The l3build package
Checking and building packages

The LaTeX Project*

Released 2023-03-22

Contents

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

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. Wrapper functions/binaries are distributed in the standard TEX distributions so that the script can be called using `l3build` on the command line; run without arguments it prints a brief synopsis of its usage.

The l3build system is designed for packages 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.)

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 LATEX 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_ε 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 safely deleted; all material within is re-generated for each command of the l3build system.

1.2 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* additional code to customize the build process.

The example scripts given in Section 1.4 largely cover the required knowledge in Lua programing. For a more advanced usage, one may consult general Lua documentations including http://www.lua.org/manual/5.3/manual.html and for the few `texlua` specific additions see section 4.2 of the LuaTEX manual available locally with `texdoc luatex` command line or at https://www.pragma-ade.com/general/manuals/luatex.pdf.

1.3 Main build targets

In the working directory of a bundle or module, l3build is run by executing

`l3build` ⟨*target*⟩ `[`⟨*option(s)*⟩`]`

where ⟨*target*⟩ can be one of the following:

- `check`
- `check` ⟨*name(s)*⟩
- `clean`
- `ctan`
- `doc`
- `doc` ⟨*name(s)*⟩
- `install`
- `manifest`
- `save` ⟨*name(s)*⟩
- `tag [`⟨*tag name*⟩`]`
- `uninstall`
- `unpack`

- `upload [⟨version⟩]`

These targets are described below.

As well as these targets, the system recognises the options

- `--config` (`-c`) Configuration(s) to use for testing
- `--date` Date to use when tagging data
- `--debug` Runs the target in debug mode (not supported by all targets)
- `--dirty` Skip cleaning up of the test area
- `--dry-run` Runs the `install` target but does not copy any files: simply lists those that would be installed
- `--email` Sets the email address for CTAN upload
- `--engine` (`-e`) Sets the engine to use for testing
- `--epoch` Sets the epoch for typesetting and testing
- `--file` (`-F`) Take the upload announcement from the given file
- `--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 `checkengines`
- `--full` Instructs the `install` target to include the `doc` and `source` trees
- `--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
- `--message` (`-m`) Text for upload announcement
- `--quiet` (`-q`) Suppresses output from unpacking
- `--rerun` Run tests without unpacking/set up
- `--show-log-on-error` To be used in addition to `-halt-on-error` and results in the full `.log` file of a failed test to be shown on the console
- `--show-saves` (`-S`) When tests fail, print the `l3build save` commands needed to regenerate the tests assuming that the failures were false negatives.
- `--shuffle` Shuffle the order in which tests run
- `--texmfhome` Sets the location of the user tree for installing

`$ l3build 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`.

The `texmfdir` is also made available to the tests (if defined and non-empty). 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 `tex` will run tests using *e.g.* `pdftex`. (Currently, this should be one of `latex` or `tex`.) 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`.

The `texmfdir` variable sets a directory which is made available for recursive searching *in addition* to any files copied from `supportdir`. No subdivison of `texmfdir` is attempted, thus it should not contain multiple files with the same name. The `texmfdir` is made available both to checking and typesetting.

`$ l3build 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.

```
$ l3build 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`).

```
$ l3build 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`.

The special case `ctanreadme` is used to allow renaming of a local `foo.xyz` file to `README.xyz`. The local `foo.xyz` should be listed in `textfiles`, and will be renamed as part of constructing the CTAN structure. The file extension will be unchanged by this process.

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).

The intermediate build directories `ctandir` and `tdsdir` are used to construct the archive.

```
$ l3build 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/doc` directory (`typesetdir`) 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.

```
$ l3build doc ⟨name(s)⟩
```

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

```
$ l3build 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` switch: the standard setting is the location set as `TEXMFHOME`.

```
$ l3build 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 1.12.

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.

`$ l3build 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 5.7.

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.

`$ l3build tag [`⟨*tag name*⟩`]`

Apply the Lua `update_tag()` function to modify the contents of files specified by `tagfiles` to update the 'release tag' (or package version) and date. The tag is given as the optional command line argument ⟨*tag name*⟩ and the date using `--date` (or `-d`). If not given, the date will default to the current date in ISO format (YYYY-MM-DD). If no ⟨*tag name*⟩ is given, the tag will default to `nil`. Both are passed as arguments to the `update_tag()` function.

The standard setup does nothing unless tag update is set up by defining a custom `update_tag()` function. See Section 4.2 for full details on this feature.

`$ l3build 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`.

```
$ l3build upload [⟨version⟩]
```

This target uses `curl` to upload the package zip file (created using `ctan`) to CTAN. To control the metadata used to upload the package, the `uploadconfig` table should be populated with a number of fields. These are documented in Table 2. Missing required fields will result in an interactive prompt for manual entry. When given, ⟨*version*⟩ overrides `uploadconfig.version`.

See Section 4.6 for full details on this feature.

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 LaTeX 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.

A collection of full examples (source files in various layouts) are available at https://github.com/latex3/l3build/tree/master/examples.

1.5 Variables

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

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

```
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.

Variable	Default	Description
`maindir`	`"."`	Top level directory for the module/bundle
`docfiledir`	`"."`	Directory containing documentation files
`sourcefiledir`	`"."`	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
`texmfdir`	`maindir .. "/texmf"`	Directory containing support files in tree form
`textfiledir`	`"."`	Directory containing plain text 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
`resultdir`	`builddir .. "/result"`	Directory for PDF files when using PDF-based tests
`testdir`	`builddir .. "/test"`	Directory for running tests
`typesetdir`	`builddir .. "/doc"`	Directory for building documentation
`unpackdir`	`builddir .. "/unpacked"`	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
`dynamicfiles`	`{ }`	Secondary files to cleared before each test is run
`excludefiles`	`{"*~"}`	Files to ignore entirely (default for Emacs backup files)
`installfiles`	`{"*.sty","*.cls"}`	Files to install to the `tex` 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"

Variable	Default	Description
`includetests`	`{"*"}`	Test names to include when checking
`excludetests`	`{}`	Test names to exclude when checking
`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
`specialformats`	⟨*table*⟩	Non-standard engine/format combinations
`test_types`	⟨*table*⟩	Custom test variants
`test_order`	`{"log", "pdf"}`	Which kinds of tests to evaluate
`checkconfigs`	`{}`	Configurations to use for tests
`typesetexe`	`"pdflatex"`	Executable for compiling `doc(s)`
`unpackexe`	`"pdftex"`	Executable for running `unpack`
`biberexe`	`"biber"`	Biber executable
`bibtexexe`	`"bibtex8"`	BibTeX executable
`makeindexexe`	`"makeindex"`	MakeIndex executable
`curlexe`	`"curl"`	Curl executable for `upload`
`checkopts`	`"-interaction=nonstopmode"`	Options passed to engine when running checks
`typesetopts`	`"-interaction=nonstopmode"`	Options passed to engine when typesetting
`unpackopts`	`""`	Options passed to engine when unpacking
`biberopts`	`""`	Biber options
`bibtexopts`	`"-W"`	BibTeX options
`makeindexopts`	`""`	MakeIndex options
`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
`specialtypesetting`	⟨*table*⟩	Non-standard typesetting combinations
`forcecheckepoch`	`"true"`	Force epoch when running tests
`forcedocepoch`	`"false"`	Force epoch when typesetting
`asciiengines`	`{"pdftex"}`	Engines which should log as pure ASCII
`checkruns`	`1`	Number of runs to complete for a test before comparing the log
`forcecheckruns`	`false`	Always run `checkruns` runs and never stop early
`ctanreadme`	`"README.md"`	Name of the file to send to CTAN as `README.`⟨*ext*⟩
`ctanzip`	`ctanpkg ... "-ctan"`	Name of the zip file (without extension) created for upload to CTAN
`epoch`	`1463734800`	Epoch (Unix date) to set for test runs
`flatten`	`true`	Switch to flatten any source structure when sending to CTAN
`flattentds`	`true`	Switch to flatten any source structure when creating a TDS structure

Variable	Default	Description
`maxprintline`	`9999`	Length of line to use in log files
`packtdszip`	`false`	Switch to build a TDS-style zip file for CTAN
`ps2pdfopts`	`""`	Options for `ps2pdf`
`typesetcmds`	`""`	Instructions to be passed to TEX when doing typesetting
`typesetruns`	`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
`tdslocations`	`{ }`	Map for non-standard file installations
`tdsdirs`	`{ }`	List of ready-to-use source locations
`uploadconfig`	⟨*table*⟩	Metadata to describe the package for CTAN (see Table 2)
`uploadconfig.pkg`	`ctanpkg`	Name of the CTAN package
`bakext`	`".bak"`	Extension of backup files
`dviext`	`".dvi"`	Extension of DVI files
`lvtext`	`".lvt"`	Extension of log-based test files
`tlgext`	`".tlg"`	Extension of test file output
`tpfext`	`".tpf"`	Extension of PDF-based test output
`lveext`	`".lve"`	Extension of auto-generating test file output
`logext`	`".log"`	Extension of checking output, before processing it into a `.tlg`
`pvtext`	`".pvt"`	Extension of PDF-based test files
`pdfext`	`".pdf"`	Extension of PDF file for checking and saving
`psext`	`".ps"`	Extension of PostScript files

1.6 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.

1.7 Selective running of tests

The variables `includetests` and `excludetests` may be used to select which tests are run: these variables take raw test *names* not full file names. The list of tests in `excludetests` overrides any matches in `includetests`, meaning that tests can be disabled selectively. It also makes it possible to disable test on for example a platform basis: the `texlua` specific variable `os.type` may be used to set `excludetests` only on some systems.

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 4. 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 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 LaTeX, 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 LaTeX 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 LaTeX 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"`.

```
-- Build script for LaTeX "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 LaTeX "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.

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

Figure 4: Example of using additional (or overriding) settings for configuring tests in a different subdirectory.

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`, `docfiledir` and `textfiledir` 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. The `flattentds` setting controls the same concept for TDS creation.

Notice that text files are treated separately from documentation files when splitting trees: this is to allow for the common case where files such as `README` and `LICENSE` are at the top level even when other documentation files are in a sub-directory.

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

For more complex layouts in which sources are laid out in TDS format and should be used directly, the table `tdsdirs` is available. Each entry is a source directory and the matching installation target, for example

```
tdsdirs = {sources = "tex"}
```

This would enable a directory `sources` in the development area to be used for testing and typesetting, and for it to be installed into the `tex` tree when building a release. When this method is used, the sources are *not* copied into the local tree: like `texmfdir`, they are added directly to the areas accessible during a testing or typesetting run. When using this approach, the files listed in `typesetfiles` *must* still be included in `docfiles`: they have to be directly visible to l3build, not found by `kpsewhich` searching.

1.11 Non-standard formats/binaries

The standard approach used by l3build is to use a combination of `engine` and `checkformat` to generate the *binary* and *format* combination used for tests. For example, when `pdftex` is the `engine` and `latex` is the `checkformat`, the system call used is

```
pdftex --fmt=pdflatex
```

i.e. the binary names is the same as the `engine`, and the format is a simple substitution of the `checkformat` into `engine`, replacing `tex`.

For more complex set ups, `specialformats` should be used. This is a table with one entry per `checkformat`. Each entry is itself a table, and these contain a list of engines and settings for `binary`, `format` and `options`. For example, for ConTEXt and appropriate set up is

```
specialformats.context = {
  luatex = {binary = "context", format = ""},
  pdftex = {binary = "texexec", format = ""},
  xetex  = {binary = "texexec", format = "", options = "--xetex"}
}
```

Additional tokens can also be injected before the loading of a test file using the `tokens` entry: this might for example be used to select a graphics driver with a DVI-based route.

1.12 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 related to loading `.fd` files (from `(`⟨*name*⟩`.fd` to the matching `)`).
- 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 additional 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 file dates to `....-..-..`, and any version numbers on the same lines to `v....`
- Conversion of register numbers in assignment lines \⟨*register*⟩=\⟨*type*⟩⟨*number*⟩ to \⟨*type*⟩⟨`...`⟩
- Conversion of box numbers in `\show` lines `> \box`⟨*number*⟩`=` to `> \box...=`
- Conversion of Lua data reference ids `<lua data reference` ⟨*number*⟩`>` to `<lua data reference ...>`
- Removal of some (u)pTEX data where it is equivalent to pdfTEX (`yoko direction`, `\displace 0.0`)
- Removal of various `\special` lines inserted due to the build process

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.

1.13 Breaking changes

Very occasionally, it is necessary to make changes to l3build that change the `.tlg` file results. This is typically when additional normalisation is required. When this is the case, you should first verify that `.tlg` files pass with the older l3build, then update only l3build, re-check the files and save the results. Where possible, we provide a mechanism to run with older setting to allow this process to take place smoothly.

1.13.1 Release 2023-03-22

This release changes the standard value of `maxprintline` for 79 to 9999, to suppress line wrapping in the log. This makes normalisation of for example file paths more reliable. To check that `.tlg` files are correct, you can set `maxprintline` in your `build.lua` file explicitly to the old default, check that tests pass, then remove this line and re-check.

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 `\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

`\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 `\SEPARATOR`s 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.
- `\BEGINTEST{`⟨*title*⟩`}` ... `\ENDTEST` is an environment form of `\TEST`, allowing verbatim material, *etc.* to appear.
- `\SHOWFILE` (ε-TeX only) Shows the content of the file given as an argument.
- `\ASSERT` and `\ASSERTSTR` Asserts if the full expansion of the two required arguments are the same: the `\ASSERT` function is token-based, the `\ASSERTSTR` works on a string basis.

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 5 (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 5: 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 5 (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,run)
  if run == 1 then
    return "biber␣" .. name
  else
    return ""
  end
end
```

Figure 6: Example `runtest_tasks` function.

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

2.5 Pre-check hook

To allow complex set up for tests, a hook `checkinit_hook()` is available to be executed once all standard set up is complete but before any tests are run. This should return an integer value: 0 indicates no error.

2.6 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 two arguments, the name of the current test (this is equivalent to TEX's `\jobname`, *i.e.* it lacks an extension) and the current run number. The function `runtest_tasks` is run after the main call to the engine for a test cycle. It should return an errorlevel value. 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 6.

2.7 Instructions for rebuilding test output

Sometimes changes to fundamental parts of the code can cause a lot of tests to fail even though the actually tested systems are still working correctly. This is especially common when the logging and error reporting systems changes and therefore all log file based tests using the component fail with these changes.

In these cases, the option `--show-saves` can be passed to `l3build check` in order to generate a list of `l3build save` commands which can be executed to regenerate the expected output of all tests which fail. Additionally it sometimes prints a list of `l3build check` commands for tests which might still fail due to engine differences after

running the `save` commands. After running all these `l3build check` commands and all `l3build save` commands listed by them, all tests will succeed.

When bundles are used `l3build check --show-saves` has to be executed separately for every module in the bundle.

This option is potentially dangerous and therefore should only be used with care. It can easily hide valid test failures between a bunch of spurious changes. Therefore you should always take a close look at the difference files generated by `l3build check` before running the generated `l3build save` commands. Additionally it should only be used when you are aware of the reason why a large number of tests failed and the change causing the failures has been tested separately to have no unintended side effects.

2.8 Epoch setting

To produce predictable output when using dates, the test system offers the ability to set the epoch to a known value. The `epoch` variable may be given as a raw value (a simple integer) or as a date in ISO format. The two flags `forcecheckepoch` and `forcedocepoch` 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.

2.9 Settings in `texmf.cnf`

To allow application of non-standard TEX trees or similar non-standard settings, l3build enables searching for a `texmf.cnf` file by setting the envirnmental variable `TEXMFCNF`. This might for example be used with a file containing

```
TEXMFAUXTREES = ../../texmf,
```

for adding a local tree within the development repository (assuming the typical l3build layout).

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.

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.

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

```
\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 7: 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 8: Test and expectation are generated from a `.dtx` file of the same name.

Combining both features enables contrasting the test with its expected outcome in a compact format. Listing 7 exemplary tests TEXs counters. Listing 8 shows the relevant part of an `.ins` file to generate it.

3.3 PDF-based tests

In most cases, testing is best handled by using the text-based methods outlined above. However, there are cases where the detail of output structure is important. This can only be fully tested by comparing PDF structure. To support this, l3build can be instructed to build and compare PDF files by setting up tests in `.pvt` files. The following normalization takes place:

- Replacement of binary streams by the marker `[BINARY STREAM]`
- Replacement of `/ID` values by `ID-STRING`
- Removal of blank lines
- Removal of comment (`%%`) lines

After this normalization takes place, the file can not usually be rendered properly. To check if the build system has produced a correct PDF, the pre-normalization PDF can be found in the `build` folder.

To allow platform-independence, PDF-based tests must use only Type 1 or OpenType fonts: Type3 fonts are system-dependent. PDF files are engine-specific, thus one `.tpf` file should be stored per engine to be tested.

3.4 Custom tests

If neither the text-based methods nor PDF-based tests are sufficient, there is the additional option of defining custom variants with individual normalization rules.

For this, the variant has to be registered in the `test_types` table and then activated in `test_order`.

Every element in `test_types` is a table with fields `test` (the extension of the test file), `reference` (the extension of the file the output is compared with), `generated` (extension of the analyzed LaTeX output file) and `rewrite` (A Lua function for normalizing the output file, taking as parameters the name of the unnormalized LaTeX output file to be read, the name of the normalized file to be written, the engine name and a potential errorcode).

For example:

```
test_types = {
  mytest = {
    test = ".mylvt",
    reference = ".mytlg",
    generated = ".log",
    rewrite = function(source, normalized, engine, errorcode)
      -- In this example we just copy the logfile without any normalization
      os.execute(string.format("cp %s %s", source, normalized)
    end,
  },
}
test_order = {"mylvt", "log", "pdf"}
```

4 Release-focussed features

4.1 Installation structure

With the standard settings, l3build will install files within the TeX directory structure (TDS) as follows

- `installfiles` within a ⟨*bundle*⟩/⟨*module*⟩ (or ⟨*module*⟩) directory inside `tex/`⟨*format*⟩
- `sourcefiles` within a ⟨*bundle*⟩/⟨*module*⟩ (or ⟨*module*⟩) directory inside `source/`⟨*format*⟩
- Typeset PDFs within a ⟨*bundle*⟩/⟨*module*⟩ (or ⟨*module*⟩) directory inside `doc/`⟨*format*⟩
- `bstfiles` within a ⟨*bundle*⟩/⟨*module*⟩ (or ⟨*module*⟩) directory inside `bibtex/bst`
- `bibfiles` within a ⟨*bundle*⟩/⟨*module*⟩ (or ⟨*module*⟩) directory inside `bibtex/bib`
- `makeindexfiles` within a ⟨*bundle*⟩/⟨*module*⟩ (or ⟨*module*⟩) directory inside `makeindex`

For more complex set ups, this can be customised using the `tdslocations` table. Each entry there should be a glob specifying the TDS position of a file or files. Any files not specified in the table will use the standard locations above. For example, to place some files in the generic tree, some in the plain TeX tree and some in the LaTeX tree, one might use the set up shown in Figure 9.

The table is read in order, and thus specific file names should come before potential wild-card matches.

```
tdslocations =
  {
    "tex/generic/mypkg/*.generic.tex" ,
    "tex/plain/mypkg/*.plain.tex"     ,
    "tex/latex/mypkg/*.latex.tex"
  }
```

Figure 9: Example `tdslocations` table.

```
-- 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 content
end
```

Figure 10: Example `update_tag` function.

4.2 Automatic tagging

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

1. file name
2. full content of the file
3. tag name
4. tag date

The `update_tag()` function 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 after all files have been updated. For example, this can be used to set a version control tag for an entire repository.

4.3 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 `biberexe`, `bibtexexe`

and `makeindexexe`). 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_tasks()` 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). Note that this hook is intended for use files *not* listed in `typesetfiles` or `typesetdemofiles`.

4.4 Pre-typesetting hook

To allow complex set up for typesetting, a hook `docinit_hook()` is available to be executed once all standard set up is complete but before any typesetting is run.

```
#!/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.

4.5 Non-standard typesetting

To allow non-standard typesetting combinations, for example per-file choice of engines, the table `specialtypesetting` may be used. This is a table with one entry per file. Each entry is itself a table, and these contain a list of engines and settings for `cmd` and `func`. For example, to choose to use LuaTeX for one file when `typesetexe` is `pdftex`

```
specialtypesetting = specialtypesetting or {}
specialtypesetting["foo.tex"] = {cmd = "luatex -interaction=nonstopmode"}
```

or to select an entirely different typesetting function

```
specialtypesetting = specialtypesetting or {}
specialtypesetting["foo.tex"] = {func = typeset_foo}
```

4.6 Automated upload to CTAN

The CTAN upload process is backed by an API, which l3build can use to send zip files for release. Along with the file, a variety of metadata must be specified about the package, including the version, license, and so on, explained at https://www.ctan.org/upload. A description of this metadata is outlined in Table 2, and a simple example of an extract from a `build.lua` file using this is shown in Figure 12.

Note that the `upload` target will *not* execute the `ctan` target first.

This upload facility assumes availablity of `curl` on your system. In the case of Windows, the system curl will not be available if you are using a 32 bit TeX implementation. Curl executables are available for a variety of operating systems from https://curl.haxx.se/download.html.

Announcement text It can be convenient not to include the announcement text within the `build.lua` file directly. The command line argument `--message` (`-m`) allows the announcement to be included as part of the `l3build` arguments, and `--file` (`-F`) reads the announcement from a specified file. The `build.lua` file may also specify that this text is to be taken from the file specified by `uploadconfig.announcement_file`, this allows the release-specific announcement to be specified outside the main `build.lua` file. If `uploadconfig.announcement_file` is `nil` or specifies a file that can not be read, and no announcement is provided by the `announcement` field or commandline arguments, `l3build` will interactively prompt for text (which may be empty).

Note that if the announcement text is empty a 'silent update' is performed; this should usually be performed for minor bug or documentation fixes only.

Note text This optional field is for passing notes to the CTAN maintainers. As for announcements, the text may be set in `uploadconfig.note` or perhaps more usefully, if `uploadconfig.note_file` is the filename of a readable file the file text is used as the note.

Uploader details The CTAN team use the uploader email address as a form of low-security sanity check that the upload is coming from a reputable source. Therefore, it is advisable not to store this information within a public `build.lua` file. It can be set on the command line with the `--email` option to `l3build`; alternatively, a private configuration file could be used to add this information at upload time.

The `update` field In most scenarios the `update` field does not need to be explicitly set. By default l3build assumes that the package being uploaded already exists on CTAN (`update=true`). If it does not, this is caught in the validation process before uploading and automatically corrected. If you set `update` explicitly this will be passed directly to CTAN in all circumstances, leading to errors if you attempt to update a non-existing package or if you attempt to upload a new package with the same name as a pre-existing one.

The `curl` options file The l3build upload options are passed to `curl` by writing the fields to a text file with a default name being ⟨*package*⟩`-ctan.curlopt`. This is then passed to curl using its `--config` commandline option. (Using an intermediate file helps keep l3build portable between systems using different commandline quoting conventions. Any backslashes are doubled when writing to this file, so they do not need to be doubled in announcement and note texts.)

By default the file is written into the current directory alongside the zip file to be uploaded. You may wish to specify that this file is ignored by any version control in that directory (using `.gitignore` or similar). Or alternatively you can use the `uploadconfig.curlopt_file` field in the `build.lua` file to specify an alternative name or location for this file.

Validating To validate your upload but not actually submit to CTAN, you may use the `--dry-run` command-line option.

Debugging If you have have difficulty with the upload process, add the option `--debug` to divert the request from CTAN to a service that redirects the input back again so it can be examined. It can also be useful to check the contents of the `curlopts` file which has a record of the options passed to curl.

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.

Table 2: Fields used in the `uploadconfig` setup table. The first section of fields are *required* and if they are omitted the user will be interactively prompted for further input. Most commands take string input, but those that are indicated with 'Multi' accept more than one entry using an array of strings. Most of the fields correspond directly to the fields in the CTAN upload API, the last group relate to file use by l3build.

Field	Req.	Multi	Description
`announcement`	•		Announcement text
`author`	•		Author name (semicolon-separated for multiple)
`ctanPath`	•		CTAN path
`email`	•		Email address of uploader
`license`	•	•	Package license(s)[a]
`pkg`	•		Package name
`summary`	•		One-line summary
`uploader`	•		Name of uploader
`version`	•		Package version
`bugtracker`		•	URL(s) of bug tracker
`description`			Short description/abstract
`development`		•	URL(s) of development channels
`home`		•	URL(s) of home page
`note`			Internal note to CTAN
`repository`		•	URL(s) of source repositories
`support`		•	URL(s) of support channels
`topic`		•	Topic(s)[b]
`update`			Boolean `true` for an update, `false` for a new package
`announcement_file`			Announcement text file
`note_file`			Note text file
`curlopt_file`			The filename containing the options passed to curl

[a] See https://ctan.org/license
[b] See https://ctan.org/topics/highscore

```
uploadconfig = {
  pkg         = "vertbars",
  version     = "v1.0c",
  author      = "Peter R Wilson; Will Robertson",
  license     = "lppl1.3c",
  summary     = "Mark vertical rules in margin of text",
  ctanPath    = "/macros/latex/contrib/vertbars",
  repository  = "https://github.com/wspr/herries-press/",
  update      = true,
}
```

Figure 12: Example of `uploadconfig` from the vertbars package.

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
`config`	Table
`date`	String
`dirty`	Boolean
`dry-run`	Boolean
`email`	String
`engine`	Table
`epoch`	String
`file`	string
`first`	Boolean
`force`	Boolean
`full`	Boolean
`halt-on-error`	Boolean
`help`	Boolean
`message`	string
`names`	Table
`quiet`	Boolean
`rerun`	Boolean
`shuffle`	Boolean
`texmfhome`	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 and is readable; 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*⟩.

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

Like `filelist()` but returning a sorted table.

`glob_to_pattern()` `glob_to_pattern(⟨glob⟩)`

Returns the ⟨*glob*⟩ converted to a Lua pattern.

`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` **DEPRECATED** A command to generate a series of 300 lines each containing the character `y`: this is useful as the Unix `yes` command cannot be used inside `os.execute` (it does not terminate).

Rather than use this function, we recommend the replacement construct

```
io.popen(<cmd>,"w"):write(string.rep("y\n", 300)):close()
```

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.

`install_files()` `install_files(`⟨*target*⟩`,`⟨*full*⟩`,`⟨*dry-run*⟩`)`

Installs the files from the module into the TDS root ⟨*target*⟩. If ⟨*full*⟩ is `true`, all files are copied: if it is `false`, the `doc` and `source` trees are skipped. If ⟨*dry-run*⟩ is `true`, no files are copied, but instead the files which would be copied are reported.

5.5 Typesetting functions

All typesetting functions return 0 on a successful completion.

`biber()` `biber(⟨name⟩,⟨dir⟩)`

Runs Biber on the ⟨*name*⟩ (*i.e.* a jobname lacking any extension) inside the ⟨*dir*⟩. If there is no `.bcf` file then no action is taken with a return value of 0.

`bibtex()` `bibtex(⟨name⟩,⟨dir⟩)`

Runs BibTEX on the ⟨*name*⟩ (*i.e.* a jobname lacking any extension) inside the ⟨*dir*⟩. If there are no `\citation` lines in the `.aux` file then no action is taken with a return value of 0.

`makeindex()` `makeindex(⟨name⟩,⟨dir⟩,⟨inext⟩,⟨outext⟩,⟨logext⟩,⟨style⟩)`

Runs MakeIndex on the ⟨*name*⟩ (*i.e.* a jobname lacking any extension) inside the ⟨*dir*⟩. The various extensions and the ⟨*style*⟩ should normally be given as it standard for MakeIndex.

`tex()` `tex(⟨file⟩,⟨dir⟩,⟨cmd⟩)`

Runs ⟨*cmd*⟩ (by default `"pdflatex" "-interaction=nonstopmode"`) on the ⟨*name*⟩ inside the ⟨*dir*⟩.

`runcmd()` `runcmd(⟨cmd⟩,⟨dir⟩,{⟨envvars⟩})`

A generic function which runs the ⟨*cmd*⟩ in the ⟨*dir*⟩, first setting up all of the environmental variables specified to point to the `local` and `working` directories. This function is useful when creating non-standard typesetting steps.

5.6 Customising the target and option lists

The targets known to l3build are stored in the global table `target_list`. Each entry should have at least a `func`, pointing to the function used to implement the target. This function will receive the list of names given at the command line as a table argument. In most cases, targets will also have a `desc`, used to construct `help()` automatically. In addition, the following may also be used:

- `bundle_func` A variant of `func` used when at the top level of a bundle
- `bundle_target` A boolean to specify that when passing the target name in a bundle, it should have `bundle` prepended.
- `pre` A function executed before the main function, and receiving the `names` as an argument; this allows checking of the `name` data without impact on the main `func`.

The functions `func`, `bundle_func` and `pre` must return 0 on success.

The list of options (switches) is controlled by the `option_list` table. The name of each entry in the table is the "long" version of the option. Each entry requires a `type`, one of `boolean`, `string` or `table`. As for targets, each entry should have a `desc` to construct the `help()`. It is also possible to provide a `short` name for the option: this should be a single letter.

5.7 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 (§5.7.1).

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 (§5.7.2), the inclusion of one-line descriptions for each file (§5.7.3), and the details of the formatting of each entry (§5.7.4).

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()`.

5.7.1 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",
       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 3 and Table 4 See the complete setup code in `l3build-manifest-setup.lua` for examples of these in use.

Table 3: 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 4: Table entries used in the manifest setup table for a subheading.

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

5.7.2 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

5.7.3 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.)

5.7.4 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.