The l3docstrip package
Code extraction and manipulation*

The LaTeX3 Project†

Released 2012/06/08

1 Extending DocStrip

The l3docstrip module adds LaTeX3 extensions to the DocStrip program for extracting code from `.dtx`. As such, this documentation should be read along with that for DocStrip.

2 Internal functions and variables

An important consideration for LaTeX3 development is separating out public and internal functions. Functions and variables which are private to one module should not be used or modified by any other module. As TeX does not have any formal namespacing system, this requires a convention for indicating which functions in a code-level module are public and which are private.

Using l3docstrip allows internal functions to be indicated using a "two part" system. Within the `.dtx` file, internal functions may be indicated using `@@` in place of the module name, for example

```
\cs_new_protected:Npn \@@_some_function:nn #1#2
  {
    % Some code here
  }
\tl_new:N \l_@@_internal_tl
```

To extract the code using l3docstrip, the "guard" concept used by DocStrip is extended by introduction of the syntax `%<@@=`⟨*module*⟩. The ⟨*module*⟩ name will be used when the code is extracted to replace the `@@`, so that

```
%<*package>
%<@@=foo>
\cs_new_protected:Npn \@@_some_function:nn #1#2
```

*This file describes v3787, last revised 2012/06/08.

†E-mail: latex-team@latex-project.org

```
    {
      % Some code here
    }
\tl_new:N \l_@@_internal_tl
%</package>
```

will be extracted as

```
\cs_new_protected:Npn \__foo_some_function:nn #1#2
    {
      % Some code here
    }
\tl_new:N \l__foo_internal_tl
```

where the `__` indicates that the functions and variables are internal to the `foo` module.

3 l3docstrip implementation

⟨*program⟩

We start by loading the existing DocStrip code using the TEX input convention.

```
\input docstrip %
```

`\checkOption` The `\checkOption` macro is defined by DocStrip and is redefined here to accommodate the new `%<@` syntax.

When the macros that process a line have found that the line starts with "`%<`", a guard line has been encountered. The first character of a guard can be an asterisk (`*`), a slash (`/`), a plus (`+`), a minus (`-`), a less-than sign (`<`) starting verbatim mode, a commercial at (`@`) or any other character that can be found in an option name. This means that we have to peek at the next token and decide what kind of guard we have. We reinsert `#1` as it may be needed by `\doOption`.

```
\def\checkOption<#1{%
  \ifcase
    \ifx*#10\else \ifx/#11\else
    \ifx+#12\else \ifx-#13\else
    \ifx<#14\else \ifx @#15\else 6\fi\fi\fi\fi\fi\fi\relax
  \expandafter\starOption\or
  \expandafter\slashOption\or
  \expandafter\plusOption\or
  \expandafter\minusOption\or
  \expandafter\verbOption\or
  \expandafter\moduleOption\or
  \expandafter\doOption\fi
  #1%
}
```

(*End definition for* `\checkOption`. *This function is documented on page* ??.)

`\moduleOption` In the case where the line starts `%<@`: the defined syntax requires that this continues to `%<@@=`. At the moment, we assume that the syntax will be correct and `#1` here is the module name for substitution into any internal functions in the extracted material.

```
\def\moduleOption @@=#1>#2\endLine{%
  \maybeMsg{<@@=#1>}%
  \prepareActiveModule{#1}%
}
```

(*End definition for* `\moduleOption`*. This function is documented on page* ??.)

`\prepareActiveModule` `\replaceModuleInLine` Here, we set up to do the search-and-replace when doing the extraction. The argument (`#1`) is the replacement text to use, or if empty an indicator that no replacement should be done. The search material is one of `__@@`, `_@@` or `@@`, done in order such that all three will end up the same in the output. The replacement function is initialised as a do-nothing for the case where `%<@@=` is never seen.

```
\begingroup
  \catcode`\_ = 12 %
  \long\gdef\prepareActiveModule#1{%
    \ifx\relax#1\relax
       \let\replaceModuleInLine\empty
    \else
      \def\replaceModuleInLine{%
        \replaceAllIn\inLine{__@@}{__#1}%
        \replaceAllIn\inLine{_@@}{__#1}%
        \replaceAllIn\inLine{@@}{__#1}%
      }%
    \fi
  }
\endgroup
\let\replaceModuleInLine\empty
```

(*End definition for* `\prepareActiveModule`*. This function is documented on page* ??.)

`\replaceAllIn` `\replaceAllInAuxI` `\replaceAllInAuxII` `\replaceAllInAuxIII` The code here is a simple search-and-replace routine for a macro `#1`, replacing `#2` by `#3`. As set up here, there is an assumption that nothing is going to be expandable, which is reasonable as l3docstrip deals with "string" material.

```
\long\def\replaceAllIn#1#2#3{%
  \long\def\tempa##1##2#2{%
    ##2\qMark\replaceAllInAuxIII#3##1%
  }%
  \edef#1{\expandafter\replaceAllInAuxI#1\qMark#2\qStop}%
}
\def\replaceAllInAuxI{%
  \expandafter\replaceAllInAuxII\tempa\replaceAllInAuxI\empty
}
\long\def\replaceAllInAuxII#1\qMark#2{#1}
\long\def\replaceAllInAuxIII#1\qStop{}
```

(*End definition for* `\replaceAllIn`*. This function is documented on page* ??.)

`\normalLine` The `\normalLine` macro is present in DocStrip but is modified here to include the search-and-replace macro `\replaceModuleInLine`. The macro `\normalLine` writes its argument (which has to be delimited with `\endLine`) on all active output files, i.e. those with off-counters equal to zero. The counter `\codeLinesPassed` is incremented by 1 for statistics (the guards for this used in DocStrip are retained).

```
\def\normalLine#1\endLine{%
⟨*stats⟩
  \advance\codeLinesPassed\@ne
⟨/stats⟩
  \maybeMsg{.}%
  \def\inLine{#1}%
  \replaceModuleInLine
  \let\do\putline@do
  \activefiles
}
```

(*End definition for* `\normalLine`. *This function is documented on page* **??**.)

⟨/program⟩