l3build: The beginner's guide

Joseph Wright

1 Introduction

For package authors, creating a release is a regular process, ideal for automation. There are several steps to creating a release to CTAN, for example ensuring documentation is updated, structuring an archive correctly and actually uploading the material.

Some time ago, the LaTeX Team extended their existing basic scripts to create an independent tool, l3build, which can cover all of those tasks. Most significantly, it included features to run comprehensive tests: this aspect was previously covered for *TUGboat* (2014, **35**:3, pp. 287–293). Here, I will give a more general overview of the tool, looking at how it can help package authors create releases in a quick and reliable manner.

2 l3build at the command line

With a modern TeX system, l3build is available as a command at the command line/terminal. It understands ⟨*targets*⟩, ⟨*options*⟩ and ⟨*arguments*⟩.

`l3build` ⟨*target*⟩ [⟨*options*⟩] [⟨*arguments*⟩]

The ⟨*target*⟩ is the task we want l3build to carry out. The most common ones are:

`check`	Runs one or more automated tests
`save`	Saves the result of one or more tests
`doc`	Typesets documentation
`ctan`	Creates a zip file ready to send to CTAN
`upload`	Sends a zip file to CTAN
`install`	Installs the package in the local `texmf` tree (there is also `uninstall` to reverse this)

The particular ⟨*options*⟩ which apply depend on the ⟨*target*⟩. For example, when running the `check` target, l3build will normally finish all of the tests then report the results. However, particularly when used with an automated system, one might want the tests to halt as soon as there is an error. That is available using the `--halt-on-error` option, which is also available as the one-letter version `-H`.

Some targets *require* one or more ⟨*arguments*⟩. For example, to save test results, you *have* to give the name of the test(s). Some targets take an optional ⟨*argument*⟩: `doc` is a good example, as you can limit this to a specific PDF (where your project has several PDFs, this can be useful). Finally, some targets do not need arguments at all: `install` is an example.

3 Configuration: the `build.lua` file

The configuration of l3build for a project is controlled by a file called `build.lua`, which should be present in the main directory. This is a Lua file, and so *can* contain sophisticated programming. However, for a large number of use cases, the requirements are simply to set either string variables or tables of strings. That means that for many projects, the `build.lua` file will comprise just a few short lines, and requires no insight into Lua programming.

Only one line is absolutely required: one to tell l3build the name of the package. This is specified as the `module` string:

```
module = "mypkg"
```

By the way, Lua will allow us to mark strings using either single or double quotes. I favour double ones, and only use single quotes if the string itself contains a double quote, but it's purely personal preference.

The standard settings in l3build are based around using one or more `.dtx` files extracted using an `.ins` file. They also assume that the documentation is in the `.dtx` files. One common structure with larger packages is to separate out the documentation from the code, so to have a `.tex` file to typeset. This can be covered using

```
typesetfiles = {"*.tex"}
```

or if we want to specify only specific files, for example:

```
typesetfiles =
  {
    "mypkg-doc.tex",
    "mypkg-example-a.tex"
  }
```

Here, we are using a Lua table: these can hold a variety of data, but all we need to know here is that we can use a comma-separated list of names inside braces.

If the project we are working on doesn't use the `.dtx` format, we need to tell l3build the name(s) of our source files, and that it can skip unpacking:

```
sourcefiles = {"*.def", "*.sty"}
unpackfiles = {}
```

Or we might unpack some files that are not on the standard list, in which case we need to tell l3build to install them:

```
installfiles = {"*.def", "*.sty"}
```

The standard settings for l3build assume that all of the source files are in the same directory as the `build.lua` file. Some authors prefer a more complex structure. For example, for LaTeX itself there are *lots* of documentation files, so they are inside a subdirectory:

```
docfiledir = "./doc"
```

You can do the same with your source files, for example if you want your main directory to hold just `build.lua` (and probably a `README.md`):

Joseph Wright

doi.org/10.47397/tb/43-1/tb133wright-l3build

```
sourcefiledir = "./source"
```

The system can cope with more complex layouts, for example with subdirectories. One new feature that can help with these more tricky cases is `tdsdirs`, which lets l3build simply copy an entire directory 'as is'. We tell the system the name of the directory, and where it matches up with in the TeX installation tree. For example, if we wanted to use the above `source` directory in its entirety, and install it into the `tex` tree, we would use

```
tdsdirs = {source = "tex"}
```

In this case, *all* of the files are used.

We will see later that there are settings that apply to tests, to creating CTAN releases, and for more advanced functions.

4 Setting up simple tests

The core mechanism for creating tests in l3build uses the fact that documents can write to the `.log` and extract information to verify that our code has worked. That can broadly be done in two ways: deliberately writing information to the `.log`, or using `\showoutput` or similar to place the result of some typesetting operation into the file.

What is also needed is a way to mark those parts of the `.log` that are of interest, and to normalise system-dependent information, such as paths, to make the results as portable as possible. Some of this is carried out by l3build itself, with the macro parts of the process implemented in the source file `regression-test.tex`. All the commands provided by the latter have all-uppercase names, to minimise the chance of clashes with normal commands.

For the case where it is possible to save a result in a macro, counter or similar, the easiest approach to testing is to write these using `\TYPEOUT`.

```
\input{regression-test}
\documentclass{article}
\usepackage{mypkg} % The package to test
\START
\TEST{A first test}{%
  \mypkfunctionA{input-tokens}%
    \outputmacro
  \TYPEOUT{\outputmacro}
}
\TEST{A second test}{%
  \mypkfunctionB
    {input-tokens}%
    {more-input-tokens}%
    \outputmacro
  \TYPEOUT{\outputmacro}
}
\END
```

Nothing before `\START` will be recorded, which makes it a good way to skip the preamble. We can skip small parts of the input using the pair `\OMIT` and `\TIMO`. The run here is stopped using `\END` as we are not interested in the typesetting of pages: this basically kills the TeX run and saves a bit of time.

The alternative approach is to look at TeX's output tracing, either using a box or `\showoutput`.

```
\input{regression-test}
\documentclass{article}
\usepackage{mypkg} % The package to test
\showoutput
\begin{document}
\START
% Assume the commands produce typeset output
\mypkfunctionA{input-tokens}

\mypkfunctionB
  {input-tokens}
  {more-input-tokens}
\newpage
\OMIT
\end{document}
```

Here, we can use `\OMIT` to skip over the information at the end of a TeX run: here we have used `\end{document}` as this allows the LaTeX `.aux` file, *etc.*, to be created. If you are relying on information passed using this mechanism, you might need to set

```
checkruns = 2
```

or some higher value.

The input files for tests, `.lvt` files, should be saved inside a directory `testfiles` within the project directory. The test results are then saved using

`l3build save` ⟨*names*⟩

where the ⟨*names*⟩ are the file names of the test inputs, but with the extension omitted.

With the standard settings, tests are run using pdfTeX, XeTeX and LuaTeX, and using the LaTeX format. Using formats other than LaTeX is outside of the scope of this short guide, but running with multiple engines is a common requirement. To save an engine-specific test result, we use the `--engine` (or `-e`) option

`l3build save -e`⟨*engine1*⟩`,`⟨*engine2*⟩ ⟨*names*⟩

This will be needed most commonly when testing typeset output: there are fundamental differences between the three common engines. When running

```
l3build check
```

the system will use engine-specific results if they exist, and otherwise will assume that they all follow the 'standard' engine: this is normally pdfTeX.

If you would rather just use one engine for tests, you can set

```
checkengines = {"pdftex"}
```

in your `build.lua` file. For Unicode-only work, in contrast, you might want

```
checkengines = {"xetex", "luatex"}
```

where the first entry given will then be the 'standard' engine.

5 Customising typesetting

There is only one command used for typesetting documentation: it can be set using the `typesetexe` setting. This is typically set to `pdflatex`: notice that this is a typesetting *command* not an *engine*.

As for tests, the number of typesetting runs can be set, using the `typesetruns` setting. More complex adjustment of the typesetting run is possible: l3build provides a set of basic operation functions (such as 'run Biber'), and these can be combined to make defined workflows. This aspect requires some Lua programming and is therefore beyond the scope of this short guide.

6 Building CTAN releases

The standard settings will collect up all sources and typeset files, plus any `README.md`, and create a zip file to send to CTAN. You can also pack a TDS-ready zip: this feature is activated using the setting

```
packtdszip = true
```

Uploading to CTAN requires some settings to 'fill out the form' for administration. As an example, l3build itself has the configuration shown in Figure 1. The `[[ ... ]]` syntax creates a multi-line string in Lua.

The information in `uploadconfig` is used by the `upload` target, which needs two key pieces of information: an email address and a release string. This will be requested by l3build if not given at the command line

`l3build upload --email` ⟨*email*⟩ ⟨*tag*⟩

You can check that your upload is valid, without actually sending it, by using the `--dry-run` option on the command line. (This option also works for the `install` target.)

7 Advanced features

Using a mixture of Lua programming and additional variables, a wide range of effects can be achieved. These include

- Supporting plain TEX and ConTEXt testing
- Automatically updating version strings and copyright in sources using the `tag` target
- Using multiple setups to run tests for different aspects of functionality
- Placing installed files in different parts of the TEX tree
- Testing the PDFs produced by typesetting

Of these, the ability to automatically tag files is probably of the broadest interest. However, as sources files are extremely varied, this does require some Lua programming; that takes us beyond the scope of this short article. For details of this and the other more advanced features, please consult the l3build manual.

8 Example `build.lua` files

8.1 A basic project: one `.dtx` and one `.ins`

The most basic setup, following the model used by the LATEX Team, is to have your code and documentation in a single `.dtx` file, which has a matching `.ins` file and (probably) a `README.md`, all in the same directory. For this, the `build.lua` file can be a single line:

```
module = "mypkg"
```

That's it: l3build will handle everything else based on its standard settings.

8.2 A 'self-extracting' `.dtx` file

Some people like to combine their `.ins` file into their `.dtx`; that is easy to support.[1]

```
module = "mypkg"
unpackfiles {"*.dtx"}
```

8.3 Documentation separate from sources

With larger projects, you may want your documentation in one or more `.tex` files separate from the code. Assuming you also want to typeset your code, you'd go with

```
module = "mypkg"
typesetfiles {"*.dtx", "*.tex"}
```

8.4 Not using DocStrip, and non-standard file types

Not everyone wants to use DocStrip, and while it won't hurt to leave unpacking enabled, we might well want to skip it. At the same time, we might have some non-standard file types: here some `.def` files and one `.lua` file.

```
module = "mypkg"
installfiles =
  {"*.def", "mypkg.lua", "*.sty"}
unpackfiles = {}
```

[1] I don't recommend this structure. You are unlikely to need to send your source by email to someone, and the only real benefit of a single-source approach is for that type of 'classical' distribution.

```
uploadconfig = {
  author      = "The LaTeX Team",
  license     = "lppl1.3c",
  summary     = "A testing and building system for (La)TeX",
  topic       = {"macro-supp", "package-devel"},
  ctanPath    = "/macros/latex/contrib/l3build",
  repository  = "https://github.com/latex3/l3build/",
  bugtracker  = "https://github.com/latex3/l3build/issues",
  update      = true,
  description = [[
The build system supports testing and building
(La)TeX code, on Linux, macOS, and Windows
systems. The package offers:
* A unit testing system for (La)TeX code;
* A system for typesetting package documentation; and
* An automated process for creating CTAN releases.
  ]]
}
```

Figure 1: `uploadconfig` for l3build itself

8.5 Source files in different directories

Some developers like to have their sources in different directories inside their project. This likely goes with having separate files for typesetting.

```
module = "mypkg"
docfiledir = "doc"
sourcefiledir = "source"
typesetfiles = {"*.tex"}
```

9 Summary of key settings

There are a large number of more specialised settings available in l3build. Table 1 summarises some of the most commonly-used ones. There is a full list in the package documentation.

Variable	Description
`module`	Name of the package
`installfiles`	List of files to place in the `texmf` tree
`sourcefiles`	List of sources/pre-extracted files
`typesetfiles`	List of sources to typeset
`unpackfiles`	List of `.ins` files to DocStrip
`docfiledir`	Location of typeset sources
`sourcefiledir`	Location of code sources
`tdsdirs`	Table of locations to install directly
`checkengines`	List of engines for test runs
`checkruns`	Number of (La)TeX runs for testing
`typesetexe`	Program to typeset documentation
`typesetruns`	Number of (La)TeX runs for typesetting
`packtdsdir`	Switch to build TDS-style zip file
`uploadconfig`	Table of information for uploading

Table 1: Summary of key settings

◇ Joseph Wright
Northampton, United Kingdom
joseph dot wright (at)
morningstar2.co.uk