qstest, a LaTeX package for unit tests

David Kastrup
David dot Kastrup (at) QuinScape dot de

Abstract

The qstest package was created because of the need to verify in a large LaTeX project that no regressions occur. The test environments ensure that macros and registers will be set to expected values in test code, and that macro calls occur in a certain sequence and with certain values. Tests are usually embedded directly into the source code of .dtx files, thus providing documentation as well as verification.

It is also possible to compare results of one test run to those of previous runs.

Several log files may be created simultaneously in order to record the results of tests ordered into various categories.

1 Using qstest

The basic idea of qstest is to let the user specify a number of tests that can be performed either at package load time or while running a separate test file through LaTeX. If you are writing .dtx files, it is a good idea to use docstrip 'modules' for specifying which lines are to be used for testing. The file qstest.dtx from which both the style file as well as this documentation have been generated has been written in this manner.

Since the tests should be ignored when the dtx file is itself compiled, we use this for skipping over the tests:

```
⟨*dtx⟩
\iffalse
⟨/dtx⟩
```

A standalone test file does not need a preamble. We can load the packages with \RequirePackage and just go ahead. Let us demonstrate how to build such a test file by testing the qstest package itself:

```
⟨*test⟩
\RequirePackage{qstest}
```

1.1 Pattern and keyword lists

See the section "Match patterns and targets" of the makematch package for an explanation of the comma-separated pattern and keyword lists. In a nutshell, both are lists of arbitrary material that is not expanded but rather used in sanitized (printable) form. Patterns may contain wildcard characters * matching zero or more characters, and may be preceded by ! in order to negate a match from an earlier pattern in the pattern list. Leading spaces before an item in either list are discarded.

1.2 Specifying test sets

\IncludeTests specifies a pattern list matched to tests' keyword lists in order to determine the tests to be included in this test run. The characters * and ! are interpreted as explained above.

For example,

```
\IncludeTests{*, !\cs}
```

will run all tests except those that have a test keyword of \cs in their list of keywords. It is a good convention to specify the principal macro or environment to be tested as the first keyword.

The default is to include all tests. If you are interspersing possibly expensive tests with your source file, you might want to specify something like

```
\IncludeTests{*, !expensive}
```

or even

```
\IncludeTests{}
```

in your document preamble, and then possibly override this on the command line with

```
latex "\AtBeginDocument{
\IncludeTests{*}}\input{file}"
```

or similar for getting a more complete test.

\TestErrors defines test patterns that will throw an error when failing. A test that throws an error will not also add a warning to the standard log file with extension log since that would be redundant.

The default is \TestErrors{*, !fails}, to have all tests that are not marked as broken throw an error when they fail.

The throwing of errors does not depend on the logging settings (see below) for the default log file.

\LogTests receives three arguments. The first is the filename extension of a log file (the extension

`log` is treated specially and uses package warning and info commands to log test failures and passes, respectively). The second is a keyword list that indicates which passed tests are to be logged. The third is a keyword list specifying which failed tests are to be logged. Let us open a file logging everything:

```
\LogTests{lgout}{*}{*}
```

The choice of `lgout` is made to make it possible to also have `lgin` for comparison purposes: the latter would be an `lgout` file from a previous, 'definitive run', renamed and checked into version control, for the sake of being able to compare the log output from different versions.

An already open log file stays open and just changes what is logged. By default, the standard `log` (pseudo-)file is already open and logs everything.

Passed and failed tests are not completely symmetric with regard to logging: test failures are logged and/or indicated on the individual failed assertions, while a successful test is only logged and/or indicated in summary.

With `\LogClose` you can explicitly close a log file if you want to reread it in the course of processing, or call an executable that would process it. The standard file with extension `log` will not actually get closed and flushed if you do this (though logging would stop on it), but all others might. An actual example for this follows after the tests. You can reopen a closed log file using `\LogTests`. It will then get rewritten from the beginning (with the exception of the standard `log` file, of course).

1.3 The tests

Tests are performed within the `qstest` environment. The environment gets two arguments. The first is the name of the test recorded in the log file. The second is a list of test keywords used for deciding which tests are performed and logged.

Before delving into the details of what kind of tests you can perform in this environment, we list the various commands that are given patterns and thus control what kind of tests are performed and logged.

`\Expect` is the workhorse for checking that values, dimensions, macros and other things are just what the test designer would expect them to be.

This macro basically receives two brace-delimited arguments[1] and checks that they are equal after being passed through `\def` and sanitized. This means that you can't normally use `#` except when followed by a digit (all from 1 to 9 are allowed) or `#`. If you precede one of those arguments with `*` it gets passed through `\edef` instead of `\def`. There may also be additional tokens like `\expandafter` before the opening brace. Note that the combination of `\edef` and `\the`⟨*token variable*⟩ can be used to pass through `#` characters without interpretation. ε-TEX provides a similar effect with `\unexpanded`. So if you want to compare a token list that may contain isolated hash characters, you can do so by writing something like

```
⟨*etex⟩
\begin{qstest}{# in isolation}
              {\Expect, #, \unexpanded}
  \toks0{# and #}
  \Expect*{\the\toks0}
         *{\unexpanded{# and #}}
\end{qstest}
⟨/etex⟩
```

Since the sanitized version will convert `#` macro parameters to the string `##`, you might also do this explicitly (and without ε-TEX) as

```
\begin{qstest}{# in isolation 2}
              {\Expect, #, \string}
  \toks0{# and #}
  \Expect*{\the\toks0}
         *{\string#\string#
           and \string#\string#}
\end{qstest}
```

If the token register is guaranteed to contain only 'proper' `#` characters that are followed by either another `#` or a digit, you can let the normal interpretation of a macro parameter for `\def` kick in and use this as

```
\begin{qstest}{# as macro parameter}
              {\Expect, #}
  \toks0{\def\xxx#1{}}
  \Expect\expandafter{\the\toks0}
                     {\def\xxx#1{}}
\end{qstest}
```

In this manner, `#1` is interpreted (and sanitized) as a macro parameter on both sides, and consequently no doubling of `#` occurs.

Before the comparison is done, both arguments are sanitized, converted into printing characters with standardized catcodes throughout.[2] A word of warning: both sanitization as well as using `\meaning` still depend on catcode settings, since single-letter control sequences (made from a catcode 11 letter) are followed by a space, and other single-character control sequences are not. For this reason,

[1] The arguments are collected with a token register assignment. This gives several options for specifying them, including giving a token register without braces around it. It also makes it possible to precede the *balanced text* with `\expandafter` and similar expandable stuff.

[2] Spaces get catcode 10, all other characters catcode 12.

a standalone test file for LaTeX class or package files will usually need to declare

```
\makeatletter
```

in order to make '@' a letter, as is usual in such files.

All of the following expectations would turn out correct:

```
\begin{qstest}{Some LaTeX definitions}
              {\Expect}
  \Expect*{\meaning\@gobble}
          {\long macro:#1->}
  \Expect*{\the\maxdimen}
          {16383.99998pt}
\end{qstest}
```

Note that there is no way to convert the contents of a box into a printable rendition, so with regard to boxes, you will mostly be reduced to checking that the box dimensions meet expectations.

1.4 Expecting ifthen conditions

`\ExpectIfThen` is used for evaluating a condition as provided by the `ifthen` package. See its docs for the kind of condition that is possible there. You just specify one argument: the condition that you expect to be true. Here is an example:

```
\RequirePackage{ifthen}
\begin{qstest}{\ExpectIfThen}
              {\ExpectIfThen}
  \ExpectIfThen{
     \lengthtest
       {\maxdimen=16383.99998pt}\AND
        \maxdimen>1000000000}
\end{qstest}
```

1.5 Dimension ranges

`\InRange` checks not whether some dimension is exactly equal to some value, but rather within some range. We do this by specifying as the second argument to `\Expect` an artificial macro with two arguments specifying the range in question. This will make `\Expect` succeed if its first argument is in the range specified by the two arguments to `\InRange`.

The range is specified as two TeX dimens. If you use a dimen register and you want to have a possible error message display the value instead of the dimen register, you can do so by using the `*` modifier before `\InRange` (which will cause the value to be expanded before printing and comparing) and put `\the` before the dimen register since such registers are not expandable by themselves.

Here are some examples:

```
\begin{qstest}{\InRange success}
              {\InRange}
  \dimen@=10pt
  \Expect*{\the\dimen@}
          \InRange{5pt}{15pt}
  \Expect*{\the\dimen@}
          \InRange{10pt}{15pt}
  \Expect*{\the\dimen@}
          \InRange{5pt}{10pt}
\end{qstest}
\begin{qstest}{\InRange failure}
              {\InRange, fails}
  \dimen@=10pt \dimen@ii=9.99998pt
  \Expect*{\the\dimen@}
           \InRange{5pt}{\dimen@ii}
  \dimen@ii=10.00002pt
  \Expect*{\the\dimen@}
          *\InRange{\the\dimen@ii}{15pt}
\end{qstest}
```

`\NearTo` requires ε-TeX's arithmetic and so will not be available for versions built without ε-TeX support. The macro is used in lieu of an expected value and is similar to `\InRange` in that it is a pseudovalue to be used for the second argument of `\Expect`. It makes `\Expect` succeed if its own mandatory argument is close to the first argument from `\Expect`, where closeness is defined as being within `0.05pt`. This size can be varied by specifying a different one as optional argument to `\NearTo`. Here is a test:

```
⟨*etex⟩
\begin{qstest}{\NearTo success}
              {\NearTo}
  \dimen@=10pt
  \Expect*{\the\dimen@}
          \NearTo{10.05pt}
  \Expect*{\the\dimen@}
          \NearTo{9.95pt}
  \Expect*{\the\dimen@}
          \NearTo[2pt]{12pt}
  \Expect*{\the\dimen@}
          \NearTo[0.1pt]{9.9pt}
\end{qstest}
\begin{qstest}{\NearTo failure}
              {\NearTo, fails}
  \dimen@=10pt
  \Expect*{\the\dimen@}
          \NearTo{10.05002pt}
  \Expect*{\the\dimen@}
          \NearTo[1pt]{11.00001pt}
\end{qstest}
⟨/etex⟩
```

1.6 Saved results

The purpose of saved results is to be able to check that the value has remained the same over multiple passes. Results are given a unique label name and are written to an auxiliary file where they can be read in for the sake of comparison. One can use the normal `aux` file for this purpose, but it might be preferable to use a separate dedicated file. That

way it is possible to input a versioned *copy* of this file and have a fixed point of reference rather than the last run.

While the `aux` file is read in automatically at the beginning of the document, this does not happen with explicitly named files. You have to read them in yourself, preferably using

```
\InputIfFileExists
{filename}{}{}
```

so that no error is thrown when the file does not yet exist.

`\SaveValueFile` gets one argument specifying which file name to use for saving results. If this is specified, a special file is opened. If `\SaveValueFile` is not called, the standard `aux` file is used instead, but then you can only save values after `\begin{document}`. `\jobname.qsout` seems like a useful file name to use here (the extension `out` is already in use by pdfTEX).

```
\begin{qstest}{\SavedValue}
              {\SavedValue}
  \SaveValueFile{\jobname.qsout}
```

If this were a real test instead of just documentation, we probably would have written something like

```
\InputIfFileExists
  {\jobname.qsin}{}{}
```

first in order to read in values from a previous run. The given file would have been a copy of a previous `qsout` file, possibly checked into version control in order to make sure behavior is consistent across runs. If it is an error to not have such a file (once you have established appropriate testing), you can just write

```
\input{\jobname.qsin}
```

instead, of course.

`\CloseValueFile` takes no argument and will close a value save file if one is open (using this has no effect if no file has been opened and values are saved on the `aux` file instead). We'll demonstrate use of it later. It is probably only necessary for testing `qstest` itself (namely, when you read in saved values in the same run), or when you do the processing/comparison with a previous version by executing commands via TEX's `\write18` mechanism.

`\SaveValue` gets the label name as first argument. If you are using the non-ε-TEX version, the label name gets sanitized using `\string` and so can't deal with non-character material except at its immediate beginning. The ε-TEX version has no such constraint.

The second argument is the same kind of argument as `\Expect` expects, namely something suitable for a token register assignment which is passed through `\def` if not preceded by `*`, and by `\edef` if preceded by `*`. The value is written out to the save file where it can be read in afterwards.

Let us save a few values under different names now:

```
\SaveValue{\InternalSetValue}
          *{\meaning\InternalSetValue}
\SaveValue{\IncludeTests}
          *{\meaning\IncludeTests}
\SaveValue{whatever}
          *{3.1415}
\SaveValue{\maxdimen}
          *{\the\maxdimen}
```

A call to `\InternalSetValue` is placed into the save file for each call of `\SaveValue`. The details are not really relevant: in case you run into problems while inputting the save file, it might be nice to know.

`\SavedValue` is used for retrieving a saved value. When used as the second argument to `\Expect`, it will default to the value of the first argument to `\Expect` unless it has been read in from a save file. This behavior is intended for making it easy to add tests and saved values and not get errors at first, until actually values from a previous test become available.

Consequently, the following tests will all turn out true before we read in a file, simply because all the saved values are not yet defined and default to the expectations:

```
\Expect{Whatever}
       \SavedValue{\InternalSetValue}
\Expect[\IncludeTests]{Whatever else}
        \SavedValue{\IncludeTests}
\Expect[whatever]{2.71828}
        \SavedValue{whatever}
\Expect[undefined]{1.618034}
        \SavedValue{undefined}
```

After closing and rereading the file, we'll need to be more careful with our expectations, but the last line still works since there still is no saved value for "undefined".

```
\CloseValueFile
\input{\jobname.qsout}
\Expect*{\meaning\InternalSetValue}
        \SavedValue{\InternalSetValue}
\Expect[\IncludeTests]
       *{\meaning\IncludeTests}%
       \SavedValue{\IncludeTests}
\Expect[whatever]{3.1415}
       \SavedValue{whatever}
\Expect[undefined]{1.618034}
       \SavedValue{undefined}
\end{qstest}
```

Now let's take the previous tests which succeeded again and let them fail now that the variables are defined:

```
\begin{qstest}{\SavedValue failure}
              {\SavedValue,fails}
  \Expect{Whatever}
         \SavedValue{\InternalSetValue}
  \Expect[\IncludeTests]{Whatever else}
         \SavedValue{\IncludeTests}
  \Expect{2.71828}\SavedValue{whatever}
\end{qstest}
```

1.7 Checking for call sequences

The environment `ExpectCallSequence` tells the test system that macros are going to be called in a certain order and with particular types of arguments.

It gets as an argument the expected call sequence. The call sequence contains entries that look like a macro definition: starting with the macro name followed with a macro argument list and a brace-enclosed substitution text that gets executed in place of the macro. The argument list given to this macro substitution will get as its first argument a macro with the original definition of the control sequence, so you can get at the original arguments for this particular macro call starting with `#2`. As a consequence, if you specify no arguments at all and an empty replacement text for the substitution, the original macro gets executed with the original arguments.

`\CalledName`: if you want to get back from the control sequence with the original meaning in `#1` to the original macro name, you can use `\CalledName` on it. This will expand to the original control sequence *name*, all in printable characters fit for output or typesetting in a typewriter font (or use in `\csname`), but without the leading backslash character. You can get to the control sequence itself by using

```
\csname \CalledName#1\endcsname
```

and to a printable version including backslash by using

```
\expandafter \string
\csname \CalledName#1\endcsname
```

Going into more detail, a substitution function is basically defined using

```
\protected \long \def
```

so it will not usually get expanded except when hit with `\expandafter` or actually being executed. Note that you can't use this on stuff that has to work at expansion time. This works mainly with macros that would also be suitable candidates for `\DeclareRobustCommand`.

It is also a bad idea to trace a conditional in this manner: while the substitution could be made to work when being executed, it will appear like an ordinary macro when being skipped, disturbing the conditional nesting.

Only macros occuring somewhere in the call sequence will get tracked, other macros are not affected. The environment can actually get nested, in which case the outer sequences will get tracked independently from the inner sequence.

Thus, `ExpectCallSequence` can be used in order to spoof, for example, both input consuming and output producing macros without knowing the exact relationship of both.

Apart from specifying macro calls, the call sequence specification can contain the following characters that also carry a special meaning:

`‘` If this is set in the call sequence, it places the initial parsing state here. This will make it an error if non-matching entries occur at the start of the sequence, which otherwise is effectively enclosed with

`.{}*(`⟨*sequence*⟩`).{}*`

meaning that nonmatching entries before the first and after the last matching item of the sequence are ignored by default (this makes it closer to normal regexp matchers). Since the matching will then start at `‘`, you can put macros before that position that you want to be flagged if they occur in the sequence, even when they are mentioned nowhere else (macros which would be an error if actually called). Also available as the more customary `^` character, but that tends to behave worse in LaTeX-aware editors.

`’` This indicates the last call sequence element to be matched. If any traced macros appear after this point, an error will get generated. Any immediately following call sequence entries will not get reached.

`.` A single dot indicates a wildcard: any of the tracked control sequences might occur here. You still have to follow this with macro arguments and a braced replacement text. Wildcards are considered as a fallback when nothing else matches.

`(...)` Parens may be used for grouping alternatives and/or combining items for the sake of repeating specifications, of which there are three:

`?` If a question mark follows either a macro call, wildcard call, parenthesized group, or call sequence end, the item before it is optional.

+ A plus sign following an item means that this item may be repeated one or more times.
* An asterisk following an item means that this item may be repeated zero or more times.
| A vertical bar separates alternatives. Alternatives extend as far as possible, to the next bar, to an enclosing paren group, or to the start and/or end of the whole call sequence specification if nothing else intervenes.

Note that in contrast to traditional regexp evaluation, no backtracking is employed: at each point in the call sequence, the next match is immediately chosen and a choice cannot (for obvious reasons) be reverted. It is the task of the user to specify a call sequence in a sufficiently non-ambiguous manner that will make the call sequence tracing not pick dead ends.

```
\begin{qstest}{ExpectCallSequence}
              {ExpectCallSequence}
  \def\e{e} \def\f{f}
  \def\g{g} \def\h{h}
  \begin{ExpectCallSequence}
        {`\e#1{%
            \Expect\expandafter
              {\csname\CalledName#1\endcsname}
              {\e }%
            \Expect*{\meaning#1}
            {macro:->e}}+\f#1{}'}
    \e \e \e \e \f
  \end{ExpectCallSequence}
\end{qstest}
```

1.8 Ending a standalone test file

One finishes a standalone test file by calling the LaTeX control sequence `\quit`. This stops processing even when we did not actually get into a document. We don't actually do this here since there will be further tests in the full documentation file. However, we will now close the log file we opened for this demonstration.

```
\LogClose{lgout}
⟨/test⟩
```

2 Conclusion

The package documentation illustrated how one can embed test cases into the source of a `dtx` package by using module guards `<test>` and `docstrip`. There are more possibilities of use, such as using `<trace>` guards and embedding `\Expect` macros and call sequence expectations right into code for regular use instead of doing separate tests. In that way, a debugging version of the package may be extracted using `docstrip`. Selecting a subset of trace commands or assertions to use can easily be accomplished with the `makematch` package.

The `qstest` package in combination with the `dtx` documentation format and `docstrip` allows to integrate documentation and unit testing. As long as one does not do actual testing, the `qstest` package is not required to be installed for either compiling documentation or using the style file. For that reason, one can safely use it without having to assume anything about the version (if any) of the `qstest` package available to some end user.