. In the absence of one or more file names, all documentation is typeset; a file list may be given at the command line for selective typesetting. If the compilation is successful the `.pdf` is moved back into the main directory.

The documentation compilation is performed with the `typesetexe` binary (default `pdflatex`), with options `typesetopts`. Additional TEX material defined in `typesetcmds` is passed to the document (e.g., for writing `\\PassOptionsToClass{l3doc}{letterpaper}`, and so on—note that backslashes need to be escaped in Lua strings).

Files that match `typesetsuppfiles` in the `support` directory (`supportdir`) are copied into the `build/local` directory (`localdir`) for the typesetting compilation process. Additional dependencies listed in the `typesetdeps` variable (empty by default) will also be installed.

Source files specified in `sourcefiles` and `typesetsourcefiles` are unpacked before the typesetting takes place. (In most cases `typesetsourcefiles` will be empty, but may be used where there are files to unpack *only* for typesetting.)

If `typesetsearch` is `true` (default), standard `texmf` search trees are used in the typesetting compilation. If set to false, *all* necessary files for compilation must be included in the `build/local` sandbox.

```
$ texlua build.lua doc ⟨name(s)⟩
```

Typesets only the files with the ⟨*name(s)*⟩ given, which should be the root name without any extension.

```
$ texlua build.lua install
```

Copies all package files (defined by `installfiles`) into the user's home `texmf` tree in the form of the TEX Directory Structure. The location of the user tree can be adjusted using the `--texmfhome` swtich: the standard setting is the location set as `TEXMFHOME`.

```
$ texlua build.lua save ⟨name(s)⟩
```

This command runs through the same execution as `check` for a specific test(s) ⟨*name(s)*⟩`.lvt`. This command saves the output of the test to a `.tlg` file. This file is then used in all subsequent checks against the ⟨*name*⟩`.lvt` test.

If the `--engine` (or `-e`) is specified (one of `pdftex`, `xetex`, or `luatex`), the saved output is stored in ⟨*name*⟩.⟨*engine*⟩`.tlg`. This is necessary if running the test through a different engine produces a different output. A normalisation process is performed when checking to avoid common differences such as register allocation; full details are listed in section 1.11.

If the `recordstatus` variable is set `true`, additional information will be added to the `.tlg` to record the "exit status" of the typesetting compilation of the `.lvt` file. If the typesetting compilation completed without throwing an error (due to TEX programming errors, for example), the "exit status" is zero, else non-zero.

```
$ texlua build.lua save -p ⟨name(s)⟩
```

This version of `save` will store the PDF files produced from ⟨*name(s)*⟩`.lvt` in addition to the `.tlg` file, and thus allows binary comparison of the result of typesetting.

This functionality requires TEX Live 2016 or later as it needs support from the engines not available in earlier releases.

```
$ texlua build.lua manifest
```

Generates a 'manifest' file which lists the files of the package as known to l3build. The filename of this file (by default `"MANIFEST.md"`) can be set with the variable `manifestfile`.

The intended purpose of this manifest file is to include it within a package as metadata. This would allow, say, for the copyright statement for the package to refer to the manifest file rather than requiring the author to manually keep a file list up-to-date in multiple locations. The manifest file can be structured and documented with a degree of flexibility. Additional information is described in Section 5.5.

In order for `manifest` to detect derived and typeset files, it should be run *after* running `unpack` and `doc`. If `manifest` is run after also running `ctan` it will include the files included in the CTAN and TDS directories as well.

Presently, this means that if you wish to include an up-to-date manifest file as part of a `ctan` release, you must run `ctan` / `manifest` / `ctan`. Improvements to this process are planned for the future.

```
$ texlua build.lua tag
```

Modifies the content of files specified by `tagfiles` to allow automatic updating of the release tag and date. The tag is given as a command line option, whilst the optional date should be passed using `--date` (`-d`); if not given, the date will default to the current date in ISO format (YYYY-MM-DD). As detailed below, the standard set up has no search pattern defined for this target and so no action will be taken *unless* tag substitution is set up.

```
$ texlua build.lua unpack
```

This is an internal target that is normally not needed on user level. It unpacks all files into the directory defined by `unpackdir`. This occurs before other build commands such as `doc`, `check`, etc.

The unpacking process is performed by executing the `unpackexe` (default `tex`) with options `unpackopts` on all files defined by the `unpackfiles` variable; by default, all files that match `{"*.ins"}`.

If additional support files are required for the unpacking process, these can be enumerated in the `unpacksuppfiles` variable. Dependencies for unpacking are defined with `unpackdeps`.

By default this process allows files to be accessed in all standard `texmf` trees; this can be disabled by setting `unpacksearch` to `false`.

1.3 The `build.lua` file

The `build.lua` file used to control l3build is a simple Lua file which is read during execution. In the current release of l3build, `build.lua` is read automatically and can access all of the global functions provided by the script. Thus it may contain a simple list of variable settings *or* additionally custom code to change the build process. A number of example scripts are given in Section 1.4.

1.4 Example build scripts

An example of a standalone build script for a package that uses self-contained `.dtx` files is shown in Figure 1. Here, the `module` only is defined, and since it doesn't use `.ins` files so the variable `unpackfiles` is redefined to run `tex` on the `.dtx` files instead to generate the necessary `.sty` files. There are some PDFs in the repository that shouldn't be part of a CTAN submission, so they're explicitly excluded, and here unpacking is done 'quietly' to minimise console output when building the package.

An example of a bundle build script for l3packages is shown in Figure 2. Note for LaTeX3 we use a common file to set all build variables in one place, and the path to the `l3build.lua` script is hard-coded so we always use our own most recent version of the script. An example of an accompanying module build script is shown in Figure 3.

1.5 Backward compatibly

Earlier releases of l3build required that the last line of `build.lua` ran the main script, *i.e.* that `build.lua` was what the user called rather than `l3build.lua`. To allow scripts to support both forms *for the transition*, a simple test may be included as showing in Figure 4.

Note that in time support for loading l3build by calling the `build.lua` script *may* be removed: the recommended approach for new scripts is to run l3build.

```
1 -- Build configuration for breqn
2
3 module = "breqn"
4
5 unpackfiles = {"*.dtx"}
6 excludefiles = {"*/breqn-abbr-test.pdf",
7                 "*/eqbreaks.pdf"}
8 unpackopts  = "-interaction=batchmode"
```

Figure 1: The build configuration for the breqn package.

```
-- Build script for LaTeX3 "l3packages" files

-- Identify the bundle: there is no module as this is the "driver"
bundle = "l3packages"

-- Location of main directory: use Unix-style path separators
maindir = ".."
```

Figure 2: The build script for the l3packages bundle.

```
-- Build script for LaTeX3 "xparse" files

-- Identify the bundle and module:
bundle = "l3packages"
module = "xparse"

-- Location of main directory: use Unix-style path separators
-- Should match that defined by the bundle.
maindir = "../.."
```

Figure 3: The build script for the xparse module.

```
if not release_date then
  dofile(kpse.lookup("l3build.lua"))
end
```

Figure 4: Final lines for a `build.lua` script usable with both older and newer releases of l3build.

1.6 Variables

This section lists all variables defined in the `l3build.lua` script that are available for customisation.

1.7 Interaction between tests

Tests are run in a single directory, so whilst they are may be isolated from the system TEX tree they do share files. This may be significant if installation-type files are generated during a test, for example by a `filecontents` environment in LATEX. Typically, you should set up your tests such that they do not use the same names for such files: this may lead to variable outcomes depending on the order in which tests are run.

Variable	Default	Description
`module`	`""`	The name of the module
`bundle`	`""`	The name of the bundle in which the module belongs (where relevant)
`ctanpkg`	`module/bundle`	Name of the CTAN package matching this module
`modules`	`{}`	The list of all modules in a bundle (when not auto-detecting)
`exclmodules`	`{}`	Directories to be excluded from automatic module detection
`maindir`	`"."`	Top level directory for the module/bundle
`docfiledir`	`"."`	Directory containing documentation files
`sourcfiledir`	`"."`	Directory containing source files
`supportdir`	`maindir .. "/support"`	Directory containing general support files
`testfiledir`	`"./testfiles"`	Directory containing test files
`testsuppdir`	`testfiledir .. "/support"`	Directory containing test-specific support files
`builddir`	`maindir .. "/build"`	Directory for building and testing
`distribdir`	`builddir .. "/distrib"`	Directory for generating distribution structure
`localdir`	`builddir .. "/local"`	Directory for extracted files in "sandboxed" TEX runs
`testdir`	`builddir .. "/test"`	Directory for running tests
`typesetdir`	`builddir .. "/doc"`	Directory for building documentation
`unpackdir`	`builddir .. "/unpack"`	Directory for unpacking sources
`ctandir`	`distribdir .. "/ctan"`	Directory for organising files for CTAN
`tdsdir`	`distribdir .. "/tds"`	Directory for organised files into TDS structure
`tdsroot`	`"latex"`	Root directory of the TDS structure for the bundle/module to be installed into
`auxfiles`	`{"*.aux", "*.lof", "*.lot", "*.toc"}`	Secondary files to be saved as part of running tests
`bibfiles`	`{"*.bib"}`	BIBTEX database files
`binaryfiles`	`{"*.pdf", "*.zip"}`	Files to be added in binary mode to zip files
`bstfiles`	`{"*.bst"}`	BIBTEX style files
`checkfiles`	`{ }`	Extra files unpacked purely for tests
`checksuppfiles`		Files needed for performing regression tests
`cleanfiles`	`{"*.log", "*.pdf", "*.zip"}`	Files to delete when cleaning
`demofiles`	`{}`	Files which show how to use a module
`docfiles`	`{}`	Files which are part of the documentation but should not be typeset

Variable	Default	Description
excludefiles	{"*~"}	Files to ignore entirely (default for Emacs backup files)
installfiles	{"*.sty","*.cls"}	Files to install to the text area of the texmf tree
makeindexfiles	{"*.ist"}	MakeIndex files to be included in a TDS-style zip
scriptfiles	{ }	Files to install to the scripts area of the texmf tree
scriptmanfiles	{ }	Files to install to the doc/man area of the texmf tree
sourcefiles	{"*.dtx", "*.ins", "*-????-??-??.sty"}	Files to copy for unpacking
tagfiles	{"*.dtx"}	Files for automatic tagging
textfiles	{"*.md", "*.txt"}	Plain text files to send to CTAN as-is
typesetdemofiles	{}	Files to typeset before the documentation for inclusion in main documentation files
typesetfiles	{"*.dtx"}	Files to typeset for documentation
typesetsuppfiles	{}	Files needed to support typesetting when "sandboxed"
typesetsourcefiles	{}	Files to copy to unpacking when typesetting.
unpackfiles	{"*.ins"}	Files to run to perform unpacking.
unpacksuppfiles	{}	Files needed to support unpacking when "sandboxed"
bakext	".bak"	Extension of backup files
dviext	".dvi"	Extension of DVI files
lvtext	".lvt"	Extension of test files
tlgext	".tlg"	Extension of test file output
lveext	".lve"	Extension of auto-generating test file output
logext	".log"	Extension of checking output, before processing it into a .tlg
pdfext	".pdf"	Extension of PDF file for checking and saving
psext	".ps"	Extension of PostScript files
checkdeps	{}	List of dependencies for running checks
typesetdeps	{}	List of dependencies for typesetting docs
unpackdeps	{}	List of dependencies for unpacking
checkengines	{"pdftex", "xetex", "luatex"}	Engines to check with check by default
stdengine	"pdftex"	Engine to generate .tlg file from
checkformat	"latex"	Format to use for tests
checkconfigs	{}	Configurations to use for tests
typesetexe	"pdflatex"	Executable for compiling doc(s)
unpackexe	"tex"	Executable for running unpack
zipexe	"zip"	Executable for creating archive with ctan
checkopts	"-interaction=nonstopmode"	Options based to engine when running checks.
typesetopts	"-interaction=nonstopmode"	Options based to engine when typesetting.
unpackopts	""	Options based to engine when unpacking.
zipopts	"-v -r -X"	Options based to zip program.
checksearch	true	Switch to search the system texmf for during checking
typesetsearch	true	Switch to search the system texmf for during typesetting
unpacksearch	true	Switch to search the system texmf for during unpacking
glossarystyle	"gglo.ist"	MakeIndex style file for glossary/changes creation
indexstyle	"gind.ist"	MakeIndex style for index creation

```
-- Special config for these tests
checksearch  = true
checkengines = {"xetex","luatex"}
testfiledir  = "testfiles-TU"
```

Figure 5: Example of using additional (or overriding) settings for configuring tests in a different subdirectory.

Variable	Default	Description
`biberexe`	`"biber"`	Biber executable
`biberopts`	`""`	Biber options
`bibtexexe`	`"bibtex8"`	BibTeX executable
`bibtexopts`	`"-W"`	BibTeX options
`makeindexexe`	`"makeindex"`	MakeIndex executable
`makeindexopts`	`""`	MakeIndex options
`forcecheckepoch`	`"true"`	Force epoch when running tests
`forcedocepoch`	`"false"`	Force epoch when typesetting
`asciiengines`	`{"pdftex"}`	Engines which should log as sure ASCII
`checkruns`	`1`	Number of runs to complete for a test before comparing the log
`epoch`	`1463734800`	Epoch (Unix date) to set for test runs
`flatten`	`true`	Switch to flatten any source structure when sending to CTAN
`maxprintline`	`79`	Length of line to use in log files
`packtdszip`	`false`	Switch to build a TDS-style zip file for CTAN
`typesetcmds`	`""`	Instructions to be passed to TeX when doing typesetting.
`typsetcycles`	`3`	Number of cycles of typesetting to carry out.
`recordstatus`	`false`	Switch to include error level from test runs in `.tlg` files
`manifestfile`	`"MANIFEST.md"`	Filename to use for the manifest file.

1.8 Multiple sets of tests

In most cases, a single set of tests will be appropriate for the module, with a common set of configuration settings applying. However, there are situations where you may need entirely independent sets of tests which have different setting values, for example using different formats or where the entire set will be engine-dependent. To support this, l3build offers the possibility of using multiple configurations for tests. This is supported using the `checkconfigs` table. This is used to list the names of each configuration (`.lua` file) which will be used to run tests.

For example, for the core LaTeX 2_ε tests the main test files are contained in a directory `testfiles`. To test font loading for XeTeX and LuaTeX there are a second set of tests in `testfiles-TU` which use the short `build-TU.lua` file shown in Figure 5. To run both sets of tests, the main `build.lua` file contains the setting `checkconfigs = {"build", "config-TU"}`. This will cause l3build to run first using no additional settings (*i.e.* reading the normal `build.lua` file alone), then running *also* loading the settings from `config-TU.lua`.

To allow selection of a one or more configurations, and to allow saving of `.tlg` files in non-standard configurations, the `--config` (`-c`) option may be used. This works in the same way as `--engine`: it takes a comma list of configurations to apply, overriding `checkconfigs`.

1.9 Dependencies

If you have multiple packages that are developed separately but still interact in some way, it's often desirable to integrate them when performing regression tests. For LaTeX3, for example, when we make changes to l3kernel it's important to check that the tests for l3packages still run correctly, so it's necessary to include the l3kernel files in the build process for l3packages.

In other words, l3packages is *dependent* on l3kernel, and this is specified in l3build by setting appropriately the variables `checkdeps`, `typesetdeps`, and `unpackdeps`. The relevant parts of the LaTeX3 repository is structured as the following.

```
l3/
   l3kernel/
              build.lua
              expl3.dtx
              expl3.ins
              ...
              testfiles/
   l3packages/
              build.lua
              xparse/
                     build.lua
                     testfiles/
                     xparse.dtx
                     xparse.ins
   support/
```

For LaTeX3 build files, `maindir` is defined as top level folder `l3`, so all support files are located here, and the build directories will be created there. To set l3kernel as a dependency of l3package, within `l3packages/xparse/build.lua` the equivalent of the following is set:

```
maindir = "../.."
checkdeps = {maindir .. "/l3kernel"}
```

This ensures that the l3kernel code is included in all processes involved in unpacking and checking and so on. The name of the script file in the dependency is set with the `scriptname` variable; by default these are `"build.lua"`.

1.10 Non-standard source layouts

A variety of source layouts are supported. In general, a "flat" layout with all source files "here" is most convenient. However, l3build supports placement of both code and documentation source files in other locations using the `sourcefiledir` and `docfiledir` variables. For pre-built trees, the glob syntax `**/*.`⟨*ext*⟩ may be useful in these cases: this enables recursive searching in the appropriate tree locations. With the standard

settings, this structure will be removed when creating a CTAN release: the variable `flatten` may be used to control this behavior.

A series of example layouts and matching `build.lua` files are available from https://github.com/latex3/l3build/tree/master/examples.

1.11 Output normalisation

To allow test files to be used between different systems (*e.g.* when multiple developers are involved in a project), the log files are normalised before comparison during checking. This removes some system-dependent data but also some variations due to different engines. This normalisation consists of two parts: removing ("ignoring") some lines and modifying others to give consistent test. Currently, the following types of line are ignored:

- Lines before the `\START`, after the `\END` and within `\OMIT`/`\TIMO` blocks
- Entirely blank lines, including those consisting only of spaces.
- Lines containing file dates in format ⟨*yyyy*⟩/⟨*mm*⟩/⟨*dd*⟩.
- Lines starting `\openin` or `\openout`.

Modifications made in lines are:

- Removal spaces at the start of lines.
- Removal of `./` at start of file names.
- Standardisation of the list of units known to TEX (pdfTEX and LuaTEX add a small number of additional units which are not known to TEX90 or XETEX, (u)pTEX adds some additoonal non-standard ones)
- Standardisation of `\csname\endcsname␣` to `\csname\endcsname` (the former is formally correct, but the latter was produced for many years due to a TEX bug).
- Conversion of `on line` ⟨*number*⟩ to `on line ...` to allow flexibility in changes to test files.
- Conversion of register numbers in assignment lines `\`⟨*register*⟩`=\`⟨*type*⟩⟨*number*⟩ to `\`⟨*type*⟩⟨`...`⟩
- Conversion of box numbers in `\show` lines `> \box`⟨*number*⟩`=` to `> \box...=`
- Removal of some (u)pTEX data where it is equivalent to pdfTEX (`yoko direction`, `\displace 0.0`)

LuaTEX makes several additional changes to the log file. As normalising these may not be desirable in all cases, they are handled separately. When creating LuaTEX-specific test files (either with LuaTEX as the standard engine or saving a LuaTEX-specific `.tlg` file) no further normalisation is undertaken. On the other hand, for cross-engine comparison the following normalisation is applied:

- Removal of additional (unused) `\discretionary` points.
- Normalisation of some `\discretionary` data to a TEX90 form.
- Removal of `U+...` notation for missing characters.

- Removal of `display` for display math boxes (included by TeX90/pdfTeX/XeTeX).
- Removal of Omega-like `direction TLT` information.
- Removal of additional whatsit containing local paragraph information (`\localinterlinepenalty`, *etc.*).
- Rounding of glue set to four decimal places (glue set may be slightly different in LuaTeX compared to other engines).
- Conversion of low chars (0 to 31) to `^^` notation.

When making comparisons between 8-bit and Unicode engines it is useful to format the top half of the 8-bit range such that it appears in the log as `^^`⟨*char*⟩ (the exact nature of the 8-bit output is otherwise dependent on the active code page). This may be controlled using the `asciiengines` option. Any engines named here will use a `.tcx` file to produce only ASCII chars in the log output, whilst for other engines normalisation is carried out from UTF-8 to ASCII. If the option is set to an empty table the latter process is skipped: suitable for cases where only Unicode engines are in use.

2 Writing test files

Test files are written in a TeX dialect using the support file `regression-test.tex`, which should be `\input` at the very beginning of each test. Additional customisations to this driver can be included in a local `regression-test.cfg` file, which will be loaded automatically if found.

The macros loaded by `regression-test.tex` set up the test system and provide a number of commands to aid the production of a structured test suite. The basis of the test suite is to output material into the `.log` file, from which a normalised test output (`.tlg`) file is produced by the build command `save`. A number of commands are provided for this; they are all written in uppercase to help avoid possible conflicts with other package commands.

2.1 Metadata and structural commands

Any commands that write content to the `.log` file that should be ignored can be surrounded by `\OMIT` ... `\TIMO`. At the appropriate location in the document where the `.log` comparisons should start (say, after `\begin{document}`), the test suite must contain the `\START` macro. The test should then include `\AUTHOR{`⟨*authors details*⟩`}` in case a test file fails in the future and needs to be re-analysed.

Some additional diagnostic information can then be included as metadata for the conditions of the test. The desired format can be indicated with `\FORMAT{`⟨*format name*⟩`}`, and any packages or classes loaded can be indicated with

`\CLASS[`⟨*options*⟩`]{`⟨*class name, version*⟩`}`
`\PACKAGE[`⟨*options*⟩`]{`⟨*package name, version*⟩`}`

These do not provide information that is useful for automated checking; after all, packages change their version numbers frequently. Rather, including this information in a test indicates the conditions under which the test was definitely known to pass at a certain time in the past.

The \END command signals the end of the test (but read on). Some additional diagnostic information is printed at this time to debug if the test did not complete 'properly' in terms of mismatched brace groups or \if...\fi groups.

In a LaTeX document, \end{document} will implicitly call \END at the very end of the compilation process. If \END is used directly (replacing \end{document} in the test), the compilation will halt almost immediately, and various tasks that \end{document} usually performs will not occur (such as potentially writing to the various .toc files, and so on). This can be an advantage if there is additional material printed to the log file in this stage that you wish to ignore, but it is a disadvantage if the test relies on various auxiliary data for a subsequent typesetting run. (See the checkruns variable for how these tests would be test up.)

2.2 Commands to help write tests

A simple command \CHECKCOMMAND\⟨*macro*⟩ is provided to check whether a particular \⟨*macro*⟩ is defined, undefined, or equivalent to \relax. This is useful to flag either that internal macros are remaining local to their definitions, or that defined commands definitely are defined, or even as a reminder that commands you intend to define in a future package need to be tested once they appear.

\TYPE is used to write material to the .log file, like LaTeX's \typeout, but it allows 'long' input. The following commands are defined to use \TYPE to output strings to the .log file.

- \SEPARATOR inserts a long line of = symbols to break up the log output.
- \NEWLINE inserts a linebreak into the log file.
- \TRUE, \FALSE, \YES, \NO output those strings to the log file.
- \ERROR is *not* defined but is commonly used to indicate a code path that should never be reached.
- The \TEST{⟨*title*⟩}{⟨*contents*⟩} command surrounds its ⟨*contents*⟩ with some \SEPARATORs and a ⟨*title*⟩.
- \TESTEXP surrounds its contents with \TYPE and formatting to match \TEST; this can be used as a shorthand to test expandable commands.
- TODO: would a \TESTFEXP command (based on \romannumeral expansion) be useful as well?
- \BEGINTEST{⟨*title*⟩} ... \ENDTEST is an environment form of \TEST, allowing verbatim material, *etc.* to appear.

An example of some of these commands is shown following.

```
\TEST{bool_set,~lazy~evaluation}
 {
  \bool_set:Nn \l_tmpa_bool
   {
    \int_compare_p:nNn 1=1
    && \bool_lazy_any_p:n
     {
```

```
      { \int_compare_p:nNn 2=3 }
      { \int_compare_p:nNn 4=4 }
      { \int_compare_p:nNn 1=\ERROR } % is skipped
    }
   && \int_compare_p:nNn 2=2
  }
 \bool_if:NTF \l_tmpa_bool \TRUE \FALSE
}
```

This test will produce the following in the output.

```
=========================================
TEST 8: bool_set, lazy evaluation
=========================================
TRUE
=========================================
```

(Only if it's the eighth test in the file of course, and assuming expl3 coding conventions are active.)

2.3 Showing box content

The commands introduced above are only useful for checking algorithmic or logical correctness. Many packages should be tested based on their typeset output instead; TEX provides a mechanism for this by printing the contents of a box to the log file. The `regression-test.tex` driver file sets up the relevant TEX parameters to produce as much output as possible when showing box output.

A plain TEX example of showing box content follows.

```
\input regression-test.tex\relax
\START
\setbox0=\hbox{\rm hello \it world $a=b+c$}
\showbox0
\END
```

This produces the output shown in Figure 6 (left side). It is clear that if the definitions used to typeset the material in the box changes, the log output will differ and the test will no longer pass.

The equivalent test in LATEX 2_ε using expl3 is similar.

```
\input{regression-test.tex}
\documentclass{article}
\usepackage{expl3}
\START
\ExplSyntaxOn
\box_new:N \l_tmp_box
\hbox_set:Nn \l_tmp_box {hello~ \emph{world}~ $a=b+c$}
\box_show:N \l_tmp_box
\ExplSyntaxOff
\END
```

```
> \box0=
\hbox(6.94444+0.83333)x90.56589
.\tenrm h
.\tenrm e
.\tenrm l
.\tenrm l
.\tenrm o
.\glue 3.33333 plus 1.66666 minus 1.11111
.\tenit w
.\tenit o
.\tenit r
.\tenit l
.\tenit d

.\glue 3.57774 plus 1.53333 minus 1.0222
.\mathon
.\teni a
.\glue(\thickmuskip) 2.77771 plus 2.77771
.\tenrm =
.\glue(\thickmuskip) 2.77771 plus 2.77771
.\teni b
.\glue(\medmuskip) 2.22217 plus 1.11108 minus 2.22217
.\tenrm +
.\glue(\medmuskip) 2.22217 plus 1.11108 minus 2.22217
.\teni c
.\mathoff

! OK.
l.9 \showbox0
```

```
> \box71=
\hbox(6.94444+0.83333)x91.35481
.\OT1/cmr/m/n/10 h
.\OT1/cmr/m/n/10 e
.\OT1/cmr/m/n/10 l
.\OT1/cmr/m/n/10 l
.\OT1/cmr/m/n/10 o
.\glue 3.33333 plus 1.66666 minus 1.11111
.\OT1/cmr/m/it/10 w
.\OT1/cmr/m/it/10 o
.\OT1/cmr/m/it/10 r
.\OT1/cmr/m/it/10 l
.\OT1/cmr/m/it/10 d
.\kern 1.03334
.\glue 3.33333 plus 1.66666 minus 1.11111
.\mathon
.\OML/cmm/m/it/10 a
.\glue(\thickmuskip) 2.77771 plus 2.77771
.\OT1/cmr/m/n/10 =
.\glue(\thickmuskip) 2.77771 plus 2.77771
.\OML/cmm/m/it/10 b
.\glue(\medmuskip) 2.22217 plus 1.11108 minus 2.22217
.\OT1/cmr/m/n/10 +
.\glue(\medmuskip) 2.22217 plus 1.11108 minus 2.22217
.\OML/cmm/m/it/10 c
.\mathoff

! OK.
<argument> \l_tmp_box

l.12 \box_show:N \l_tmp_box
```

Figure 6: Output from displaying the contents of a simple box to the log file, using plain TEX (left) and expl3 (right). Some blank lines have been added to the plain TEX version to help with the comparison.

The output from this test is shown in Figure 6 (right side). There is marginal difference (mostly related to font selection and different logging settings in LATEX) between the plain and expl3 versions.

When examples are not self-contained enough to be typeset into boxes, it is possible to ask TEX to output the entire contents of a page. Insert `\showoutput` for LATEX or set `\tracingoutput` positive for plain TEX; ensure that the test ends with `\newpage` or equivalent because TEX waits until the entire page is finished before outputting it.

TODO: should we add something like `\TRACEPAGES` to be format-agnostic here? Should this perhaps even be active by default?

2.4 Testing entire pages

There may be occasions where creating entire test pages is necessary to observe the test output required. That is best achieved by applying `\showoutput` and forcing a complete page to be produced, for example

```
function runtest_tasks(name)
  return "biber " .. name
end
```

Figure 7: Example `runtest_tasks` function.

```
\input{regression-test.tex}
\documentclass{article}
\usepackage{expl3}
\START
\showoutput
% Test content here
\vfil\break
\END
```

2.5 Additional test tasks

A standard test will run the file ⟨*name*⟩`.lvt` using one or more engines, but will not carry out any additional processing. For some tests, for example bibliography generation, it may be desirable to call one or more tools in addition to the engine. This can be arranged by defining `runtest_tasks`, a function taking one argument, the name of the current test (this is equivalent to TEX's `\jobname`, *i.e.* it lacks an extension). The function `runtest_tasks` is is into a call to the system to run the engine. As such, it should take return a string with the appropriate command(s) and option(s). If more than one task is required, these should be separated by use of `os_concat`, a string variable defined by l3build as the correct concatenation marker for the system. An example of `runtest_tasks` suitable for calling Biber is shown in Listing 7.

2.6 Epoch setting

To produce predictable output when using dates, the test system offers the ability to set the epoch to a known value. The `1463734800` variable may be given as a raw value (a simple integer) or as a date in ISO format. The two flags `"true"` and `"false"` then determine whether this is applied in testing and typesetting, respectively.

The epoch may also be given as a command line option, `-E`, which again takes either a date or raw epoch. When given, this will automatically activate forcing of the epoch in both testing and typesetting.

3 Alternative test formats

3.1 Generating test files with DocStrip

It is possible to pack tests inside source files. Tests generated during the unpacking process will be available to the `check` and `save` commands as if they were stored in the `testfiledir`. Any explicit test files inside `testfiledir` take priority over generated ones with the same names.

```
\input regression-test.tex\relax
\START
\TEST{counter-math}{
%<*test>
  \OMIT
  \newcounter{numbers}
  \setcounter{numbers}{2}
  \addtocounter{numbers}{2}
  \stepcounter{numbers}
  \TIMO
  \typeout{\arabic{numbers}}
%</test>
%<expect>  \typeout{5}
}
\END
```

Figure 8: Test and expectation can be specified side-by-side in a single `.dtx` file.

```
 \generate{\file{\jobname.lvt}{\from{\jobname.dtx}{test}}
           \file{\jobname.lve}{\from{\jobname.dtx}{expect}}}
```

Figure 9: Test and expectation are generated from a `.dtx` file of the same name.

3.2 Specifying expectations

Regression tests check whether changes introduced in the code modify the test output. Especially while developing a complex package there is not yet a baseline to save a test goal with. It might then be easier to formulate the expected effects and outputs of tests directly. To achieve this, you may create an `.lve` instead of a `.tlg` file.[1] It is processed exactly like the `.lvt` to generate the expected outcome. The test fails when both differ.

Combining both features enables contrasting the test with its expected outcome in a compact format. Listing 8 exemplary tests TEXs counters. Listing 9 shows the relevant part of an `.ins` file to generate it.

4 Release-focussed features

4.1 Automatic tagging

As detailed above, the `tag` target will automatically edit source files to modify date and release tag. As standard, no automatic replacement takes place, but setting up a `updae_tag()` function will allow this to happen. This takes four arguments:

- The file name
- The full content of the file
- The tag name
- The tag date

[1] Mnemonic: `lvt`: **t**est, `lve`: **e**xpectation

```
-- Detail how to set the version automatically
function update_tag(file,content,tagname,tagdate)
  if string.match(file, "%.dtx$") then
    return string.gsub(content,
      "\n%% \\date{Released %d%d%d%d/%d%d/%d%d}\n",
      "\n%% \\date{Released " .. tagname .. "}\n")
  elseif string.match(file, "%.md$") then
    return string.gsub(content,
      "\nRelease %d%d%d%d/%d%d/%d%d\n",
      "\nRelease " .. tagname .. "\n")
  elseif string.match(file, "%.lua$") then
    return string.gsub(content,
      '\nrelease_date = "%d%d%d%d/%d%d/%d%d"\n',
      '\nrelease_date = "' .. tagname .. '"\n')
  end
  return contents
end
```

Figure 10: Example `update_tag` function.

and should return the (modified) contents for writing to disk. For example, the function used by `l3build` itself is shown in Figure 10.

To allow more complex tasks to take place, a hook `tag_hook()` is also available. It will receive the tag name and date as arguments, and may be used to carry out arbitrary tasks during tagging (for example, setting a version control tag for an entire repository).

4.2 Typesetting documentation

As part of the overall build process, l3build will create PDF documentation as described earlier. The standard build process for PDFs will attempt to run Biber, BibTeX and MakeIndex as appropriate (the exact binaries used are defined by `"biber"`, `"bibtex8"` and `"makeindex"`). However, there is no attempt to create an entire PDF creation system in the style of `latexmk` or similar.

For package authors who have more complex requirements than those covered by the standard set up, the Lua script offers the possibility for customisation. The Lua function `typeset` may be defined before reading `l3build.lua` and should take one argument, the name of the file to be typeset. Within this function, the auxiliary Lua functions `biber`, `bibtex`, `makeindex` and `tex` can be used, along with custom code, to define a PDF typesetting pathway. The functions `biber` and `bibtex` take a single argument: the name of the file to work with *minus* any extension. The `tex` takes as an argument the full name of the file. The most complex function `makeindex` requires the name, input extension, output extension, log extension and style name. For example, Figure 11 shows a simple script which might apply to a case where multiple BibTeX runs are needed (perhaps where citations can appear within other references).

Where there are complex requirements for pre-compiled demonstration files, the hook `typeset_demo_hook()` is available: it runs after copying files to the typesetting location but before the main typesetting run. This may be used for example to script a very large number of demonstrations using a single source (see the beamer package for an example of this).

5 Lua interfaces

Whilst for the majority of users the simple variable-based control methods outlined above will suffice, for more advanced applications there will be a need to adjust behavior by using interfaces within the Lua code. This section details the global variables and functions provided.

```
#!/usr/bin/env texlua

-- Build script with custom PDF route

module = "mymodule"

function typeset(file)
  local name = jobname(file)
  local errorlevel = tex (file)
  if errorlevel == 0 then
    -- Return a non-zero errorlevel if anything goes wrong
    errorlevel =(
      bibtex(name) +
      tex(file)    +
      bibtex(name) +
      tex(file)    +
      tex(file)
    )
  end
  return errorlevel
end
```

Figure 11: A customised PDF creation script.

5.1 Global variables

`options` The `options` table holds the values passed to l3build at the command line. The possible entries in the table are given in the table below.

Entry	Type
`date`	String
`engine`	Table
`force`	Boolean
`halt`	Boolean
`help`	Boolean
`names`	Table
`pdf`	Boolean
`quiet`	Boolean
`rerun`	Boolean
`testfiledir`	Table
`version`	String

5.2 Utility functions

The utility functions are largely focussed on file operations, though a small number of others are provided. File paths should be given in Unix style (using `/` as a path separator). File operations take place relative to the path from which l3build is called. File operation syntax is largely modelled on Unix command line commands but reflect the need to work on Windows in a flexible way.

`abspath()` `abspath(⟨target⟩)`

Returns a string which gives the absolute location of the ⟨*target*⟩ directory.

`dirname()` `dirname(⟨file⟩)`

Returns a string comprising the path to a ⟨*file*⟩ with the name removed (*i.e.* up to the last `/`). Where the ⟨*file*⟩ has no path data, `"."` is returned.

`basename()` `basename(⟨file⟩)`

Returns a string comprising the full name of the ⟨*file*⟩ with the path removed (*i.e.* from the last `/` onward).

`cleandir()` `cleandir(⟨dir⟩)`

Removes any content within the ⟨*dir*⟩; returns an error level.

`cp()` `cp(⟨glob⟩, ⟨source⟩, ⟨destination⟩)`

Copies files matching the ⟨*glob*⟩ from the ⟨*source*⟩ directory to the ⟨*destination*⟩; returns an error level.

`direxists()` `direxists(⟨dir⟩)`

Tests if the ⟨*dir*⟩ exists; returns a boolean value.

`fileexists()` `fileexists(⟨file⟩)`

Tests if the ⟨*file*⟩ exists; returns a boolean value.

`filelist()` `filelist(⟨path⟩, [⟨glob⟩])`

Returns a table containing all of the files with the ⟨*path*⟩ which match the ⟨*glob*⟩; if the latter is absent returns a list of all files in the ⟨*path*⟩.

`jobname()` `jobname(⟨file⟩)`

Returns a string comprising the jobname of the file with the path and extension removed (*i.e.* from the last `/` up to the last `.`).

`mkdir()` `mkdir(⟨dir⟩)`

Creates the ⟨*dir*⟩; returns an error level.

`ren()` `ren(⟨dir⟩, ⟨source⟩, ⟨destination⟩)`

Renames the ⟨*source*⟩ file to the ⟨*destination*⟩ name within the ⟨*dir*⟩; returns an error level.

`rm()` `rm(⟨dir⟩, ⟨glob⟩)`

Removes files in the ⟨*dir*⟩ matching the ⟨*glob*⟩; returns an error level.

`run()` `run(⟨dir⟩, ⟨cmd⟩)`

Executes the ⟨*cmd*⟩, starting it in the ⟨*dir*⟩; returns an error level.

`splitpath()` `splitpath(⟨file⟩)`

Returns two strings split at the last `/`: the `dirname()` and the `basename()`.

`normalize_path()` `normalize_path(⟨path⟩)`

When called on Windows, returns a string comprising the ⟨*path*⟩ with `/` characters replaced by `\\`. In other cases returns the path unchanged.

5.3 System-dependent strings

To support creation of additional functionality, the following low-level strings are exposed by l3build: these all have system-dependent definitions and avoid the need to test `os.type` during the construction of system calls.

`os_concat` The concatenation operation for using multiple commands in one system call, *e.g.*

```
os.execute("tex " .. file .. os_concat .. "tex " .. file)
```

`os_null` The location to redirect commands which should produce no output at the terminal: almost always used preceded by `>`, *e.g.*

```
os.execute("tex " .. file .. " > " .. os_null)
```

`os_pathsep` The separator used when setting an environment variable to multiple paths, *e.g.*

```
os.execute(os_setenv .. " PATH=../a" .. os_pathsep .. "../b")
```

`os_setenv` The command to set an environmental variable, *e.g.*

```
os.execute(os_setenv .. " PATH=../a")
```

`os_yes` A command to generate a series of 200 lines each containing the character `y`: this is useful as the Unix `yes` command cannot be used inside `os.execute` (it does not terminate).

5.4 Components of `l3build`

`call()` `call(`⟨*dirs*⟩`, `⟨*target*⟩`, [`⟨*options*⟩`])`

Runs the `l3build` ⟨*target*⟩ (a string) for each directory in the ⟨*dirs*⟩ (a table). This will pass command line options for the parent script to the child processes. The ⟨*options*⟩ table should take the same form as the global ⟨*options*⟩, described above. If it is absent then the global list is used. Note that any entry for the `target` in this table is ignored.

5.5 Customising the manifest file

The default setup for the manifest file creating with the `manifest` target attempt to reflect the defaults for l3build itself. The groups (and hence the files) displayed can be completely customised by defining a new setup function which creates a Lua table with the appropriate settings (§5.5.1).

The formatting within the manifest file can be customised by redefining a number of Lua functions. This includes how the files are sorted within each group (§5.5.2), the inclusion of one-line descriptions for each file (§5.5.3), and the details of the formatting of each entry (§5.5.4).

To perform such customisations, either include the re-definitions directly within your package's `build.lua` file, or make a copy of `l3build-manifest-setup.lua`, rename it, and load it within your `build.lua` using `dofile()`.

5.5.1 Custom manifest groups

The setup code for defining each group of files within the manifest looks something like the following:

```
manifest_setup = function()
  local groups = {
    {
       subheading = "Repository files",
```

Table 2: Table entries used in the manifest setup table for a group.

Entry	Description
`name`	The heading of the group
`description`	The description printed below the heading
`files`	Files to include in this group
`exclude`	Files to exclude (default `{excludefiles}`)
`dir`	The directory to search (default `maindir`)
`rename`	An array with a `gsub` redefinition for the filename
`skipfiledescription`	Whether to extract file descriptions from these files (default `false`)

Table 3: Table entries used in the manifest setup table for a subheading.

Entry	Description
`subheading`	The subheading
`description`	The description printed below the subheading

```
      description = [[
Files located in the package development repository.
      ]],
    },
    {
      name    = "Source files",
      description = [[
These are source files generating the package files.
      ]],
      files   = {sourcefiles},
    },
    {
      name    = "Typeset documentation source files",
      description = [[
These files are typeset using LaTeX to produce the PDF documentation for the package.
      ]],
      files   = {typesetfiles,typesetsourcefiles,typesetdemofiles},
    },
    ...
  }
  return groups
end
```

The `groups` variable is an ordered array of tables which contain the metadata about each 'group' in the manifest listing. The keys supported in these tables are outlined in Table 2 and Table 3 See the complete setup code in `l3build-manifest-setup.lua` for examples of these in use.

5.5.2 Sorting within each manifest group

Within a single group in the manifest listing, files can be matched against multiple variables. For example, for `sourcefiles={*.dtx,*.ins}` the following (unsorted) file listing might result:

- foo.dtx
- bar.dtx
- foo.ins
- bar.ins

This listing can be sorted using two separate functions by the default manifest code. The first, default, is to sort alphabetically within a single variable match. This keeps all files of a single extension contiguous in the listing. To edit how this sort is performed, redefine the `manifest_sort_within_match` function.

The second approach to sorting is to apply a sorting function to the entire set of matched files. (This happens *after* any sorting is applied for each match.) By default this is a no-op but can be edited by redefining the `manifest_sort_within_group` function. For example:

```
manifest_sort_within_group = function(files)
  local f = files
  table.sort(f)
  return f
end
```

This will produce an alphabetical listing of files:

- bar.dtx
- bar.ins
- foo.dtx
- foo.ins

5.5.3 File descriptions

By default the manifest contains lists of files, and with a small addition these lists can be augmented with a one-line summary of each file. If the Lua function `manifest_extract_filedesc` is defined, it will be used to search the contents of each file to extract a description for that file. For example, perhaps you are using multiple `.dtx` files for a project and the argument to the first `\section` in each can be used as a file description:

```
manifest_extract_filedesc = function(filehandle,filename)

  local all_file = filehandle:read("*all")
  local matchstr = "\\section{(.-)}"

  filedesc = string.match(all_file,matchstr)

  return filedesc
end
```

(Note the `matchstr` above is only an example and doesn't handle nested braces.)

5.5.4 Custom formatting

After the manifest code has built a complete listing of files to print, a series of file writing operations are performed which create the manifest file. The following functions can be re-defined to change the formatting of the manifest file:

- `manifest_write_opening`: Write the heading of the manifest file and its opening paragraph.
- `manifest_write_subheading`: Write a subheading and description
- `manifest_write_group_heading`: Write the section heading of the manifest group and the group description
- `manifest_write_group_file`: Write the filename (when not writing file descriptions)
- `manifest_write_group_file_descr`: Write the filename and the file description

Full descriptions of their usage and arguments can be found within the `l3build-manifest-setup.lua` code itself.

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.