The l3file module
File and I/O operations

The LaTeX Project*

Released 2024-01-04

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix `\file_...`, while others are used to work with files on a line by line basis and have prefix `\ior_...` (reading) or `\iow_...` (writing).

It is important to remember that when reading external files TeX attempts to locate them using both the operating system path and entries in the TeX file database (most TeX systems use such a database). Thus the "current path" for TeX is somewhat broader than that for other programs.

For functions which expect a ⟨*file name*⟩ argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Active characters (as declared in `\l_char_active_seq`) are *not* expanded, allowing the direct use of these in file names. Quote tokens (`"`) are not permitted in file names as they are reserved for internal use by some TeX primitives.

Spaces are trimmed at the beginning and end of the file name: this reflects the fact that some file systems do not allow or interact unpredictably with spaces in these positions. When no extension is given, this will trim spaces from the start of the name only.

1 Input–output stream management

As TeX engines have a limited number of input and output streams, direct use of the streams by the programmer is not supported in LaTeX3. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

Note that I/O operations are global: streams should all be declared with global names and treated accordingly.

`\ior_new:N`
`\ior_new:c`
`\iow_new:N`
`\iow_new:c`

New: 2011-09-26
Updated: 2011-12-27

`\ior_new:N` ⟨*stream*⟩
`\iow_new:N` ⟨*stream*⟩

Globally reserves the name of the ⟨*stream*⟩, either for reading or for writing as appropriate. The ⟨*stream*⟩ is not opened until the appropriate `\..._open:Nn` function is used. Attempting to use a ⟨*stream*⟩ which has not been opened is an error, and the ⟨*stream*⟩ will behave as the corresponding `\c_term_...`.

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

\ior_open:Nn
\ior_open:cn

Updated: 2012-02-10

\ior_open:Nn ⟨*stream*⟩ {⟨*file name*⟩}

Opens ⟨*file name*⟩ for reading using ⟨*stream*⟩ as the control sequence for file access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*file name*⟩ until a `\ior_close:N` instruction is given or the TEX run ends. If the file is not found, an error is raised.

\ior_open:NnTF
\ior_open:cnTF

New: 2013-01-12

\ior_open:NnTF ⟨*stream*⟩ {⟨*file name*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Opens ⟨*file name*⟩ for reading using ⟨*stream*⟩ as the control sequence for file access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*file name*⟩ until a `\ior_close:N` instruction is given or the TEX run ends. The ⟨*true code*⟩ is then inserted into the input stream. If the file is not found, no error is raised and the ⟨*false code*⟩ is inserted into the input stream.

\iow_open:Nn
\iow_open:(NV|cn|cV)

Updated: 2012-02-09

\iow_open:Nn ⟨*stream*⟩ {⟨*file name*⟩}

Opens ⟨*file name*⟩ for writing using ⟨*stream*⟩ as the control sequence for file access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*file name*⟩ until a `\iow_close:N` instruction is given or the TEX run ends. Opening a file for writing clears any existing content in the file (*i.e.* writing is *not* additive).

\ior_shell_open:Nn

New: 2019-05-08

\ior_shell_open:Nn ⟨*stream*⟩ {⟨*shell command*⟩}

Opens the *pseudo*-file created by the output of the ⟨*shell command*⟩ for reading using ⟨*stream*⟩ as the control sequence for access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*shell command*⟩ until a `\ior_close:N` instruction is given or the TEX run ends. If piped system calls are disabled an error is raised.

For details of handling of the ⟨*shell command*⟩, see `\sys_get_shell:nnNTF`.

\iow_shell_open:Nn

New: 2023-05-25

\iow_shell_open:Nn ⟨*stream*⟩ {⟨*shell command*⟩}

Opens the *pseudo*-file created by the output of the ⟨*shell command*⟩ for writing using ⟨*stream*⟩ as the control sequence for access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*shell command*⟩ until a `\iow_close:N` instruction is given or the TEX run ends. If piped system calls are disabled an error is raised.

For details of handling of the ⟨*shell command*⟩, see `\sys_get_shell:nnNTF`.

\ior_close:N
\ior_close:c
\iow_close:N
\iow_close:c

Updated: 2012-07-31

\ior_close:N ⟨*stream*⟩
\iow_close:N ⟨*stream*⟩

Closes the ⟨*stream*⟩. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.

\ior_show:N
\ior_show:c
\ior_log:N
\ior_log:c
\iow_show:N
\iow_show:c
\iow_log:N
\iow_log:c

\ior_show:N ⟨*stream*⟩
\ior_log:N ⟨*stream*⟩
\iow_show:N ⟨*stream*⟩
\iow_log:N ⟨*stream*⟩

Display (to the terminal or log file) the file name associated to the (read or write) ⟨*stream*⟩.

New: 2021-05-11

\ior_show_list:
\ior_log_list:
\iow_show_list:
\iow_log_list:

\ior_show_list:
\ior_log_list:
\iow_show_list:
\iow_log_list:

New: 2017-06-27

Display (to the terminal or log file) a list of the file names associated with each open (read or write) stream. This is intended for tracking down problems.

1.1 Reading from files

Reading from files and reading from the terminal are separate processes in expl3. The functions \ior_get:NN and \ior_str_get:NN, and their branching equivalents, are designed to work with *files*.

\ior_get:NN
\ior_get:NNTF

\ior_get:NN ⟨*stream*⟩ ⟨*token list variable*⟩
\ior_get:NNTF ⟨*stream*⟩ ⟨*token list variable*⟩ ⟨*true code*⟩ ⟨*false code*⟩

New: 2012-06-24
Updated: 2019-03-23

Function that reads one or more lines (until an equal number of left and right braces are found) from the file input ⟨*stream*⟩ and stores the result locally in the ⟨*token list*⟩ variable. The material read from the ⟨*stream*⟩ is tokenized by TEX according to the category codes and \endlinechar in force when the function is used. Assuming normal settings, any lines which do not end in a comment character % have the line ending converted to a space, so for example input

```
a b  c
```

results in a token list a␣b␣c␣. Any blank line is converted to the token \par. Therefore, blank lines can be skipped by using a test such as

```
\ior_get:NN \l_my_stream \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl { \par }
\tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl
...
```

Also notice that if multiple lines are read to match braces then the resulting token list can contain \par tokens. In the non-branching version, where the ⟨*stream*⟩ is not open the ⟨*tl var*⟩ is set to \q_no_value.

TEXhackers note: This protected macro is a wrapper around the TEX primitive \read. Regardless of settings, TEX replaces trailing space and tab characters (character codes 32 and 9) in each line by an end-of-line character (character code \endlinechar, omitted if \endlinechar is negative or too large) before turning characters into tokens according to current category codes. With default settings, spaces appearing at the beginning of lines are also ignored.

\ior_str_get:NN `\ior_str_get:NN ⟨stream⟩ ⟨token list variable⟩`
\ior_str_get:NN*TF* `\ior_str_get:NNTF ⟨stream⟩ ⟨token list variable⟩ ⟨true code⟩ ⟨false code⟩`

New: 2016-12-04
Updated: 2019-03-23

Function that reads one line from the file input ⟨*stream*⟩ and stores the result locally in the ⟨*token list*⟩ variable. The material is read from the ⟨*stream*⟩ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). Multiple whitespace characters are retained by this process. It always only reads one line and any blank lines in the input result in the ⟨*token list variable*⟩ being empty. Unlike `\ior_get:NN`, line ends do not receive any special treatment. Thus input

```
a b  c
```

results in a token list `a b  c` with the letters `a`, `b`, and `c` having category code 12. In the non-branching version, where the⟨*stream*⟩ is not open the ⟨*tl var*⟩ is set to `\q_no_value`.

TEXhackers note: This protected macro is a wrapper around the ε-TEX primitive `\readline`. Regardless of settings, TEX removes trailing space and tab characters (character codes 32 and 9). However, the end-line character normally added by this primitive is not included in the result of `\ior_str_get:NN`.

All mappings are done at the current group level, *i.e.* any local assignments made by the ⟨*function*⟩ or ⟨*code*⟩ discussed below remain in effect after the loop.

\ior_map_inline:Nn `\ior_map_inline:Nn ⟨stream⟩ {⟨inline function⟩}`

New: 2012-02-11

Applies the ⟨*inline function*⟩ to each set of ⟨*lines*⟩ obtained by calling `\ior_get:NN` until reaching the end of the file. TEX ignores any trailing new-line marker from the file it reads. The ⟨*inline function*⟩ should consist of code which receives the ⟨*line*⟩ as `#1`.

\ior_str_map_inline:Nn `\ior_str_map_inline:Nn ⟨stream⟩ {⟨inline function⟩}`

New: 2012-02-11

Applies the ⟨*inline function*⟩ to every ⟨*line*⟩ in the ⟨*stream*⟩. The material is read from the ⟨*stream*⟩ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The ⟨*inline function*⟩ should consist of code which receives the ⟨*line*⟩ as `#1`. Note that TEX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. TEX also ignores any trailing new-line marker from the file it reads.

\ior_map_variable:NNn `\ior_map_variable:NNn ⟨stream⟩ ⟨tl var⟩ {⟨code⟩}`

New: 2019-01-13

For each set of ⟨*lines*⟩ obtained by calling `\ior_get:NN` until reaching the end of the file, stores the ⟨*lines*⟩ in the ⟨*tl var*⟩ then applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last set of ⟨*lines*⟩, or its original value if the ⟨*stream*⟩ is empty. TEX ignores any trailing new-line marker from the file it reads. This function is typically faster than `\ior_map_inline:Nn`.

\ior_str_map_variable:NNn `\ior_str_map_variable:NNn ⟨stream⟩ ⟨variable⟩ {⟨code⟩}`

New: 2019-01-13 For each ⟨*line*⟩ in the ⟨*stream*⟩, stores the ⟨*line*⟩ in the ⟨*variable*⟩ then applies the ⟨*code*⟩. The material is read from the ⟨*stream*⟩ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*line*⟩, or its original value if the ⟨*stream*⟩ is empty. Note that TEX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. TEX also ignores any trailing new-line marker from the file it reads. This function is typically faster than `\ior_str_map_inline:Nn`.

\ior_map_break: `\ior_map_break:`

New: 2012-06-29 Used to terminate a `\ior_map_...` function before all lines from the ⟨*stream*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \ior_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a `\ior_map_...` scenario leads to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\ior_map_break:n `\ior_map_break:n {⟨code⟩}`

New: 2012-06-29 Used to terminate a `\ior_map_...` function before all lines in the ⟨*stream*⟩ have been processed, inserting the ⟨*code*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \ior_map_break:n { <code> } }
      {
        % Do something useful
      }
  }
```

Use outside of a `\ior_map_...` scenario leads to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨*code*⟩ is inserted into the input stream. This depends on the design of the mapping function.

\ior_if_eof_p:N ⋆ `\ior_if_eof_p:N ⟨stream⟩`
\ior_if_eof:NTF ⋆ `\ior_if_eof:NTF ⟨stream⟩ {⟨true code⟩} {⟨false code⟩}`

Updated: 2012-02-10

Tests if the end of a file ⟨*stream*⟩ has been reached during a reading operation. The test also returns a `true` value if the ⟨*stream*⟩ is not open.

1.2 Reading from the terminal

\ior_get_term:nN
\ior_str_get_term:nN

`\ior_get_term:nN ⟨prompt⟩ ⟨token list variable⟩`

New: 2019-03-23

Function that reads one or more lines (until an equal number of left and right braces are found) from the terminal and stores the result locally in the ⟨*token list*⟩ variable. Tokenization occurs as described for `\ior_get:NN` or `\ior_str_get:NN`, respectively. When the ⟨*prompt*⟩ is empty, TEX will wait for input without any other indication: typically the programmer will have provided a suitable text using e.g. `\iow_term:n`. Where the ⟨*prompt*⟩ is given, it will appear in the terminal followed by an `=`, e.g.

```
prompt=
```

1.3 Writing to files

\iow_now:Nn
\iow_now:(NV|Ne|cn|cV|ce)

`\iow_now:Nn ⟨stream⟩ {⟨tokens⟩}`

Updated: 2012-06-05

This function writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ immediately (*i.e.* the write operation is called on expansion of `\iow_now:Nn`).

\iow_log:n
\iow_log:e

`\iow_log:n {⟨tokens⟩}`

This function writes the given ⟨*tokens*⟩ to the log (transcript) file immediately: it is a dedicated version of `\iow_now:Nn`.

\iow_term:n
\iow_term:e

`\iow_term:n {⟨tokens⟩}`

This function writes the given ⟨*tokens*⟩ to the terminal file immediately: it is a dedicated version of `\iow_now:Nn`.

\iow_shipout:Nn
\iow_shipout:(Ne|cn|ce)

`\iow_shipout:Nn ⟨stream⟩ {⟨tokens⟩}`

This function writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ when the current page is finalised (*i.e.* at shipout). The e-type variants expand the ⟨*tokens*⟩ at the point where the function is used but *not* when the resulting tokens are written to the ⟨*stream*⟩ (*cf.* `\iow_shipout_e:Nn`).

TEXhackers note: When using expl3 with a format other than LATEX, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

\iow_shipout_e:Nn
\iow_shipout_e:(Ne|cn|ce)

Updated: 2023-09-17

\iow_shipout_e:Nn ⟨*stream*⟩ {⟨*tokens*⟩}

This function writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ when the current page is finalised (*i.e.* at shipout). The ⟨*tokens*⟩ are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

TEXhackers note: This is a wrapper around the TEX primitive `\write`. When using expl3 with a format other than LATEX, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

\iow_char:N ⋆

\iow_char:N \⟨*char*⟩

Inserts ⟨*char*⟩ into the output stream. Useful when trying to write difficult characters such as `%`, `{`, `}`, *etc.* in messages, for example:

```
\iow_now:Ne \g_my_iow { \iow_char:N \{ text \iow_char:N \} }
```

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

\iow_newline: ⋆

\iow_newline:

Function to add a new line within the ⟨*tokens*⟩ written to a file. The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

TEXhackers note: When using expl3 with a format other than LATEX, the character inserted by `\iow_newline:` is not recognized by TEX, which may lead to the insertion of additional unwanted line-breaks. This issue only affects `\iow_shipout:Nn`, `\iow_shipout_e:Nn` and direct uses of primitive operations.

1.4 Wrapping lines in output

\iow_wrap:nnnN
\iow_wrap:nenN

New: 2012-06-28
Updated: 2017-12-04

\iow_wrap:nnnN {⟨*text*⟩} {⟨*run-on text*⟩} {⟨*set up*⟩} ⟨*function*⟩

This function wraps the ⟨*text*⟩ to a fixed number of characters per line. At the start of each line which is wrapped, the ⟨*run-on text*⟩ is inserted. The line character count targeted is the value of `\l_iow_line_count_int` minus the number of characters in the ⟨*run-on text*⟩ for all lines except the first, for which the target number of characters is simply `\l_iow_line_count_int` since there is no run-on text. The ⟨*text*⟩ and ⟨*run-on text*⟩ are exhaustively expanded by the function, with the following substitutions:

- `\\` or `\iow_newline:` may be used to force a new line,
- `\␣` may be used to represent a forced space (for example after a control sequence),
- `\#`, `\%`, `\{`, `\}`, `\~` may be used to represent the corresponding character,
- `\iow_wrap_allow_break:` may be used to allow a line-break without inserting a space,
- `\iow_indent:n` may be used to indent a part of the ⟨*text*⟩ (not the ⟨*run-on text*⟩).

Additional functions may be added to the wrapping by using the ⟨*set up*⟩, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the ⟨*text*⟩ which is not to be expanded on wrapping should be converted to a string using `\token_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N`, *etc.*

The result of the wrapping operation is passed as a braced argument to the ⟨*function*⟩, which is typically a wrapper around a write operation. The output of `\iow_wrap:nnnN` (*i.e.* the argument passed to the ⟨*function*⟩) consists of characters of category "other" (category code 12), with the exception of spaces which have category "space" (category code 10). This means that the output does *not* expand further when written to a file.

TEXhackers note: Internally, `\iow_wrap:nnnN` carries out an `e`-type expansion on the ⟨*text*⟩ to expand it. This is done in such a way that `\exp_not:N` or `\exp_not:n` *could* be used to prevent expansion of material. However, this is less conceptually clear than conversion to a string, which is therefore the supported method for handling expandable material in the ⟨*text*⟩.

\iow_wrap_allow_break:

\iow_wrap_allow_break:

New: 2023-04-25

In the first argument of `\iow_wrap:nnnN` (for instance in messages), inserts a break-point that allows a line break. If no break occurs, this function adds nothing to the output.

\iow_indent:n

\iow_indent:n {⟨*text*⟩}

New: 2011-09-21

In the first argument of `\iow_wrap:nnnN` (for instance in messages), indents ⟨*text*⟩ by four spaces. This function does not cause a line break, and only affects lines which start within the scope of the ⟨*text*⟩. In case the indented ⟨*text*⟩ should appear on separate lines from the surrounding text, use `\\` to force line breaks.

\l_iow_line_count_int

New: 2012-06-24

The maximum number of characters in a line to be written by the `\iow_wrap:nnnN` function. This value depends on the TEX system in use: the standard value is 78, which is typically correct for unmodified TEX Live and MiKTEX systems.

1.5 Constant input–output streams, and variables

\g_tmpa_ior
\g_tmpb_ior

New: 2017-12-11

Scratch input stream for global use. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\c_log_iow
\c_term_iow

Constant output streams for writing to the log and to the terminal (plus the log), respectively.

\g_tmpa_iow
\g_tmpb_iow

New: 2017-12-11

Scratch output stream for global use. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

1.6 Primitive conditionals

\if_eof:w ⋆

```
\if_eof:w ⟨stream⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests if the ⟨*stream*⟩ returns "end of file", which is true for non-existent files. The `\else:` branch is optional.

TEXhackers note: This is the TEX primitive `\ifeof`.

2 File opertions

2.1 Basic file operations

\g_file_curr_dir_str
\g_file_curr_name_str
\g_file_curr_ext_str

New: 2017-06-21

Contain the directory, name and extension of the current file. The directory is empty if the file was loaded without an explicit path (*i.e.* if it is in the TEX search path), and does *not* end in `/` other than the case that it is exactly equal to the root directory. The ⟨*name*⟩ and ⟨*ext*⟩ parts together make up the file name, thus the ⟨*name*⟩ part may be thought of as the "job name" for the current file.

Note that TEX does not provide information on the ⟨*dir*⟩ and ⟨*ext*⟩ part for the main (top level) file and that this file always has empty ⟨*dir*⟩ and ⟨*ext*⟩ components. Also, the ⟨*name*⟩ here will be equal to `\c_sys_jobname_str`, which may be different from the real file name (if set using `--jobname`, for example).

\l_file_search_path_seq

New: 2017-06-18
Updated: 2023-06-15

Each entry is the path to a directory which should be searched when seeking a file. Each path can be relative or absolute, and need not include the trailing slash. Spaces need not be quoted.

TEXhackers note: When working as a package in LaTeX 2ε, expl3 will automatically append the current `\input@path` to the set of values from `\l_file_search_path_seq`.

\file_if_exist_p:n ⋆
\file_if_exist_p:V ⋆
\file_if_exist:nTF ⋆
\file_if_exist:VTF ⋆

Updated: 2023-09-18

`\file_if_exist_p:n {⟨file name⟩}`
`\file_if_exist:nTF {⟨file name⟩} {⟨true code⟩} {⟨false code⟩}`

Expands the argument of the `\file name` to give a string, then searches for this string using the current TEX search path and the additional paths controlled by `\l_file_search_path_seq`.

2.2 Information about files and file contents

Functions in this section return information about files as expl3 `str` data, *except* that the non-expandable functions set their return *token list* to `\q_no_value` if the file requested is not found. As such, comparison of file names, hashes, sizes, etc., should use `\str_if_eq:nnTF` rather than `\tl_if_eq:nnTF` and so on.

\file_hex_dump:n ☆
\file_hex_dump:V ☆
\file_hex_dump:nnn ☆
\file_hex_dump:Vnn ☆

New: 2019-11-19

`\file_hex_dump:n {⟨file name⟩}`
`\file_hex_dump:nnn {⟨file name⟩} {⟨start index⟩} {⟨end index⟩}`

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by `\l_file_search_path_seq`. It then expands to leave the hexadecimal dump of the file content in the input stream. The file is read as bytes, which means that in contrast to most TEX behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty. The `{`⟨*start index*⟩`}` and `{`⟨*end index*⟩`}` values work as described for `\str_range:nnn`.

\file_get_hex_dump:nN
\file_get_hex_dump:VN
\file_get_hex_dump:nNTF
\file_get_hex_dump:VNTF
\file_get_hex_dump:nnnN
\file_get_hex_dump:VnnN
\file_get_hex_dump:nnnNTF
\file_get_hex_dump:VnnNTF

New: 2019-11-19

`\file_get_hex_dump:nN {⟨file name⟩} ⟨tl var⟩`
`\file_get_hex_dump:nnnN {⟨file name⟩} {⟨start index⟩} {⟨end index⟩} ⟨tl var⟩`

Sets the ⟨*tl var*⟩ to the result of applying `\file_hex_dump:n`/`\file_hex_dump:nnn` to the ⟨*file*⟩. If the file is not found, the ⟨*tl var*⟩ will be set to `\q_no_value`.

\file_mdfive_hash:n ☆
\file_mdfive_hash:V ☆

New: 2019-09-03

\file_mdfive_hash:n {⟨*file name*⟩}

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by \l_file_search_path_seq. It then expands to leave the MD5 sum generated from the contents of the file in the input stream. The file is read as bytes, which means that in contrast to most TEX behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty.

\file_get_mdfive_hash:nN
\file_get_mdfive_hash:VN
\file_get_mdfive_hash:nN*TF*
\file_get_mdfive_hash:VN*TF*

New: 2017-07-11
Updated: 2019-02-16

\file_get_mdfive_hash:nN {⟨*file name*⟩} ⟨*tl var*⟩

Sets the ⟨*tl var*⟩ to the result of applying \file_mdfive_hash:n to the ⟨*file*⟩. If the file is not found, the ⟨*tl var*⟩ will be set to \q_no_value.

\file_size:n ☆
\file_size:V ☆

New: 2019-09-03

\file_size:n {⟨*file name*⟩}

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by \l_file_search_path_seq. It then expands to leave the size of the file in bytes in the input stream. When the file is not found, the result of expansion is empty.

\file_get_size:nN
\file_get_size:VN
\file_get_size:nN*TF*
\file_get_size:VN*TF*

New: 2017-07-09
Updated: 2019-02-16

\file_get_size:nN {⟨*file name*⟩} ⟨*tl var*⟩

Sets the ⟨*tl var*⟩ to the result of applying \file_size:n to the ⟨*file*⟩. If the file is not found, the ⟨*tl var*⟩ will be set to \q_no_value. This is not available in older versions of XeTEX.

\file_timestamp:n ☆
\file_timestamp:V ☆

New: 2019-09-03

\file_timestamp:n {⟨*file name*⟩}

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by \l_file_search_path_seq. It then expands to leave the modification timestamp of the file in the input stream. The timestamp is of the form D:⟨*year*⟩⟨*month*⟩⟨*day*⟩⟨*hour*⟩⟨*minute*⟩⟨*second*⟩⟨*offset*⟩, where the latter may be Z (UTC) or ⟨*plus-minus*⟩⟨*hours*⟩'⟨*minutes*⟩'. When the file is not found, the result of expansion is empty. This is not available in older versions of XeTEX.

\file_get_timestamp:nN
\file_get_timestamp:VN
\file_get_timestamp:nN*TF*
\file_get_timestamp:VN*TF*

New: 2017-07-09
Updated: 2019-02-16

\file_get_timestamp:nN {⟨*file name*⟩} ⟨*tl var*⟩

Sets the ⟨*tl var*⟩ to the result of applying \file_timestamp:n to the ⟨*file*⟩. If the file is not found, the ⟨*tl var*⟩ will be set to \q_no_value. This is not available in older versions of XeTEX.

\file_compare_timestamp_p:nNn ⋆ \file_compare_timestamp_p:nNn {⟨*file-1*⟩} ⟨*comparator*⟩
\file_compare_timestamp_p:(nNV|VNn|VNV) ⋆ {⟨*file-2*⟩}
\file_compare_timestamp:nNnTF ⋆ \file_compare_timestamp:nNnTF {⟨*file-1*⟩} ⟨*comparator*⟩
\file_compare_timestamp:(nNV|VNn|VNV)TF ⋆ {⟨*file-2*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

New: 2019-05-13
Updated: 2019-09-20

Compares the file stamps on the two ⟨*files*⟩ as indicated by the ⟨*comparator*⟩, and inserts either the ⟨*true code*⟩ or ⟨*false case*⟩ as required. A file which is not found is treated as older than any file which is found. This allows for example the construct

```
\file_compare_timestamp:nNnT { source-file } > { derived-file }
  {
    % Code to regenerate derived file
  }
```

to work when the derived file is entirely absent. The timestamp of two absent files is regarded as different. This is not available in older versions of XeTeX.

\file_get_full_name:nN \file_get_full_name:nN {⟨*file name*⟩} ⟨*tl*⟩
\file_get_full_name:VN \file_get_full_name:nNTF {⟨*file name*⟩} ⟨*tl*⟩ {⟨*true code*⟩} {⟨*false code*⟩}
\file_get_full_name:nNTF
\file_get_full_name:VNTF

Updated: 2019-02-16

Searches for ⟨*file name*⟩ in the path as detailed for \file_if_exist:nTF, and if found sets the ⟨*tl var*⟩ the fully-qualified name of the file, *i.e.* the path and file name. This includes an extension .tex when the given ⟨*file name*⟩ has no extension but the file found has that extension. In the non-branching version, the ⟨*tl var*⟩ will be set to \q_no_value in the case that the file does not exist.

\file_full_name:n ☆ \file_full_name:n {⟨*file name*⟩}
\file_full_name:V ☆

New: 2019-09-03

Searches for ⟨*file name*⟩ in the path as detailed for \file_if_exist:nTF, and if found leaves the fully-qualified name of the file, *i.e.* the path and file name, in the input stream. This includes an extension .tex when the given ⟨*file name*⟩ has no extension but the file found has that extension. If the file is not found on the path, the expansion is empty.

\file_parse_full_name:nNNN \file_parse_full_name:nNNN {⟨*full name*⟩} ⟨*dir*⟩ ⟨*name*⟩ ⟨*ext*⟩
\file_parse_full_name:VNNN

New: 2017-06-23
Updated: 2020-06-24

Parses the ⟨*full name*⟩ and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- The ⟨*dir*⟩: everything up to the last / (path separator) in the ⟨*file path*⟩. As with system PATH variables and related functions, the ⟨*dir*⟩ does *not* include the trailing / unless it points to the root directory. If there is no path (only a file name), ⟨*dir*⟩ is empty.
- The ⟨*name*⟩: everything after the last / up to the last ., where both of those characters are optional. The ⟨*name*⟩ may contain multiple . characters. It is empty if ⟨*full name*⟩ consists only of a directory name.
- The ⟨*ext*⟩: everything after the last . (including the dot). The ⟨*ext*⟩ is empty if there is no . after the last /.

Before parsing, the ⟨*full name*⟩ is expanded until only non-expandable tokens remain, except that active characters are also not expanded. Quotes (") are invalid in file names and are discarded from the input.

`\file_parse_full_name:n` ⋆ `\file_parse_full_name:n {`⟨*full name*⟩`}`
`\file_parse_full_name:V` ⋆

Parses the ⟨*full name*⟩ as described for `\file_parse_full_name:nNNN`, and leaves ⟨*dir*⟩, ⟨*name*⟩, and ⟨*ext*⟩ in the input stream, each inside a pair of braces.

New: 2020-06-24

`\file_parse_full_name_apply:nN` ⋆ `\file_parse_full_name_apply:nN {`⟨*full name*⟩`}` ⟨*function*⟩
`\file_parse_full_name_apply:VN` ⋆

New: 2020-06-24

Parses the ⟨*full name*⟩ as described for `\file_parse_full_name:nNNN`, and passes ⟨*dir*⟩, ⟨*name*⟩, and ⟨*ext*⟩ as arguments to ⟨*function*⟩, as an `n`-type argument each, in this order.

2.3 Accessing file contents

`\file_get:nnN` `\file_get:nnN {`⟨*file name*⟩`}` `{`⟨*setup*⟩`}` ⟨*tl*⟩
`\file_get:VnN` `\file_get:nnNTF {`⟨*file name*⟩`}` `{`⟨*setup*⟩`}` ⟨*tl*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`
`\file_get:nnNTF`
`\file_get:VnNTF`

Defines ⟨*tl*⟩ to the contents of ⟨*file name*⟩. Category codes may need to be set appropriately via the ⟨*setup*⟩ argument. The non-branching version sets the ⟨*tl*⟩ to `\q_no_value` if the file is not found. The branching version runs the ⟨*true code*⟩ after the assignment to ⟨*tl*⟩ if the file is found, and ⟨*false code*⟩ otherwise. The file content will be tokenized using the current category code régime,

New: 2019-01-16
Updated: 2019-02-16

`\file_input:n` `\file_input:n {`⟨*file name*⟩`}`
`\file_input:V`

Searches for ⟨*file name*⟩ in the path as detailed for `\file_if_exist:nTF`, and if found reads in the file as additional LaTeX source. All files read are recorded for information and the file name stack is updated by this function. An error is raised if the file is not found.

Updated: 2017-06-26

`\file_input_raw:n` ⋆ `\file_input_raw:n {`⟨*file name*⟩`}`
`\file_input_raw:V` ⋆

Searches for ⟨*file name*⟩ in the path as detailed for `\file_if_exist:nTF`, and if found reads in the file as additional TeX source. No data concerning the file is tracked. If the file is not found, no action is taken.

New: 2023-05-18

TeXhackers note: This function is intended only for contexts where files must be read purely by expansion, for example at the start of a table cell in an `\halign`.

`\file_if_exist_input:n` `\file_if_exist_input:n {`⟨*file name*⟩`}`
`\file_if_exist_input:V` `\file_if_exist_input:nF {`⟨*file name*⟩`}` `{`⟨*false code*⟩`}`
`\file_if_exist_input:nF`
`\file_if_exist_input:VF`

Searches for ⟨*file name*⟩ using the current TeX search path and the additional paths included in `\l_file_search_path_seq`. If found then reads in the file as additional LaTeX source as described for `\file_input:n`, otherwise inserts the ⟨*false code*⟩. Note that these functions do not raise an error if the file is not found, in contrast to `\file_input:n`.

New: 2014-07-02

\file_input_stop: \file_input_stop:

New: 2017-07-07 Ends the reading of a file started by \file_input:n or similar before the end of the file is reached. Where the file reading is being terminated due to an error, \msg_critical:nn(nn) should be preferred.

TEXhackers note: This function must be used on a line on its own: TEX reads files line-by-line and so any additional tokens in the "current" line will still be read.

This is also true if the function is hidden inside another function (which will be the normal case), i.e., all tokens on the same line in the source file are still processed. Putting it on a line by itself in the definition doesn't help as it is the line where it is used that counts!

\file_show_list: \file_show_list:
\file_log_list: \file_log_list:

These functions list all files loaded by LATEX 2ε commands that populate \@filelist or by \file_input:n. While \file_show_list: displays the list in the terminal, \file_log_list: outputs it to the log file only.

3 l3file implementation

The following test files are used for this code: m3file001.

```
1 ⟨*package⟩
```

3.1 Input operations

```
2 ⟨@@=ior⟩
```

3.1.1 Variables and constants

\l__ior_internal_tl Used as a short-term scratch variable.

```
3 \tl_new:N  \l__ior_internal_tl
```

(*End of definition for* \l__ior_internal_tl.)

\c__ior_term_ior Reading from the terminal (with a prompt) is done using a positive but non-existent stream number. Unlike writing, there is no concept of reading from the log.

```
4 \int_const:Nn \c__ior_term_ior { 16 }
```

(*End of definition for* \c__ior_term_ior.)

\g__ior_streams_seq A list of the currently-available input streams to be used as a stack.

```
5 \seq_new:N \g__ior_streams_seq
```

(*End of definition for* \g__ior_streams_seq.)

\l__ior_stream_tl Used to recover the raw stream number from the stack.

```
6 \tl_new:N \l__ior_stream_tl
```

(*End of definition for* \l__ior_stream_tl.)

\g__ior_streams_prop The name of the file attached to each stream is tracked in a property list. To get the correct number of reserved streams in package mode the underlying mechanism needs to be queried. For LaTeX 2ε and plain TeX this data is stored in `\count16`: with the etex package loaded we need to subtract 1 as the register holds the number of the next stream to use. In ConTeXt, we need to look at `\count38` but there is no subtraction: like the original plain TeX/LaTeX 2ε mechanism it holds the value of the *last* stream allocated.

```
\prop_new:N \g__ior_streams_prop
\int_step_inline:nnn
  { 0 }
  {
    \cs_if_exist:NTF \contextversion
      { \tex_count:D 38 ~ }
      {
        \tex_count:D 16 ~ %
        \cs_if_exist:NT \loccount { - 1 }
      }
  }
  {
    \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved~by~format }
  }
```

(*End of definition for* `\g__ior_streams_prop`.)

3.1.2 Stream management

\ior_new:N
\ior_new:c Reserving a new stream is done by defining the name as equal to using the terminal.

```
\cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c__ior_term_ior }
\cs_generate_variant:Nn \ior_new:N { c }
```

(*End of definition for* `\ior_new:N`. *This function is documented on page* 1.)

\g_tmpa_ior
\g_tmpb_ior The usual scratch space.

```
\ior_new:N \g_tmpa_ior
\ior_new:N \g_tmpb_ior
```

(*End of definition for* `\g_tmpa_ior` *and* `\g_tmpb_ior`. *These variables are documented on page* 9.)

\ior_open:Nn
\ior_open:cn Use the conditional version, with an error if the file is not found.

```
\cs_new_protected:Npn \ior_open:Nn #1#2
  { \ior_open:NnF #1 {#2} { \__kernel_file_missing:n {#2} } }
\cs_generate_variant:Nn \ior_open:Nn { c }
```

(*End of definition for* `\ior_open:Nn`. *This function is documented on page* 2.)

\l__ior_file_name_tl Data storage.

```
\tl_new:N \l__ior_file_name_tl
```

(*End of definition for* `\l__ior_file_name_tl`.)

\ior_open:NnTF
\ior_open:cnTF

An auxiliary searches for the file in the TEX, LATEX 2ε and LATEX3 paths. Then pass the file found to the lower-level function which deals with streams. The `full_name` is empty when the file is not found.

```
\prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
  {
    \file_get_full_name:nNTF {#2} \l__ior_file_name_tl
      {
        \__kernel_ior_open:No #1 \l__ior_file_name_tl
        \prg_return_true:
      }
      { \prg_return_false: }
  }
\prg_generate_conditional_variant:Nnn \ior_open:Nn { c } { T , F , TF }
```

(*End of definition for* `\ior_open:NnTF`. *This function is documented on page* 2.)

__ior_new:N

Streams are reserved using `\newread` before they can be managed by ior. To prevent ior from being affected by redefinitions of `\newread` (such as done by the third-party package morewrites), this macro is saved here under a private name. The complicated code ensures that `\__ior_new:N` is not `\outer` despite plain TEX's `\newread` being `\outer`. For ConTEXt, we have to deal with the fact that `\newread` works like our own: it actually checks before altering definition.

```
\exp_args:NNf \cs_new_protected:Npn \__ior_new:N
  { \exp_args:NNc \exp_after:wN \exp_stop_f: { newread } }
\cs_if_exist:NT \contextversion
  {
    \cs_new_eq:NN \__ior_new_aux:N \__ior_new:N
    \cs_gset_protected:Npn \__ior_new:N #1
      {
        \cs_undefine:N #1
        \__ior_new_aux:N #1
      }
  }
```

(*End of definition for* `\__ior_new:N`.)

__kernel_ior_open:Nn
__kernel_ior_open:No
__ior_open_stream:Nn

The stream allocation itself uses the fact that there is a list of all of those available. Life gets more complex as it's important to keep things in sync. That is done using a two-part approach: any streams that have already been taken up by ior but are now free are tracked, so we first try those. If that fails, ask plain TEX or LATEX 2ε for a new stream and use that number (after a bit of conversion).

```
\cs_new_protected:Npn \__kernel_ior_open:Nn #1#2
  {
    \ior_close:N #1
    \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
      { \__ior_open_stream:Nn #1 {#2} }
      {
        \__ior_new:N #1
        \__kernel_tl_set:Ne \l__ior_stream_tl { \int_eval:n {#1} }
        \__ior_open_stream:Nn #1 {#2}
      }
  }
\cs_generate_variant:Nn \__kernel_ior_open:Nn { No }
```

Here, we act defensively in case LuaTEX is in use with an extensionless file name.

```
\cs_new_protected:Npe \__ior_open_stream:Nn #1#2
  {
    \tex_global:D \tex_chardef:D #1 = \exp_not:N \l__ior_stream_tl \scan_stop:
    \prop_gput:NVn \exp_not:N \g__ior_streams_prop #1 {#2}
    \tex_openin:D #1
      \sys_if_engine_luatex:TF
        { {#2} }
        {  \exp_not:N \__kernel_file_name_quote:n {#2} \scan_stop: }
  }
```

(*End of definition for* `\__kernel_ior_open:Nn` *and* `\__ior_open_stream:Nn`.)

`\ior_shell_open:Nn`
`\__ior_shell_open:nN`
`\__ior_shell_open:oN`

Actually much easier than either the standard open or input versions! When calling `\__kernel_ior_open:Nn` the file the pipe is added to signal a shell command, but the quotes are not added yet—they are added later by `\__kernel_file_name_quote:n`.

```
\cs_new_protected:Npn \ior_shell_open:Nn #1#2
  {
    \sys_if_shell:TF
      { \__ior_shell_open:oN { \tl_to_str:n {#2} } #1 }
      { \msg_error:nn { kernel } { pipe-failed } }
  }
\cs_new_protected:Npn \__ior_shell_open:nN #1#2
  {
    \tl_if_in:nnTF {#1} { " }
      {
        \msg_error:nne
          { kernel } { quote-in-shell } {#1}
      }
      { \__kernel_ior_open:Nn #2 { |#1 } }
  }
\cs_generate_variant:Nn \__ior_shell_open:nN { o }
\msg_new:nnnn { kernel } { pipe-failed }
  { Cannot~run~piped~system~commands. }
  {
    LaTeX~tried~to~call~a~system~process~but~this~was~not~possible.\\
    Try~the~"--shell-escape"~(or~"--enable-pipes")~option.
  }
```

(*End of definition for* `\ior_shell_open:Nn` *and* `\__ior_shell_open:nN`. *This function is documented on page 2.*)

`\ior_close:N`
`\ior_close:c`

Closing a stream means getting rid of it at the TEX level and removing from the various data structures. Unless the name passed is an invalid stream number (outside the range $[0, 15]$), it can be closed. On the other hand, it only gets added to the stack if it was not already there, to avoid duplicates building up.

```
\cs_new_protected:Npn \ior_close:N #1
  {
    \int_compare:nT { -1 < #1 < \c__ior_term_ior }
      {
        \tex_closein:D #1
        \prop_gremove:NV \g__ior_streams_prop #1
        \seq_if_in:NVF \g__ior_streams_seq #1
          { \seq_gpush:NV \g__ior_streams_seq #1 }
```

```
        \cs_gset_eq:NN #1 \c__ior_term_ior
      }
  }
\cs_generate_variant:Nn \ior_close:N { c }
```

(*End of definition for* `\ior_close:N`. *This function is documented on page* 2.)

`\ior_show:N` `\ior_log:N` `\__ior_show:NN`

Seek the stream in the `\g__ior_streams_prop` list, then show the stream as open or closed accordingly.

```
\cs_new_protected:Npn \ior_show:N { \__ior_show:NN \tl_show:n }
\cs_generate_variant:Nn \ior_show:N { c }
\cs_new_protected:Npn \ior_log:N { \__ior_show:NN \tl_log:n }
\cs_generate_variant:Nn \ior_log:N { c }
\cs_new_protected:Npn \__ior_show:NN #1#2
  {
    \__kernel_chk_defined:NT #2
      {
        \prop_get:NVNTF \g__ior_streams_prop #2 \l__ior_internal_tl
          {
            \exp_args:Ne #1
              { \token_to_str:N #2 ~ open: ~ \l__ior_internal_tl }
          }
          { \exp_args:Ne #1 { \token_to_str:N #2 ~ closed } }
      }
  }
```

(*End of definition for* `\ior_show:N`, `\ior_log:N`, *and* `\__ior_show:NN`. *These functions are documented on page* 3.)

`\ior_show_list:` `\ior_log_list:` `\__ior_list:N`

Show the property lists, but with some "pretty printing". See the l3msg module. The first argument of the message is `ior` (as opposed to `iow`) and the second is empty if no read stream is open and non-empty (the list of streams formatted using `\msg_show_item_unbraced:nn`) otherwise. The code of the message `show-streams` takes care of translating `ior`/`iow` to English.

```
\cs_new_protected:Npn \ior_show_list: { \__ior_list:N \msg_show:nneeee }
\cs_new_protected:Npn \ior_log_list: { \__ior_list:N \msg_log:nneeee }
\cs_new_protected:Npn \__ior_list:N #1
  {
    #1 { kernel } { show-streams }
      { ior }
      {
        \prop_map_function:NN \g__ior_streams_prop
          \msg_show_item_unbraced:nn
      }
      { } { }
  }
```

(*End of definition for* `\ior_show_list:`, `\ior_log_list:`, *and* `\__ior_list:N`. *These functions are documented on page* 3.)

3.1.3 Reading input

`\if_eof:w`

The primitive conditional

```
\cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

(*End of definition for* `\if_eof:w`. *This function is documented on page 9.*)

`\ior_if_eof_p:N`
`\ior_if_eof:NTF`

To test if some particular input stream is exhausted the following conditional is provided. The primitive test can only deal with numbers in the range $[0, 15]$ so we catch outliers (they are exhausted).

```
\prg_new_conditional:Npnn \ior_if_eof:N #1 { p , T , F , TF }
  {
    \if_int_compare:w -1 < #1
      \if_int_compare:w #1 < \c__ior_term_ior
        \if_eof:w #1
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      \else:
        \prg_return_true:
      \fi:
    \else:
      \prg_return_true:
    \fi:
  }
```

(*End of definition for* `\ior_if_eof:NTF`. *This function is documented on page 6.*)

`\ior_get:NN`
`\__ior_get:NN`
`\ior_get:NNTF`

And here we read from files.

```
\cs_new_protected:Npn \ior_get:NN #1#2
  { \ior_get:NNF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_new_protected:Npn \__ior_get:NN #1#2
  { \tex_read:D #1 to #2 }
\prg_new_protected_conditional:Npnn \ior_get:NN #1#2 { T , F , TF }
  {
    \ior_if_eof:NTF #1
      { \prg_return_false: }
      {
        \__ior_get:NN #1 #2
        \prg_return_true:
      }
  }
```

(*End of definition for* `\ior_get:NN`, `\__ior_get:NN`, *and* `\ior_get:NNTF`. *These functions are documented on page 3.*)

`\ior_str_get:NN`
`\__ior_str_get:NN`
`\ior_str_get:NNTF`

Reading as strings is a more complicated wrapper, as we wish to remove the endline character and restore it afterwards.

```
\cs_new_protected:Npn \ior_str_get:NN #1#2
  { \ior_str_get:NNF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_new_protected:Npn \__ior_str_get:NN #1#2
  {
    \exp_args:Nno \use:n
      {
        \int_set:Nn \tex_endlinechar:D { -1 }
        \tex_readline:D #1 to #2
        \int_set:Nn \tex_endlinechar:D
      }   { \int_use:N \tex_endlinechar:D }
```

```
  }
\prg_new_protected_conditional:Npnn \ior_str_get:NN #1#2 { T , F , TF }
  {
    \ior_if_eof:NTF #1
      { \prg_return_false: }
      {
        \__ior_str_get:NN #1 #2
        \prg_return_true:
      }
  }
```

(*End of definition for* `\ior_str_get:NN`, `\__ior_str_get:NN`, *and* `\ior_str_get:NNTF`. *These functions are documented on page* 4.)

`\c__ior_term_noprompt_ior` For reading without a prompt.

```
\int_const:Nn \c__ior_term_noprompt_ior { -1 }
```

(*End of definition for* `\c__ior_term_noprompt_ior`.)

`\ior_get_term:nN` `\ior_str_get_term:nN` `\__ior_get_term:NnN` Getting from the terminal is better with pretty-printing.

```
\cs_new_protected:Npn \ior_get_term:nN #1#2
  { \__ior_get_term:NnN \__ior_get:NN {#1} #2 }
\cs_new_protected:Npn \ior_str_get_term:nN #1#2
  { \__ior_get_term:NnN \__ior_str_get:NN {#1} #2 }
\cs_new_protected:Npn \__ior_get_term:NnN #1#2#3
  {
    \group_begin:
      \tex_escapechar:D = -1 \scan_stop:
      \tl_if_blank:nTF {#2}
        { \exp_args:NNc #1 \c__ior_term_noprompt_ior }
        { \exp_args:NNc #1 \c__ior_term_ior }
          {#2}
    \exp_args:NNNv \group_end:
    \tl_set:Nn #3 {#2}
  }
```

(*End of definition for* `\ior_get_term:nN`, `\ior_str_get_term:nN`, *and* `\__ior_get_term:NnN`. *These functions are documented on page* 6.)

`\ior_map_break:` `\ior_map_break:n` Usual map breaking functions.

```
\cs_new:Npn \ior_map_break:
  { \prg_map_break:Nn \ior_map_break: { } }
\cs_new:Npn \ior_map_break:n
  { \prg_map_break:Nn \ior_map_break: }
```

(*End of definition for* `\ior_map_break:` *and* `\ior_map_break:n`. *These functions are documented on page* 5.)

`\ior_map_inline:Nn` `\ior_str_map_inline:Nn` `\__ior_map_inline:NNn` `\__ior_map_inline:NNNn` `\__ior_map_inline_loop:NNN` Mapping over an input stream can be done on either a token or a string basis, hence the set up. Within that, there is a check to avoid reading past the end of a file, hence the two applications of `\ior_if_eof:N` and its lower-level analogue `\if_eof:w`. This mapping cannot be nested with twice the same stream, as the stream has only one "current line".

```
\cs_new_protected:Npn \ior_map_inline:Nn
  { \__ior_map_inline:NNn \__ior_get:NN }
\cs_new_protected:Npn \ior_str_map_inline:Nn
```

```
  { \__ior_map_inline:NNn \__ior_str_get:NN }
\cs_new_protected:Npn \__ior_map_inline:NNn
  {
    \int_gincr:N \g__kernel_prg_map_int
    \exp_args:Nc \__ior_map_inline:NNNn
      { __ior_map_ \int_use:N \g__kernel_prg_map_int :n }
  }
\cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
  {
    \cs_gset_protected:Npn #1 ##1 {#4}
    \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
    \prg_break_point:Nn \ior_map_break:
      { \int_gdecr:N \g__kernel_prg_map_int }
  }
\cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
  {
    #2 #3 \l__ior_internal_tl
    \if_eof:w #3
      \exp_after:wN \ior_map_break:
    \fi:
    \exp_args:No #1 \l__ior_internal_tl
    \__ior_map_inline_loop:NNN #1#2#3
  }
```

(*End of definition for* `\ior_map_inline:Nn` *and others. These functions are documented on page 4.*)

`\ior_map_variable:NNn`
`\ior_str_map_variable:NNn`
`\__ior_map_variable:NNNn`
`\__ior_map_variable_loop:NNNn`

Since the TEX primitive (`\read` or `\readline`) assigns the tokens read in the same way as a token list assignment, we simply call the appropriate primitive. The end-of-loop is checked using the primitive conditional for speed.

```
\cs_new_protected:Npn \ior_map_variable:NNn
  { \__ior_map_variable:NNNn \ior_get:NN }
\cs_new_protected:Npn \ior_str_map_variable:NNn
  { \__ior_map_variable:NNNn \ior_str_get:NN }
\cs_new_protected:Npn \__ior_map_variable:NNNn #1#2#3#4
  {
    \ior_if_eof:NF #2 { \__ior_map_variable_loop:NNNn #1#2#3 {#4} }
    \prg_break_point:Nn \ior_map_break: { }
  }
\cs_new_protected:Npn \__ior_map_variable_loop:NNNn #1#2#3#4
  {
    #1 #2 #3
    \if_eof:w #2
      \exp_after:wN \ior_map_break:
    \fi:
    #4
    \__ior_map_variable_loop:NNNn #1#2#3 {#4}
  }
```

(*End of definition for* `\ior_map_variable:NNn` *and others. These functions are documented on page 4.*)

3.2 Output operations

⟨@@=iow⟩

There is a lot of similarity here to the input operations, at least for many of the basics. Thus quite a bit is copied from the earlier material with minor alterations.

3.2.1 Variables and constants

\l__iow_internal_tl Used as a short-term scratch variable.

```
\tl_new:N  \l__iow_internal_tl
```

(*End of definition for* `\l__iow_internal_tl`.)

\c_log_iow
\c_term_iow Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`) and to both the terminal and transcript file (`\c_term_iow`). Recent LuaTEX provide 128 write streams; we also use `\c_term_iow` as the first non-allowed write stream so its value depends on the engine.

```
\int_const:Nn \c_log_iow  { -1 }
\int_const:Nn \c_term_iow
  {
    \bool_lazy_and:nnTF
      { \sys_if_engine_luatex_p: }
      { \int_compare_p:nNn \tex_luatexversion:D > { 80 } }
      { 128 }
      { 16 }
  }
```

(*End of definition for* `\c_log_iow` *and* `\c_term_iow`. *These variables are documented on page* 9.)

\g__iow_streams_seq A list of the currently-available output streams to be used as a stack.

```
\seq_new:N \g__iow_streams_seq
```

(*End of definition for* `\g__iow_streams_seq`.)

\l__iow_stream_tl Used to recover the raw stream number from the stack.

```
\tl_new:N \l__iow_stream_tl
```

(*End of definition for* `\l__iow_stream_tl`.)

\g__iow_streams_prop As for reads with the appropriate adjustment of the register numbers to check on.

```
\prop_new:N \g__iow_streams_prop
\int_step_inline:nnn
  { 0 }
  {
    \cs_if_exist:NTF \contextversion
      { \tex_count:D 39 ~ }
      {
        \tex_count:D 17 ~
        \cs_if_exist:NT \loccount { - 1 }
      }
  }
  {
    \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved~by~format }
  }
```

(*End of definition for* `\g__iow_streams_prop`.)

3.2.2 Internal auxiliaries

\s__iow_mark
\s__iow_stop

Internal scan marks.

```
\scan_new:N \s__iow_mark
\scan_new:N \s__iow_stop
```

(*End of definition for* `\s__iow_mark` *and* `\s__iow_stop`.)

__iow_use_i_delimit_by_s_stop:nw

Functions to gobble up to a scan mark.

```
\cs_new:Npn \__iow_use_i_delimit_by_s_stop:nw #1 #2 \s__iow_stop {#1}
```

(*End of definition for* `\__iow_use_i_delimit_by_s_stop:nw`.)

\q__iow_nil

Internal quarks.

```
\quark_new:N \q__iow_nil
```

(*End of definition for* `\q__iow_nil`.)

3.3 Stream management

\iow_new:N
\iow_new:c

Reserving a new stream is done by defining the name as equal to writing to the terminal: odd but at least consistent.

```
\cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
\cs_generate_variant:Nn \iow_new:N { c }
```

(*End of definition for* `\iow_new:N`. *This function is documented on page* 1.)

\g_tmpa_iow
\g_tmpb_iow

The usual scratch space.

```
\iow_new:N \g_tmpa_iow
\iow_new:N \g_tmpb_iow
```

(*End of definition for* `\g_tmpa_iow` *and* `\g_tmpb_iow`. *These variables are documented on page* 9.)

__iow_new:N

As for read streams, copy `\newwrite`, making sure that it is not `\outer`. For ConTEXt, we have to deal with the fact that `\newwrite` works like our own: it actually checks before altering definition.

```
\exp_args:NNf \cs_new_protected:Npn \__iow_new:N
  { \exp_args:NNc \exp_after:wN \exp_stop_f: { newwrite } }
\cs_if_exist:NT \contextversion
  {
    \cs_new_eq:NN \__iow_new_aux:N \__iow_new:N
    \cs_gset_protected:Npn \__iow_new:N #1
      {
        \cs_undefine:N #1
        \__iow_new_aux:N #1
      }
  }
```

(*End of definition for* `\__iow_new:N`.)

\l__iow_file_name_tl

Data storage.

```
\tl_new:N \l__iow_file_name_tl
```

(*End of definition for* `\l__iow_file_name_tl`.)

\iow_open:Nn
\iow_open:NV
\iow_open:cn
\iow_open:cV
__iow_open_stream:Nn
__iow_open_stream:NV

The same idea as for reading, but without the path and without the need to allow for a conditional version.

```
\cs_new_protected:Npn \iow_open:Nn #1#2
  {
    \__kernel_tl_set:Ne \l__iow_file_name_tl
      { \__kernel_file_name_sanitize:n {#2} }
    \iow_close:N #1
    \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
      { \__iow_open_stream:NV #1 \l__iow_file_name_tl }
      {
        \__iow_new:N #1
        \__kernel_tl_set:Ne \l__iow_stream_tl { \int_eval:n {#1} }
        \__iow_open_stream:NV #1 \l__iow_file_name_tl
      }
  }
\cs_generate_variant:Nn \iow_open:Nn { NV , c , cV }
\cs_new_protected:Npn \__iow_open_stream:Nn #1#2
  {
    \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
    \prop_gput:NVn \g__iow_streams_prop #1 {#2}
    \tex_immediate:D \tex_openout:D
        #1 \__kernel_file_name_quote:n {#2} \scan_stop:
  }
\cs_generate_variant:Nn \__iow_open_stream:Nn { NV }
```

(*End of definition for* `\iow_open:Nn` *and* `\__iow_open_stream:Nn`*. This function is documented on page 2.*)

\iow_shell_open:Nn
__iow_shell_open:nN
__iow_shell_open:oN

Very similar to the `ior` version

```
\cs_new_protected:Npn \iow_shell_open:Nn #1#2
  {
    \sys_if_shell:TF
      { \__iow_shell_open:oN { \tl_to_str:n {#2} } #1 }
      { \msg_error:nn { kernel } { pipe-failed } }
  }
\cs_new_protected:Npn \__iow_shell_open:nN #1#2
  {
    \tl_if_in:nnTF {#1} { " }
      {
        \msg_error:nne
          { kernel } { quote-in-shell } {#1}
      }
      { \__kernel_iow_open:Nn #2 { |#1 } }
  }
\cs_generate_variant:Nn \__iow_shell_open:nN { o }
```

(*End of definition for* `\iow_shell_open:Nn` *and* `\__iow_shell_open:nN`*. This function is documented on page 2.*)

\iow_close:N
\iow_close:c

Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

```
\cs_new_protected:Npn \iow_close:N #1
  {
```

```
    \int_compare:nT { \c_log_iow < #1 < \c_term_iow }
      {
        \tex_immediate:D \tex_closeout:D #1
        \prop_gremove:NV \g__iow_streams_prop #1
        \seq_if_in:NVF \g__iow_streams_seq #1
          { \seq_gpush:NV \g__iow_streams_seq #1 }
        \cs_gset_eq:NN #1 \c_term_iow
      }
  }
\cs_generate_variant:Nn \iow_close:N { c }
```

(*End of definition for* `\iow_close:N`*. This function is documented on page 2.*)

`\iow_show:N` `\iow_log:N` `\__iow_show:NN`

Seek the stream in the `\g__iow_streams_prop` list, then show the stream as open or closed accordingly.

```
\cs_new_protected:Npn \iow_show:N { \__iow_show:NN \tl_show:n }
\cs_generate_variant:Nn \iow_show:N { c }
\cs_new_protected:Npn \iow_log:N { \__iow_show:NN \tl_log:n }
\cs_generate_variant:Nn \iow_log:N { c }
\cs_new_protected:Npn \__iow_show:NN #1#2
  {
    \__kernel_chk_defined:NT #2
      {
        \prop_get:NVNTF \g__iow_streams_prop #2 \l__iow_internal_tl
          {
            \exp_args:Ne #1
              { \token_to_str:N #2 ~ open: ~ \l__iow_internal_tl }
          }
          { \exp_args:Ne #1 { \token_to_str:N #2 ~ closed } }
      }
  }
```

(*End of definition for* `\iow_show:N`*,* `\iow_log:N`*, and* `\__iow_show:NN`*. These functions are documented on page 3.*)

`\iow_show_list:` `\iow_log_list:` `\__iow_list:N`

Done as for input, but with a copy of the auxiliary so the name is correct.

```
\cs_new_protected:Npn \iow_show_list: { \__iow_list:N \msg_show:nneeee }
\cs_new_protected:Npn \iow_log_list: { \__iow_list:N \msg_log:nneeee }
\cs_new_protected:Npn \__iow_list:N #1
  {
    #1 { kernel } { show-streams }
      { iow }
      {
        \prop_map_function:NN \g__iow_streams_prop
          \msg_show_item_unbraced:nn
      }
      { } { }
  }
```

(*End of definition for* `\iow_show_list:`*,* `\iow_log_list:`*, and* `\__iow_list:N`*. These functions are documented on page 3.*)

3.3.1 Deferred writing

\iow_shipout_e:Nn
\iow_shipout_e:Ne
\iow_shipout_e:cn
\iow_shipout_e:ce

First the easy part, this is the primitive, which expects its argument to be braced.

```
\cs_new_protected:Npn \iow_shipout_e:Nn #1#2
  { \tex_write:D #1 {#2} }
\cs_generate_variant:Nn \iow_shipout_e:Nn { Ne , c, ce }
```

(*End of definition for* `\iow_shipout_e:Nn`*. This function is documented on page* 7.)

\iow_shipout:Nn
\iow_shipout:Ne
\iow_shipout:Nx
\iow_shipout:cn
\iow_shipout:ce
\iow_shipout:cx

With ε-TEX available deferred writing without expansion is easy.

```
\cs_new_protected:Npn \iow_shipout:Nn #1#2
  { \tex_write:D #1 { \exp_not:n {#2} } }
\cs_generate_variant:Nn \iow_shipout:Nn { Ne , c, ce }
\cs_generate_variant:Nn \iow_shipout:Nn { Nx , cx }
```

(*End of definition for* `\iow_shipout:Nn`*. This function is documented on page* 6.)

3.3.2 Immediate writing

__kernel_iow_with:Nnn
__iow_with:nNnn
__iow_with:oNnn

If the integer `#1` is equal to `#2`, just leave `#3` in the input stream. Otherwise, pass the old value to an auxiliary, which sets the integer to the new value, runs the code, and restores the integer.

```
\cs_new_protected:Npn \__kernel_iow_with:Nnn #1#2
  {
    \int_compare:nNnTF {#1} = {#2}
      { \use:n }
      { \__iow_with:oNnn { \int_use:N #1 } #1 {#2} }
  }
\cs_new_protected:Npn \__iow_with:nNnn #1#2#3#4
  {
    \int_set:Nn #2 {#3}
    #4
    \int_set:Nn #2 {#1}
  }
\cs_generate_variant:Nn \__iow_with:nNnn { o }
```

(*End of definition for* `\__kernel_iow_with:Nnn` *and* `\__iow_with:nNnn`.)

\iow_now:Nn
\iow_now:NV
\iow_now:Ne
\iow_now:Nx
\iow_now:cn
\iow_now:cV
\iow_now:ce
\iow_now:cx

This routine writes the second argument onto the output stream without expansion. If this stream isn't open, the output goes to the terminal instead. If the first argument is no output stream at all, we get an internal error. We don't use the expansion done by `\write` to get the `Nx` variant, because it differs in subtle ways from `x`-expansion, namely, macro parameter characters would not need to be doubled. We set the `\newlinechar` to 10 using `\__kernel_iow_with:Nnn` to support formats such as plain TEX: otherwise, `\iow_newline:` would not work. We do not do this for `\iow_shipout:Nn` or `\iow_shipout_x:Nn`, as TEX looks at the value of the `\newlinechar` at shipout time in those cases.

```
\cs_new_protected:Npn \iow_now:Nn #1#2
  {
    \__kernel_iow_with:Nnn \tex_newlinechar:D { `\^^J }
      { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
  }
\cs_generate_variant:Nn \iow_now:Nn { NV , Ne , c , cV , ce }
\cs_generate_variant:Nn \iow_now:Nn { Nx , cx }
```

(*End of definition for* `\iow_now:Nn`. *This function is documented on page 6.*)

`\iow_log:n` Writing to the log and the terminal directly are relatively easy; as we need the two `e`-type
`\iow_log:e` variants for bootstrapping, they are redefinitions here.
`\iow_log:x`
`\iow_term:n`
`\iow_term:e`
`\iow_term:x`

```
\cs_new_protected:Npn \iow_log:n  { \iow_now:Nn \c_log_iow  }
\cs_set_protected:Npn \iow_log:e  { \iow_now:Ne \c_log_iow  }
\cs_generate_variant:Nn \iow_log:n { x }
\cs_new_protected:Npn \iow_term:n { \iow_now:Nn \c_term_iow }
\cs_set_protected:Npn \iow_term:e { \iow_now:Ne \c_term_iow }
\cs_generate_variant:Nn \iow_term:n { x }
```

(*End of definition for* `\iow_log:n` *and* `\iow_term:n`. *These functions are documented on page 6.*)

3.3.3 Special characters for writing

`\iow_newline:` Global variable holding the character that forces a new line when something is written to an output stream.

```
\cs_new:Npn \iow_newline: { ^^J }
```

(*End of definition for* `\iow_newline:`. *This function is documented on page 7.*)

`\iow_char:N` Function to write any escaped char to an output stream.

```
\cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(*End of definition for* `\iow_char:N`. *This function is documented on page 7.*)

3.3.4 Hard-wrapping lines to a character count

The code here implements a generic hard-wrapping function. This is used by the messaging system, but is designed such that it is available for other uses.

`\l_iow_line_count_int` This is the "raw" number of characters in a line which can be written to the terminal. The standard value is the line length typically used by TEX Live and MiKTEX.

```
\int_new:N  \l_iow_line_count_int
\int_set:Nn \l_iow_line_count_int { 78 }
```

(*End of definition for* `\l_iow_line_count_int`. *This variable is documented on page 9.*)

`\l__iow_newline_tl` The token list inserted to produce a new line, with the ⟨*run-on text*⟩.

```
\tl_new:N \l__iow_newline_tl
```

(*End of definition for* `\l__iow_newline_tl`.)

`\l__iow_line_target_int` This stores the target line count: the full number of characters in a line, minus any part for a leader at the start of each line.

```
\int_new:N \l__iow_line_target_int
```

(*End of definition for* `\l__iow_line_target_int`.)

__iow_set_indent:n
__iow_unindent:w
\l__iow_one_indent_tl
\l__iow_one_indent_int

The one_indent variables hold one indentation marker and its length. The __iow_unindent:w auxiliary removes one indentation. The function __iow_set_indent:n (that could possibly be public) sets the indentation in a consistent way. We set it to four spaces by default.

```
\tl_new:N \l__iow_one_indent_tl
\int_new:N \l__iow_one_indent_int
\cs_new:Npn \__iow_unindent:w { }
\cs_new_protected:Npn \__iow_set_indent:n #1
  {
    \__kernel_tl_set:Ne \l__iow_one_indent_tl
      { \exp_args:No \__kernel_str_to_other_fast:n { \tl_to_str:n {#1} } }
    \int_set:Nn \l__iow_one_indent_int
      { \str_count:N \l__iow_one_indent_tl }
    \exp_last_unbraced:NNo
      \cs_set:Npn \__iow_unindent:w \l__iow_one_indent_tl { }
  }
\exp_args:Ne \__iow_set_indent:n { \prg_replicate:nn { 4 } { ~ } }
```

(*End of definition for* __iow_set_indent:n *and others.*)

\l__iow_indent_tl
\l__iow_indent_int

The current indentation (some copies of \l__iow_one_indent_tl) and its number of characters.

```
\tl_new:N \l__iow_indent_tl
\int_new:N \l__iow_indent_int
```

(*End of definition for* \l__iow_indent_tl *and* \l__iow_indent_int.)

\l__iow_line_tl
\l__iow_line_part_tl

These hold the current line of text and a partial line to be added to it, respectively.

```
\tl_new:N \l__iow_line_tl
\tl_new:N \l__iow_line_part_tl
```

(*End of definition for* \l__iow_line_tl *and* \l__iow_line_part_tl.)

\l__iow_line_break_bool

Indicates whether the line was broken precisely at a chunk boundary.

```
\bool_new:N \l__iow_line_break_bool
```

(*End of definition for* \l__iow_line_break_bool.)

\l__iow_wrap_tl

Used for the expansion step before detokenizing, and for the output from wrapping text: fully expanded and with lines which are not overly long.

```
\tl_new:N \l__iow_wrap_tl
```

(*End of definition for* \l__iow_wrap_tl.)

\c__iow_wrap_marker_tl
\c__iow_wrap_end_marker_tl
\c__iow_wrap_newline_marker_tl
\c__iow_wrap_allow_break_marker_tl
\c__iow_wrap_indent_marker_tl
\c__iow_wrap_unindent_marker_tl

Every special action of the wrapping code is starts with the same recognizable string, \c__iow_wrap_marker_tl. Upon seeing that "word", the wrapping code reads one space-delimited argument to know what operation to perform. The setting of \escapechar here is not very important, but makes \c__iow_wrap_marker_tl look marginally nicer.

```
\group_begin:
  \int_set:Nn \tex_escapechar:D { -1 }
  \tl_const:Ne \c__iow_wrap_marker_tl
    { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } }
\group_end:
\tl_map_inline:nn
```

```
  { { end } { newline } { allow_break } { indent } { unindent } }
  {
    \tl_const:ce { c__iow_wrap_ #1 _marker_tl }
      {
        \c__iow_wrap_marker_tl
        #1
        \c_catcode_other_space_tl
      }
  }
```

(*End of definition for* `\c__iow_wrap_marker_tl` *and others.*)

`\iow_wrap_allow_break:`
`\__iow_wrap_allow_break:`
`\__iow_wrap_allow_break_error:`

We set `\iow_wrap_allow_break:n` to produce an error when outside messages. Within wrapped message, it is set to `\__iow_wrap_allow_break:` when valid and otherwise to `\__iow_wrap_allow_break_error:`. The second produces an error expandably.

```
\cs_new_protected:Npn \iow_wrap_allow_break:
  {
    \msg_error:nnnn { kernel } { iow-indent }
      { \iow_wrap:nnnN } { \iow_wrap_allow_break: }
  }
\cs_new:Npe \__iow_wrap_allow_break: { \c__iow_wrap_allow_break_marker_tl }
\cs_new:Npn \__iow_wrap_allow_break_error:
  {
    \msg_expandable_error:nnnn { kernel } { iow-indent }
      { \iow_wrap:nnnN } { \iow_wrap_allow_break: }
  }
```

(*End of definition for* `\iow_wrap_allow_break:`, `\__iow_wrap_allow_break:`, *and* `\__iow_wrap_allow_break_error:`. *This function is documented on page 8.*)

`\iow_indent:n`
`\__iow_indent:n`
`\__iow_indent_error:n`

We set `\iow_indent:n` to produce an error when outside messages. Within wrapped message, it is set to `\__iow_indent:n` when valid and otherwise to `\__iow_indent_error:n`. The first places the instruction for increasing the indentation before its argument, and the instruction for unindenting afterwards. The second produces an error expandably. Note that there are no forced line-break, so the indentation only changes when the next line is started.

```
\cs_new_protected:Npn \iow_indent:n #1
  {
    \msg_error:nnnnn { kernel } { iow-indent }
      { \iow_wrap:nnnN } { \iow_indent:n } {#1}
    #1
  }
\cs_new:Npe \__iow_indent:n #1
  {
    \c__iow_wrap_indent_marker_tl
    #1
    \c__iow_wrap_unindent_marker_tl
  }
\cs_new:Npn \__iow_indent_error:n #1
  {
    \msg_expandable_error:nnnnn { kernel } { iow-indent }
      { \iow_wrap:nnnN } { \iow_indent:n } {#1}
    #1
  }
```

(*End of definition for* `\iow_indent:n`, `\__iow_indent:n`, *and* `\__iow_indent_error:n`. *This function is documented on page 8.*)

`\iow_wrap:nnnN`
`\iow_wrap:nenN`

The main wrapping function works as follows. First give `\\`, `\␣` and other formatting commands the correct definition for messages and perform the given setup `#3`. The definition of `\␣` uses an "other" space rather than a normal space, because the latter might be absorbed by TEX to end a number or other `f`-type expansions. Use `\conditionally@traceoff` if defined; it is introduced by the `trace` package and suppresses uninteresting tracing of the wrapping code.

```
\cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
  {
    \group_begin:
      \cs_if_exist_use:N \conditionally@traceoff
      \int_set:Nn \tex_escapechar:D { -1 }
      \cs_set:Npe \{ { \token_to_str:N \{ }
      \cs_set:Npe \# { \token_to_str:N \# }
      \cs_set:Npe \} { \token_to_str:N \} }
      \cs_set:Npe \% { \token_to_str:N \% }
      \cs_set:Npe \~ { \token_to_str:N \~ }
      \int_set:Nn \tex_escapechar:D { 92 }
      \cs_set_eq:NN \\ \iow_newline:
      \cs_set_eq:NN \  \c_catcode_other_space_tl
      \cs_set_eq:NN \iow_wrap_allow_break: \__iow_wrap_allow_break:
      \cs_set_eq:NN \iow_indent:n \__iow_indent:n
      #3
```

Then fully-expand the input: in package mode, the expansion uses LaTeX 2_ε's `\protect` mechanism in the same way as `\typeout`. In generic mode this setting is useless but harmless. As soon as the expansion is done, reset `\iow_indent:n` to its error definition: it only works in the first argument of `\iow_wrap:nnnN`.

```
      \cs_set_eq:NN \protect \token_to_str:N
      \__kernel_tl_set:Ne \l__iow_wrap_tl {#1}
      \cs_set_eq:NN \iow_wrap_allow_break: \__iow_wrap_allow_break_error:
      \cs_set_eq:NN \iow_indent:n \__iow_indent_error:n
```

Afterwards, set the newline marker (two assignments to fully expand, then convert to a string) and initialize the target count for lines (the first line has target count `\l_iow_line_count_int` instead).

```
      \__kernel_tl_set:Ne \l__iow_newline_tl { \iow_newline: #2 }
      \__kernel_tl_set:Ne \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
      \int_set:Nn \l__iow_line_target_int
        { \l_iow_line_count_int - \str_count:N \l__iow_newline_tl + 1 }
```

Sanity check.

```
      \int_compare:nNnT { \l__iow_line_target_int } < 0
        {
          \tl_set:Nn \l__iow_newline_tl { \iow_newline: }
          \int_set:Nn \l__iow_line_target_int
            { \l_iow_line_count_int + 1 }
        }
```

There is then a loop over the input, which stores the wrapped result in `\l__iow_wrap_tl`. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The `\tl_to_str:N` step converts the "other" spaces back to normal spaces. The `f`-expansion removes a leading space from `\l__iow_wrap_tl`.

```
      \__iow_wrap_do:
    \exp_args:NNf \group_end:
    #4 { \tl_to_str:N \l__iow_wrap_tl }
  }
\cs_generate_variant:Nn \iow_wrap:nnnN { ne }
```

(*End of definition for* `\iow_wrap:nnnN`. *This function is documented on page* 8.)

`\__iow_wrap_do:`
`\__iow_wrap_fix_newline:w`
`\__iow_wrap_start:w`

Escape spaces and change newlines to `\c__iow_wrap_newline_marker_tl`. Set up a few variables, in particular the initial value of `\l__iow_wrap_tl`: the space stops the `f`-expansion of the main wrapping function and `\use_none:n` removes a newline marker inserted by later code. The main loop consists of repeatedly calling the `chunk` auxiliary to wrap chunks delimited by (newline or indentation) markers.

```
\cs_new_protected:Npn \__iow_wrap_do:
  {
    \__kernel_tl_set:Ne \l__iow_wrap_tl
      {
        \exp_args:No \__kernel_str_to_other_fast:n \l__iow_wrap_tl
        \c__iow_wrap_end_marker_tl
      }
    \__kernel_tl_set:Ne \l__iow_wrap_tl
      {
        \exp_after:wN \__iow_wrap_fix_newline:w \l__iow_wrap_tl
          ^^J \q__iow_nil ^^J \s__iow_stop
      }
    \exp_after:wN \__iow_wrap_start:w \l__iow_wrap_tl
  }
\cs_new:Npn \__iow_wrap_fix_newline:w #1 ^^J #2 ^^J
  {
    #1
    \if_meaning:w \q__iow_nil #2
      \__iow_use_i_delimit_by_s_stop:nw
    \fi:
    \c__iow_wrap_newline_marker_tl
    \__iow_wrap_fix_newline:w #2 ^^J
  }
\cs_new_protected:Npn \__iow_wrap_start:w
  {
    \bool_set_false:N \l__iow_line_break_bool
    \tl_clear:N \l__iow_line_tl
    \tl_clear:N \l__iow_line_part_tl
    \tl_set:Nn \l__iow_wrap_tl { ~ \use_none:n }
    \int_zero:N \l__iow_indent_int
    \tl_clear:N \l__iow_indent_tl
    \__iow_wrap_chunk:nw { \l_iow_line_count_int }
  }
```

(*End of definition for* `\__iow_wrap_do:`, `\__iow_wrap_fix_newline:w`, *and* `\__iow_wrap_start:w`.)

`\__iow_wrap_chunk:nw`
`\__iow_wrap_next:nw`

The `chunk` and `next` auxiliaries are defined indirectly to obtain the expansions of `\c_catcode_other_space_tl` and `\c__iow_wrap_marker_tl` in their definition. The `next` auxiliary calls a function corresponding to the type of marker (its `##2`), which can be `newline` or `indent` or `unindent` or `end`. The first argument of the `chunk` auxiliary is a target number of characters and the second is some string to wrap. If the chunk is

empty simply call `next`. Otherwise, set up a call to `\__iow_wrap_line:nw`, including the indentation if the current line is empty, and including a trailing space (`#1`) before the `\__iow_wrap_end_chunk:w` auxiliary.

```
\cs_set_protected:Npn \__iow_tmp:w #1#2
  {
    \cs_new_protected:Npn \__iow_wrap_chunk:nw ##1##2 #2
      {
        \tl_if_empty:nTF {##2}
          {
            \tl_clear:N \l__iow_line_part_tl
            \__iow_wrap_next:nw {##1}
          }
          {
            \tl_if_empty:NTF \l__iow_line_tl
              {
                \__iow_wrap_line:nw
                  { \l__iow_indent_tl }
                  ##1 - \l__iow_indent_int ;
              }
              { \__iow_wrap_line:nw { } ##1 ; }
            ##2 #1
            \__iow_wrap_end_chunk:w 7 6 5 4 3 2 1 0 \s__iow_stop
          }
      }
    \cs_new_protected:Npn \__iow_wrap_next:nw ##1##2 #1
      { \use:c { __iow_wrap_##2:n } {##1} }
  }
\exp_args:NVV \__iow_tmp:w \c_catcode_other_space_tl \c__iow_wrap_marker_tl
```

(*End of definition for* `\__iow_wrap_chunk:nw` *and* `\__iow_wrap_next:nw`.)

`\__iow_wrap_line:nw`
`\__iow_wrap_line_loop:w`
`\__iow_wrap_line_aux:Nw`
`\__iow_wrap_line_seven:nnnnnnn`
`\__iow_wrap_line_end:NnnnnnnnN`
`\__iow_wrap_line_end:nw`
`\__iow_wrap_end_chunk:w`

This is followed by `{`⟨*string*⟩`}` ⟨*int expr*⟩ `;`. It stores the ⟨*string*⟩ and up to ⟨*int expr*⟩ characters from the current chunk into `\l__iow_line_part_tl`. Characters are grabbed 8 at a time and left in `\l__iow_line_part_tl` by the `line_loop` auxiliary. When $k < 8$ remain to be found, the `line_aux` auxiliary calls the `line_end` auxiliary followed by (the single digit) k, then $7 - k$ empty brace groups, then the chunk's remaining characters. The `line_end` auxiliary leaves k characters from the chunk in the line part, then ends the assignment. Ignore the `\use_none:nnnnn` line for now. If the next character is a space the line can be broken there: store what we found into the result and get the next line. Otherwise some work is needed to find a break-point. So far we have ignored what happens if the chunk is shorter than the requested number of characters: this is dealt with by the `end_chunk` auxiliary, which gets treated like a character by the rest of the code. It ends up being called either as one of the arguments `#2`–`#9` of the `line_loop` auxiliary or as one of the arguments `#2`–`#8` of the `line_end` auxiliary. In both cases stop the assignment and work out how many characters are still needed. Notice that when we have exactly seven arguments to clean up, a `\exp_stop_f:` has to be inserted to stop the `\exp:w`. The weird `\use_none:nnnnn` ensures that the required data is in the right place.

```
\cs_new_protected:Npn \__iow_wrap_line:nw #1
  {
    \tex_edef:D \l__iow_line_part_tl { \if_false: } \fi:
    #1
```

```
    \exp_after:wN \__iow_wrap_line_loop:w
    \int_value:w \int_eval:w
  }
\cs_new:Npn \__iow_wrap_line_loop:w #1 ; #2#3#4#5#6#7#8#9
  {
    \if_int_compare:w #1 < 8 \exp_stop_f:
      \__iow_wrap_line_aux:Nw #1
    \fi:
    #2 #3 #4 #5 #6 #7 #8 #9
    \exp_after:wN \__iow_wrap_line_loop:w
    \int_value:w \int_eval:w #1 - 8 ;
  }
\cs_new:Npn \__iow_wrap_line_aux:Nw #1#2#3 \exp_after:wN #4 ;
  {
    #2
    \exp_after:wN \__iow_wrap_line_end:NnnnnnnnN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
    \exp_after:wN \exp_after:wN
    \if_case:w #1 \exp_stop_f:
         \prg_do_nothing:
    \or: \use_none:n
    \or: \use_none:nn
    \or: \use_none:nnn
    \or: \use_none:nnnn
    \or: \use_none:nnnnn
    \or: \use_none:nnnnnn
    \or: \__iow_wrap_line_seven:nnnnnnn
    \fi:
    { } { } { } { } { } { } { } #3
  }
\cs_new:Npn \__iow_wrap_line_seven:nnnnnnn #1#2#3#4#5#6#7 { \exp_stop_f: }
\cs_new:Npn \__iow_wrap_line_end:NnnnnnnnN #1#2#3#4#5#6#7#8#9
  {
    #2 #3 #4 #5 #6 #7 #8
    \use_none:nnnnn \int_eval:w 8 - ; #9
    \token_if_eq_charcode:NNTF \c_space_token #9
      { \__iow_wrap_line_end:nw { } }
      { \if_false: { \fi: } \__iow_wrap_break:w #9 }
  }
\cs_new:Npn \__iow_wrap_line_end:nw #1
  {
    \if_false: { \fi: }
    \__iow_wrap_store_do:n {#1}
    \__iow_wrap_next_line:w
  }
\cs_new:Npn \__iow_wrap_end_chunk:w
    #1 \int_eval:w #2 - #3 ; #4#5 \s__iow_stop
  {
    \if_false: { \fi: }
    \exp_args:Nf \__iow_wrap_next:nw { \int_eval:n { #2 - #4 } }
  }
```

(*End of definition for* `\__iow_wrap_line:nw` *and others.*)

__iow_wrap_break:w
__iow_wrap_break_first:w
__iow_wrap_break_none:w
__iow_wrap_break_loop:w
__iow_wrap_break_end:w

Functions here are defined indirectly: `\__iow_tmp:w` is eventually called with an "other" space as its argument. The goal is to remove from `\l__iow_line_part_tl` the part after the last space. In most cases this is done by repeatedly calling the `break_loop` auxiliary, which leaves "words" (delimited by spaces) until it hits the trailing space: then its argument `##3` is `? \__iow_wrap_break_end:w` instead of a single token, and that `break_end` auxiliary leaves in the assignment the line until the last space, then calls `\__iow_wrap_line_end:nw` to finish up the line and move on to the next. If there is no space in `\l__iow_line_part_tl` then the `break_first` auxiliary calls the `break_none` auxiliary. In that case, if the current line is empty, the complete word (including `##4`, characters beyond what we had grabbed) is added to the line, making it over-long. Otherwise, the word is used for the following line (and the last space of the line so far is removed because it was inserted due to the presence of a marker).

```
\cs_set_protected:Npn \__iow_tmp:w #1
  {
    \cs_new:Npn \__iow_wrap_break:w
      {
        \tex_edef:D \l__iow_line_part_tl
          { \if_false: } \fi:
            \exp_after:wN \__iow_wrap_break_first:w
            \l__iow_line_part_tl
            #1
            { ? \__iow_wrap_break_end:w }
            \s__iow_mark
      }
    \cs_new:Npn \__iow_wrap_break_first:w ##1 #1 ##2
      {
        \use_none:nn ##2 \__iow_wrap_break_none:w
        \__iow_wrap_break_loop:w ##1 #1 ##2
      }
    \cs_new:Npn \__iow_wrap_break_none:w ##1##2 #1 ##3 \s__iow_mark ##4 #1
      {
        \tl_if_empty:NTF \l__iow_line_tl
          { ##2 ##4 \__iow_wrap_line_end:nw { } }
          { \__iow_wrap_line_end:nw { \__iow_wrap_trim:N } ##2 ##4 #1 }
      }
    \cs_new:Npn \__iow_wrap_break_loop:w ##1 #1 ##2 #1 ##3
      {
        \use_none:n ##3
        ##1 #1
        \__iow_wrap_break_loop:w ##2 #1 ##3
      }
    \cs_new:Npn \__iow_wrap_break_end:w ##1 #1 ##2 ##3 #1 ##4 \s__iow_mark
      { ##1 \__iow_wrap_line_end:nw { } ##3 }
  }
\exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl
```

(*End of definition for* `\__iow_wrap_break:w` *and others.*)

__iow_wrap_next_line:w

The special case where the end of a line coincides with the end of a chunk is detected here, to avoid a spurious empty line. Otherwise, call `\__iow_wrap_line:nw` to find characters for the next line (remembering to account for the indentation).

```
\cs_new_protected:Npn \__iow_wrap_next_line:w #1#2 \s__iow_stop
  {
```

```
    \tl_clear:N \l__iow_line_tl
    \token_if_eq_meaning:NNTF #1 \__iow_wrap_end_chunk:w
      {
        \tl_clear:N \l__iow_line_part_tl
        \bool_set_true:N \l__iow_line_break_bool
        \__iow_wrap_next:nw { \l__iow_line_target_int }
      }
      {
        \__iow_wrap_line:nw
          { \l__iow_indent_tl }
          \l__iow_line_target_int - \l__iow_indent_int ;
          #1 #2 \s__iow_stop
      }
  }
```

(*End of definition for* `\__iow_wrap_next_line:w`.)

`\__iow_wrap_allow_break:n` This is called after a chunk has been wrapped. The `\l__iow_line_part_tl` typically ends with a space (except at the beginning of a line?), which we remove since the `allow_break` marker should not insert a space. Then move on with the next chunk, making sure to adjust the target number of characters for the line in case we did remove a space.

```
\cs_new_protected:Npn \__iow_wrap_allow_break:n #1
  {
    \__kernel_tl_set:Ne \l__iow_line_tl
      { \l__iow_line_tl \__iow_wrap_trim:N \l__iow_line_part_tl }
    \bool_set_false:N \l__iow_line_break_bool
    \tl_if_empty:NTF \l__iow_line_part_tl
      { \__iow_wrap_chunk:nw {#1} }
      { \exp_args:Nf \__iow_wrap_chunk:nw { \int_eval:n { #1 + 1 } } }
  }
```

(*End of definition for* `\__iow_wrap_allow_break:n`.)

`\__iow_wrap_indent:n` `\__iow_wrap_unindent:n` These functions are called after a chunk has been wrapped, when encountering `indent`/`unindent` markers. Add the line part (last line part of the previous chunk) to the line so far and reset a boolean denoting the presence of a line-break. Most importantly, add or remove one indent from the current indent (both the integer and the token list). Finally, continue wrapping.

```
\cs_new_protected:Npn \__iow_wrap_indent:n #1
  {
    \tl_put_right:Ne \l__iow_line_tl { \l__iow_line_part_tl }
    \bool_set_false:N \l__iow_line_break_bool
    \int_add:Nn \l__iow_indent_int { \l__iow_one_indent_int }
    \tl_put_right:No \l__iow_indent_tl { \l__iow_one_indent_tl }
    \__iow_wrap_chunk:nw {#1}
  }
\cs_new_protected:Npn \__iow_wrap_unindent:n #1
  {
    \tl_put_right:Ne \l__iow_line_tl { \l__iow_line_part_tl }
    \bool_set_false:N \l__iow_line_break_bool
    \int_sub:Nn \l__iow_indent_int { \l__iow_one_indent_int }
    \__kernel_tl_set:Ne \l__iow_indent_tl
      { \exp_after:wN \__iow_unindent:w \l__iow_indent_tl }
    \__iow_wrap_chunk:nw {#1}
  }
```

(*End of definition for* `\__iow_wrap_indent:n` *and* `\__iow_wrap_unindent:n`.)

`\__iow_wrap_newline:n`
`\__iow_wrap_end:n`

These functions are called after a chunk has been line-wrapped, when encountering a `newline`/`end` marker. Unless we just took a line-break, store the line part and the line so far into the whole `\l__iow_wrap_tl`, trimming a trailing space. In the `newline` case look for a new line (of length `\l__iow_line_target_int`) in a new chunk.

```
\cs_new_protected:Npn \__iow_wrap_newline:n #1
  {
    \bool_if:NF \l__iow_line_break_bool
      { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
    \bool_set_false:N \l__iow_line_break_bool
    \__iow_wrap_chunk:nw { \l__iow_line_target_int }
  }
\cs_new_protected:Npn \__iow_wrap_end:n #1
  {
    \bool_if:NF \l__iow_line_break_bool
      { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
    \bool_set_false:N \l__iow_line_break_bool
  }
```

(*End of definition for* `\__iow_wrap_newline:n` *and* `\__iow_wrap_end:n`.)

`\__iow_wrap_store_do:n`

First add the last line part to the line, then append it to `\l__iow_wrap_tl` with the appropriate new line (with "run-on" text), possibly with its last space removed (`#1` is empty or `\__iow_wrap_trim:N`).

```
\cs_new_protected:Npn \__iow_wrap_store_do:n #1
  {
    \__kernel_tl_set:Ne \l__iow_line_tl
      { \l__iow_line_tl \l__iow_line_part_tl }
    \__kernel_tl_set:Ne \l__iow_wrap_tl
      {
        \l__iow_wrap_tl
        \l__iow_newline_tl
        #1 \l__iow_line_tl
      }
    \tl_clear:N \l__iow_line_tl
  }
```

(*End of definition for* `\__iow_wrap_store_do:n`.)

`\__iow_wrap_trim:N`
`\__iow_wrap_trim:w`
`\__iow_wrap_trim_aux:w`

Remove one trailing "other" space from the argument if present.

```
\cs_set_protected:Npn \__iow_tmp:w #1
  {
    \cs_new:Npn \__iow_wrap_trim:N ##1
      { \exp_after:wN \__iow_wrap_trim:w ##1 \s__iow_mark #1 \s__iow_mark \s__iow_stop }
    \cs_new:Npn \__iow_wrap_trim:w ##1 #1 \s__iow_mark
      { \__iow_wrap_trim_aux:w ##1 \s__iow_mark }
    \cs_new:Npn \__iow_wrap_trim_aux:w ##1 \s__iow_mark ##2 \s__iow_stop {##1}
  }
\exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl
```

(*End of definition for* `\__iow_wrap_trim:N`, `\__iow_wrap_trim:w`, *and* `\__iow_wrap_trim_aux:w`.)

⟨`@@=file`⟩

3.4 File operations

\l__file_internal_tl

Used as a short-term scratch variable.

```
\tl_new:N \l__file_internal_tl
```

(*End of definition for* `\l__file_internal_tl`.)

\g_file_curr_dir_str
\g_file_curr_ext_str
\g_file_curr_name_str

The name of the current file should be available at all times: the name itself is set dynamically.

```
\str_new:N \g_file_curr_dir_str
\str_new:N \g_file_curr_ext_str
\str_new:N \g_file_curr_name_str
```

(*End of definition for* `\g_file_curr_dir_str`, `\g_file_curr_ext_str`, *and* `\g_file_curr_name_str`. *These variables are documented on page* 9.)

\g__file_stack_seq

The input list of files is stored as a sequence stack. In package mode we can recover the information from the details held by LaTeX 2_ε (we must be in the preamble and loaded using `\usepackage` or `\RequirePackage`). As LaTeX 2_ε doesn't store directory and name separately, we stick to the same convention here. In pre-loading, `\@currnamestack` is empty so is skipped.

```
\seq_new:N \g__file_stack_seq
\group_begin:
  \cs_set_protected:Npn \__file_tmp:w #1#2#3
    {
      \tl_if_blank:nTF {#1}
        {
          \cs_set:Npn \__file_tmp:w ##1 " ##2 " ##3 \s__file_stop
            { { } {##2} {  } }
          \seq_gput_right:Ne \g__file_stack_seq
            {
              \exp_after:wN \__file_tmp:w \tex_jobname:D
                " \tex_jobname:D " \s__file_stop
            }
        }
        {
          \seq_gput_right:Nn \g__file_stack_seq { { } {#1} {#2} }
          \__file_tmp:w
        }
    }
  \cs_if_exist:NT \@currnamestack
    {
      \tl_if_empty:NF \@currnamestack
        { \exp_after:wN \__file_tmp:w \@currnamestack }
    }
\group_end:
```

(*End of definition for* `\g__file_stack_seq`.)

\g__file_record_seq

The total list of files used is recorded separately from the current file stack, as nothing is ever popped from this list. The current file name should be included in the file list! We will eventually copy the contents of `\@filelist`.

```
\seq_new:N \g__file_record_seq
```

(*End of definition for* `\g__file_record_seq`.)

\l__file_base_name_tl
\l__file_full_name_tl

For storing the basename and full path whilst passing data internally.

```
\tl_new:N \l__file_base_name_tl
\tl_new:N \l__file_full_name_tl
```

(*End of definition for* `\l__file_base_name_tl` *and* `\l__file_full_name_tl`.)

\l__file_dir_str
\l__file_ext_str
\l__file_name_str

Used in parsing a path into parts: in contrast to the above, these are never used outside of the current module.

```
\str_new:N \l__file_dir_str
\str_new:N \l__file_ext_str
\str_new:N \l__file_name_str
```

(*End of definition for* `\l__file_dir_str`, `\l__file_ext_str`, *and* `\l__file_name_str`.)

\l_file_search_path_seq

The current search path.

```
\seq_new:N \l_file_search_path_seq
```

(*End of definition for* `\l_file_search_path_seq`. *This variable is documented on page 10.*)

\l__file_tmp_seq

Scratch space for comma list conversion.

```
\seq_new:N \l__file_tmp_seq
```

(*End of definition for* `\l__file_tmp_seq`.)

3.4.1 Internal auxiliaries

\s__file_stop

Internal scan marks.

```
\scan_new:N \s__file_stop
```

(*End of definition for* `\s__file_stop`.)

\q__file_nil

Internal quarks.

```
\quark_new:N \q__file_nil
```

(*End of definition for* `\q__file_nil`.)

__file_quark_if_nil_p:n
__file_quark_if_nil:n*TF*

Branching quark conditional.

```
\__kernel_quark_new_conditional:Nn \__file_quark_if_nil:n { TF }
```

(*End of definition for* `\__file_quark_if_nil:nTF`.)

\q__file_recursion_tail
\q__file_recursion_stop

Internal recursion quarks.

```
\quark_new:N \q__file_recursion_tail
\quark_new:N \q__file_recursion_stop
```

(*End of definition for* `\q__file_recursion_tail` *and* `\q__file_recursion_stop`.)

__file_if_recursion_tail_break:NN
__file_if_recursion_tail_stop_do:Nn

Functions to query recursion quarks.

```
\__kernel_quark_new_test:N \__file_if_recursion_tail_stop:N
\__kernel_quark_new_test:N \__file_if_recursion_tail_stop_do:nn
```

(*End of definition for* `\__file_if_recursion_tail_break:NN` *and* `\__file_if_recursion_tail_stop_do:Nn`.)

__kernel_file_name_sanitize:n
__file_name_expand:n
__file_name_expand_cleanup:Nw
__file_name_expand_cleanup:w
__file_name_expand_end:
__file_name_expand_error:Nw
__file_name_expand_error_aux:Nw
__file_name_strip_quotes:n
__file_name_strip_quotes:nnnw
__file_name_strip_quotes:nnn
__file_name_trim_spaces:n
__file_name_trim_spaces:nw
__file_name_trim_spaces_aux:n
__file_name_trim_spaces_aux:w

Expanding the file name uses a `\csname`-based approach, and relies on active characters (for example from UTF-8 characters) being properly set up to expand to a expansion-safe version using `\ifcsname`. This is less conservative than the token-by-token approach used before, but it is much faster.

```
\cs_new:Npn \__kernel_file_name_sanitize:n #1
  {
    \exp_args:Ne \__file_name_trim_spaces:n
      {
        \exp_args:Ne \__file_name_strip_quotes:n
          { \__file_name_expand:n {#1} }
      }
  }
```

We'll use `\cs:w` to start expanding the file name, and to avoid creating csnames equal to `\relax` with "common" names, there's a prefix `__file_name=` to the csname. There's also a guard token at the end so we can check if there was an error during the process and (try to) clean up gracefully.

```
\cs_new:Npn \__file_name_expand:n #1
  {
    \exp_after:wN \__file_name_expand_cleanup:Nw
      \cs:w __file_name = #1 \cs_end:
        \__file_name_expand_end:
  }
```

With the csname built, we grab it, and grab the remaining tokens delimited by `\__file_name_expand_end:`. If there are any remaining tokens, something bad happened, so we'll call the error procedure `\__file_name_expand_error:Nw`. If everything went according to plan, then use `\token_to_str:N` on the csname built, and call `\__file_name_expand_cleanup:w` to remove the prefix we added a while back. `\__file_name_expand_cleanup:w` takes a leading argument so we don't have to bother about the value of `\tex_escapechar:D`.

```
\cs_new:Npn \__file_name_expand_cleanup:Nw #1 #2 \__file_name_expand_end:
  {
    \tl_if_empty:nF {#2}
      { \__file_name_expand_error:Nw #2 \__file_name_expand_end: }
    \exp_after:wN \__file_name_expand_cleanup:w \token_to_str:N #1
  }
\exp_last_unbraced:NNNNo
\cs_new:Npn \__file_name_expand_cleanup:w #1 \tl_to_str:n { __file_name = } { }
```

In non-error cases `\__file_name_expand_end:` should not expand. It will only do so in case there is a `\csname` too much in the file name, so it will throw an error (while expanding), then insert the missing `\cs_end:` and yet another `\__file_name_expand_end:` that will be used as a delimiter by `\__file_name_expand_cleanup:Nw` (or that will expand again if yet another `\endcsname` is missing).

```
\cs_new:Npn \__file_name_expand_end:
  {
    \msg_expandable_error:nn
      { kernel } { filename-missing-endcsname }
    \cs_end: \__file_name_expand_end:
  }
```

Now to the error case. `\__file_name_expand_error:Nw` adds an extra `\cs_end:` so that in case there was an extra `\csname` in the file name, then `\__file_name_expand_error_aux:Nw` throws the error.

```
\cs_new:Npn \__file_name_expand_error:Nw #1 #2 \__file_name_expand_end:
  { \__file_name_expand_error_aux:Nw #1 #2 \cs_end: \__file_name_expand_end: }
\cs_new:Npn \__file_name_expand_error_aux:Nw #1 #2 \cs_end: #3
    \__file_name_expand_end:
  {
    \msg_expandable_error:nnff
      { kernel } { filename-chars-lost }
        { \token_to_str:N #1 } { \exp_stop_f: #2 }
  }
```

Quoting file name uses basically the same approach as for `luaquotejobname`: count the `"` tokens and remove them.

```
\cs_new:Npn \__file_name_strip_quotes:n #1
  {
    \__file_name_strip_quotes:nw { 0 }
      #1 " \q__file_recursion_tail " \q__file_recursion_stop {#1}
  }
\cs_new:Npn \__file_name_strip_quotes:nw #1#2 "
  {
    \if_meaning:w \q__file_recursion_tail #2
      \__file_name_strip_quotes_end:wnwn
    \fi:
    #2
    \__file_name_strip_quotes:nw { #1 + 1 }
  }
\cs_new:Npn \__file_name_strip_quotes_end:wnwn \fi: #1
    \__file_name_strip_quotes:nw #2 \q__file_recursion_stop #3
  {
    \fi:
    \int_if_odd:nT {#2}
      {
        \msg_expandable_error:nnn
          { kernel } { unbalanced-quote-in-filename } {#3}
      }
  }
```

Spaces need to be trimmed from the start of the name and from the end of any extension. However, the name we are passed might not have an extension: that means we have to look for one. If there is no extension, we still use the standard trimming function but deliberately prevent any spaces being removed at the end.

```
\cs_new:Npn \__file_name_trim_spaces:n #1
  { \__file_name_trim_spaces:nw {#1} #1 . \q__file_nil . \s__file_stop }
\cs_new:Npn \__file_name_trim_spaces:nw #1#2 . #3 . #4 \s__file_stop
  {
    \__file_quark_if_nil:nTF {#3}
      {
        \tl_trim_spaces_apply:nN { #1 \s__file_stop }
          \__file_name_trim_spaces_aux:n
      }
      { \tl_trim_spaces:n {#1} }
  }
```

```
\cs_new:Npn \__file_name_trim_spaces_aux:n #1
  { \__file_name_trim_spaces_aux:w #1 }
\cs_new:Npn \__file_name_trim_spaces_aux:w #1 \s__file_stop {#1}
```

(*End of definition for* `\__kernel_file_name_sanitize:n` *and others.*)

`\__kernel_file_name_quote:n`
`\__file_name_quote:nw`

```
\cs_new:Npn \__kernel_file_name_quote:n #1
  { \__file_name_quote:nw {#1} #1 ~ \q__file_nil \s__file_stop }
\cs_new:Npn \__file_name_quote:nw #1 #2 ~ #3 \s__file_stop
  {
    \__file_quark_if_nil:nTF {#3}
      { #1 }
      { "#1" }
  }
```

(*End of definition for* `\__kernel_file_name_quote:n` *and* `\__file_name_quote:nw`.)

`\c__file_marker_tl` The same idea as the marker for rescanning token lists: this pair of tokens cannot appear in a file that is being input.

```
\tl_const:Ne \c__file_marker_tl { : \token_to_str:N : }
```

(*End of definition for* `\c__file_marker_tl`.)

`\file_get:nnNTF`
`\file_get:VnNTF`
`\file_get:nnN`
`\__file_get_aux:nnN`
`\__file_get_do:Nw`

The approach here is similar to that for `\tl_set_rescan:Nnn`. The file contents are grabbed as an argument delimited by `\c__file_marker_tl`. A few subtleties: braces in `\if_false: ... \fi:` to deal with possible alignment tabs, `\tracingnesting` to avoid a warning about a group being closed inside the `\scantokens`, and `\prg_return_true:` is placed after the end-of-file marker.

```
\cs_new_protected:Npn \file_get:nnN #1#2#3
  {
    \file_get:nnNF {#1} {#2} #3
      { \tl_set:Nn #3 { \q_no_value } }
  }
\cs_generate_variant:Nn \file_get:nnN { V }
\prg_new_protected_conditional:Npnn \file_get:nnN #1#2#3 { T , F , TF }
  {
    \file_get_full_name:nNTF {#1} \l__file_full_name_tl
      {
        \exp_args:NV \__file_get_aux:nnN
          \l__file_full_name_tl
          {#2} #3
        \prg_return_true:
      }
      { \prg_return_false: }
  }
\prg_generate_conditional_variant:Nnn \file_get:nnN { V } { T , F , TF }
\cs_new_protected:Npe \__file_get_aux:nnN #1#2#3
  {
    \exp_not:N \if_false: { \exp_not:N \fi:
    \group_begin:
      \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
      \exp_not:N \exp_args:No \tex_everyeof:D
        { \exp_not:N \c__file_marker_tl }
```

```
      #2 \scan_stop:
      \exp_not:N \exp_after:wN \exp_not:N \__file_get_do:Nw
      \exp_not:N \exp_after:wN #3
      \exp_not:N \exp_after:wN \exp_not:N \prg_do_nothing:
      \exp_not:N \tex_input:D
      \sys_if_engine_luatex:TF
        { {#1} }
        { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
    \exp_not:N \if_false: } \exp_not:N \fi:
  }
\exp_args:Nno \use:nn
  { \cs_new_protected:Npn \__file_get_do:Nw #1#2 }
  { \c__file_marker_tl }
  {
    \group_end:
    \tl_set:No #1 {#2}
  }
```

(*End of definition for* `\file_get:nnNTF` *and others. These functions are documented on page* 13.)

`\__file_size:n` A copy of the primitive where it's available.

```
\cs_new_eq:NN \__file_size:n \tex_filesize:D
```

(*End of definition for* `\__file_size:n`.)

`\file_full_name:n` `\__file_full_name:n` `\__file_full_name_aux:n` `\__file_full_name_auxi:nn` `\__file_full_name_auxii:nn` `\__file_full_name_aux:Nnn` `\__file_full_name_slash:n` `\__file_full_name_slash:w` `\__file_full_name_aux:nN` `\__file_full_name_aux:nnN` `\__file_name_cleanup:w` `\__file_name_end:` `\__file_name_ext_check:nn` `\__file_name_ext_check:nnw` `\__file_name_ext_check:nnnw` `\__file_name_ext_check:nnn` `\__file_name_ext_check:nnnn`

File searching can be carried out if the `\pdffilesize` primitive or an equivalent is available. That of course means we need to arrange for everything else to here to be done by expansion too. We start off by sanitizing the name and quoting if required: we may need to remove those quotes, so the raw name is passed too.

```
\cs_new:Npn \file_full_name:n #1
  {
    \exp_args:Ne \__file_full_name:n
      { \__kernel_file_name_sanitize:n {#1} }
  }
\cs_generate_variant:Nn \file_full_name:n { V }
```

First, we check of the file is just here: no mapping so we do not need the break part of the broader auxiliary. We are using the fact that the primitive here returns nothing if the file is entirely absent. To avoid unnecessary filesystem lookups, the result of `\pdffilesize` is kept available as an argument. For package mode, `\input@path` is a token list not a sequence.

```
\cs_new:Npn \__file_full_name:n #1
  {
    \tl_if_blank:nF {#1}
      { \exp_args:Nne \__file_full_name_auxii:nn {#1} { \__file_full_name_aux:n {#1} } }
  }
```

To avoid repeated reading of files we need to cache the loading: this is important as the code here is used by *all* file checks. The same marker is used in the LaTeX 2_ε kernel, meaning that we get a double-saving with for example `\IfFileExists`. As this is all about performance, we use the low-level approach for the conditionals. For a file already seen, the size is reported as -1 so it's distinct from any non-cached ones.

```
\cs_new:Npn \__file_full_name_aux:n #1
  {
```

```
    \if_cs_exist:w __file_seen_ \tl_to_str:n {#1} : \cs_end:
      -1
    \else:
      \exp_args:Ne \__file_full_name_auxi:nn { \__file_size:n {#1} } {#1}
    \fi:
  }
```

We will need the size of files later, and we have to avoid the `\scan_stop:` causing issues if we are raising the flag. Thus there is a slightly odd gobble here.

```
\cs_new:Npn \__file_full_name_auxi:nn #1#2
  {
    \if:w \scan_stop: #1 \scan_stop:
    \else:
      \exp_after:wN \use_none:n
        \cs:w __file_seen_ \tl_to_str:n {#2} : \cs_end:
      #1
    \fi:
  }
\cs_new:Npn \__file_full_name_auxii:nn #1 #2
  {
    \tl_if_blank:nTF {#2}
      {
        \seq_map_tokens:Nn \l_file_search_path_seq
          { \__file_full_name_aux:Nnn \seq_map_break:n {#1} }
        \cs_if_exist:NT \input@path
          {
            \tl_map_tokens:Nn \input@path
              { \__file_full_name_aux:Nnn \tl_map_break:n {#1} }
          }
        \__file_name_end:
      }
      { \__file_ext_check:nn {#1} {#2} }
  }
```

Two pars to the auxiliary here so we can avoid doing quoting twice in the event we find the right file.

```
\cs_new:Npn \__file_full_name_aux:Nnn #1#2#3
  {
    \exp_args:Ne \__file_full_name_aux:nN
      { \__file_full_name_slash:n {#3} #2 }
      #1
  }
\cs_new:Npn \__file_full_name_slash:n #1
  {
    \__file_full_name_slash:nw {#1} #1 \q_nil / \q_nil / \q_nil \q_stop
  }
\cs_new:Npn \__file_full_name_slash:nw #1#2 / \q_nil / #3 \q_stop
  {
    \quark_if_nil:nTF {#3}
      { #1 / }
      { #2 / }
  }
\cs_new:Npn \__file_full_name_aux:nN #1
  { \exp_args:Nne \__file_full_name_aux:nnN {#1} { \__file_full_name_aux:n {#1} } }
\cs_new:Npn \__file_full_name_aux:nnN #1 #2 #3
```

```
  {
    \tl_if_blank:nF {#2}
      {
        #3
          {
            \__file_ext_check:nn {#1} {#2}
            \__file_name_cleanup:w
          }
      }
  }
\cs_new:Npn \__file_name_cleanup:w #1 \__file_name_end: { }
\cs_new:Npn \__file_name_end: { }
```

As TEX automatically adds `.tex` if there is no extension, there is a little clean up to do here. First, make sure we are not in the directory part, saving that. Then check for an extension.

```
\cs_new:Npn \__file_ext_check:nn #1 #2
{ \__file_ext_check:nnw {#2} { / } #1 / \q__file_nil / \s__file_stop }
\cs_new:Npn \__file_ext_check:nnw #1 #2 #3 / #4 / #5 \s__file_stop
  {
    \__file_quark_if_nil:nTF {#4}
      {
        \exp_args:No \__file_ext_check:nnnw
          { \use_none:n #2 } {#1} {#3} #3 . \q__file_nil . \s__file_stop
      }
      { \__file_ext_check:nnw {#1} { #2 #3 / } #4 / #5 \s__file_stop }
  }
\cs_new:Npe \__file_ext_check:nnnw #1#2#3#4 . #5 . #6 \s__file_stop
  {
    \exp_not:N \__file_quark_if_nil:nTF {#5}
      {
        \exp_not:N \__file_ext_check:nnn
          { #1 #3 \tl_to_str:n { .tex } } { #1 #3 } {#2}
      }
      { #1 #3 }
  }
\cs_new:Npn \__file_ext_check:nnn #1
  { \exp_args:Nne \__file_ext_check:nnnn {#1} { \__file_full_name_aux:n {#1} } }
\cs_new:Npn \__file_ext_check:nnnn #1#2#3#4
  {
    \tl_if_blank:nTF {#2}
      {#3}
      {
        \bool_lazy_or:nnTF
          { \int_compare_p:nNn {#4} = {#2} }
          { \int_compare_p:nNn {#2} = { -1 } }
          {#1}
          {#3}
      }
  }
```

(*End of definition for* `\file_full_name:n` *and others. This function is documented on page 12.*)

`\file_get_full_name:nN`
`\file_get_full_name:VN`
`\file_get_full_name:nNTF`
`\file_get_full_name:VNTF`
`\__file_get_full_name_search:nN`

These functions pre-date using `\tex_filesize:D` for file searching, so are `get` functions with protection. To avoid having different search set ups, they are simply wrappers

around the code above.

```
\cs_new_protected:Npn \file_get_full_name:nN #1#2
  {
    \file_get_full_name:nNF {#1} #2
      { \tl_set:Nn #2 { \q_no_value } }
  }
\cs_generate_variant:Nn \file_get_full_name:nN { V }
\prg_new_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
  {
    \__kernel_tl_set:Ne #2
      { \file_full_name:n {#1} }
    \tl_if_empty:NTF #2
      { \prg_return_false: }
      { \prg_return_true: }
  }
\prg_generate_conditional_variant:Nnn \file_get_full_name:nN
  { V } { T , F ,  TF }
```

(*End of definition for* `\file_get_full_name:nN`*,* `\file_get_full_name:nNTF`*, and* `\__file_get_full_name_search:nN`*. These functions are documented on page 12.*)

`\g__file_internal_ior`

A reserved stream to test for opening a shell.

```
\ior_new:N \g__file_internal_ior
```

(*End of definition for* `\g__file_internal_ior`*.*)

`\file_mdfive_hash:n`
`\file_mdfive_hash:V`
`\file_size:n`
`\file_size:V`
`\file_timestamp:n`
`\file_timestamp:V`
`\__file_details:nn`
`\__file_details_aux:nn`
`\__file_mdfive_hash:n`

Getting file details by expansion is relatively easy if a bit repetitive. As the MD5 function has a slightly different syntax from the other commands, there is a little cleaning up to do.

```
\cs_new:Npn \file_size:n #1
  { \__file_details:nn {#1} { size } }
\cs_generate_variant:Nn \file_size:n { V }
\cs_new:Npn \file_timestamp:n #1
  { \__file_details:nn {#1} { moddate } }
\cs_generate_variant:Nn \file_timestamp:n { V }
\cs_new:Npn \__file_details:nn #1#2
  {
    \exp_args:Ne \__file_details_aux:nn
      { \file_full_name:n {#1} } {#2}
  }
\cs_new:Npn \__file_details_aux:nn #1#2
  {
    \tl_if_blank:nF {#1}
      { \use:c { tex_file #2 :D } {#1} }
  }
\cs_new:Npn \file_mdfive_hash:n #1
  { \exp_args:Ne \__file_mdfive_hash:n { \file_full_name:n {#1} } }
\cs_generate_variant:Nn \file_mdfive_hash:n { V }
\cs_new:Npn \__file_mdfive_hash:n #1
  { \tex_mdfivesum:D file {#1} }
```

(*End of definition for* `\file_mdfive_hash:n` *and others. These functions are documented on page 11.*)

\file_hex_dump:nnn
\file_hex_dump:Vnn
__file_hex_dump_auxi:nnn
__file_hex_dump_auxii:nnnn
__file_hex_dump_auxiii:nnnn
__file_hex_dump_auxiiv:nnn
\file_hex_dump:n
\file_hex_dump:V
__file_hex_dump:n

These are separate as they need multiple arguments *or* the file size. For LuaTEX, the emulation does not need the file size so we save a little on expansion.

```
\cs_new:Npn \file_hex_dump:nnn #1#2#3
  {
    \exp_args:Neee \__file_hex_dump_auxi:nnn
      { \file_full_name:n {#1} }
      { \int_eval:n {#2} }
      { \int_eval:n {#3} }
  }
\cs_generate_variant:Nn \file_hex_dump:nnn { V }
\cs_new:Npn \__file_hex_dump_auxi:nnn #1#2#3
  {
    \bool_lazy_any:nF
      {
        { \tl_if_blank_p:n {#1} }
        { \int_compare_p:nNn {#2} = 0 }
        { \int_compare_p:nNn {#3} = 0 }
      }
      {
        \exp_args:Ne \__file_hex_dump_auxii:nnnn
          { \__file_details_aux:nn {#1} { size } }
          {#1} {#2} {#3}
      }
  }
\cs_new:Npn \__file_hex_dump_auxii:nnnn #1#2#3#4
  {
    \int_compare:nNnTF {#3} > 0
      { \__file_hex_dump_auxiii:nnnn {#3} }
      {
        \exp_args:Ne \__file_hex_dump_auxiii:nnnn
          { \int_eval:n { #1 + #3 } }
      }
        {#1} {#2} {#4}
  }
\cs_new:Npn \__file_hex_dump_auxiii:nnnn #1#2#3#4
  {
    \int_compare:nNnTF {#4} > 0
      { \__file_hex_dump_auxiv:nnn {#4} }
      {
        \exp_args:Ne \__file_hex_dump_auxiv:nnn
          { \int_eval:n { #2 + #4 } }
      }
        {#1} {#3}
  }
\cs_new:Npn \__file_hex_dump_auxiv:nnn #1#2#3
  {
    \tex_filedump:D
      offset ~ \int_eval:n { #2 - 1 } ~
      length ~ \int_eval:n { #1 - #2 + 1 }
      {#3}
  }
\cs_new:Npn \file_hex_dump:n #1
  { \exp_args:Ne \__file_hex_dump:n { \file_full_name:n {#1} } }
\cs_generate_variant:Nn \file_hex_dump:n { V }
```

```
\sys_if_engine_luatex:TF
  {
    \cs_new:Npn \__file_hex_dump:n #1
      {
        \tl_if_blank:nF {#1}
          { \tex_filedump:D whole {#1} {#1} }
      }
  }
  {
    \cs_new:Npn \__file_hex_dump:n #1
      {
        \tl_if_blank:nF {#1}
          { \tex_filedump:D length \tex_filesize:D {#1} {#1} }
      }
  }
```

(*End of definition for* `\file_hex_dump:nnn` *and others. These functions are documented on page 10.*)

`\file_get_hex_dump:nN`
`\file_get_hex_dump:VN`
`\file_get_hex_dump:nNTF`
`\file_get_hex_dump:VNTF`
`\file_get_mdfive_hash:nN`
`\file_get_mdfive_hash:VN`
`\file_get_mdfive_hash:nNTF`
`\file_get_mdfive_hash:VNTF`
`\file_get_size:nN`
`\file_get_size:VN`
`\file_get_size:nNTF`
`\file_get_size:VNTF`
`\file_get_timestamp:nN`
`\file_get_timestamp:VN`
`\file_get_timestamp:nNTF`
`\file_get_timestamp:VNTF`
`\__file_get_details:nnN`

Non-expandable wrappers around the above in the case where appropriate primitive support exists.

```
\cs_new_protected:Npn \file_get_hex_dump:nN #1#2
  { \file_get_hex_dump:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_generate_variant:Nn \file_get_hex_dump:nN { V }
\cs_new_protected:Npn \file_get_mdfive_hash:nN #1#2
  { \file_get_mdfive_hash:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_generate_variant:Nn \file_get_mdfive_hash:nN { V }
\cs_new_protected:Npn \file_get_size:nN #1#2
  { \file_get_size:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_generate_variant:Nn \file_get_size:nN { V }
\cs_new_protected:Npn \file_get_timestamp:nN #1#2
  { \file_get_timestamp:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_generate_variant:Nn \file_get_timestamp:nN { V }
\prg_new_protected_conditional:Npnn \file_get_hex_dump:nN #1#2 { T , F , TF }
  { \__file_get_details:nnN {#1} { hex_dump } #2 }
\prg_generate_conditional_variant:Nnn \file_get_hex_dump:nN
  { V } { T , F , TF }
\prg_new_protected_conditional:Npnn \file_get_mdfive_hash:nN #1#2 { T , F , TF }
  { \__file_get_details:nnN {#1} { mdfive_hash } #2 }
\prg_generate_conditional_variant:Nnn \file_get_mdfive_hash:nN
  { V } { T , F , TF }
\prg_new_protected_conditional:Npnn \file_get_size:nN #1#2 { T , F , TF }
  { \__file_get_details:nnN {#1} { size } #2 }
\prg_generate_conditional_variant:Nnn \file_get_size:nN
  { V } { T , F , TF }
\prg_new_protected_conditional:Npnn \file_get_timestamp:nN #1#2 { T , F , TF }
  { \__file_get_details:nnN {#1} { timestamp } #2 }
\prg_generate_conditional_variant:Nnn \file_get_timestamp:nN
  { V } { T , F , TF }
\cs_new_protected:Npn \__file_get_details:nnN #1#2#3
  {
    \__kernel_tl_set:Ne #3
      { \use:c { file_ #2 :n } {#1} }
    \tl_if_empty:NTF #3
      { \prg_return_false: }
```

```
      { \prg_return_true: }
  }
```

(*End of definition for* `\file_get_hex_dump:nNTF` *and others. These functions are documented on page* 10.)

`\file_get_hex_dump:nnnN`
`\file_get_hex_dump:VnnN`
`\file_get_hex_dump:nnnNTF`
`\file_get_hex_dump:VnnNTF`

Custom code due to the additional arguments.

```
\cs_new_protected:Npn \file_get_hex_dump:nnnN #1#2#3#4
  {
    \file_get_hex_dump:nnnNF {#1} {#2} {#3} #4
      { \tl_set:Nn #4 { \q_no_value } }
  }
\cs_generate_variant:Nn \file_get_hex_dump:nnnN { V }
\prg_new_protected_conditional:Npnn \file_get_hex_dump:nnnN #1#2#3#4
  { T , F , TF }
  {
    \__kernel_tl_set:Ne #4
      { \file_hex_dump:nnn {#1} {#2} {#3} }
    \tl_if_empty:NTF #4
      { \prg_return_false: }
      { \prg_return_true: }
  }
\prg_generate_conditional_variant:Nnn \file_get_hex_dump:nnnN
  { V } { T , F , TF }
```

(*End of definition for* `\file_get_hex_dump:nnnNTF`*. This function is documented on page* 10.)

`\__file_str_cmp:nn`

As we are doing a fixed-length "big" integer comparison, it is easiest to use the low-level behavior of string comparisons.

```
\cs_new_eq:NN \__file_str_cmp:nn \tex_strcmp:D
```

(*End of definition for* `\__file_str_cmp:nn`.)

`\file_compare_timestamp_p:nNn`
`\file_compare_timestamp_p:nNV`
`\file_compare_timestamp_p:VNn`
`\file_compare_timestamp_p:VNV`
`\file_compare_timestamp:nNnTF`
`\file_compare_timestamp:nNVTF`
`\file_compare_timestamp:VNnTF`
`\file_compare_timestamp:VNVTF`
`\__file_compare_timestamp:nnN`
`\__file_timestamp:n`

Comparison of file date can be done by using the low-level nature of the string comparison functions.

```
\prg_new_conditional:Npnn \file_compare_timestamp:nNn #1#2#3
  { p , T , F , TF }
  {
    \exp_args:Nee \__file_compare_timestamp:nnN
      { \file_full_name:n {#1} }
      { \file_full_name:n {#3} }
      #2
  }
\prg_generate_conditional_variant:Nnn \file_compare_timestamp:nNn
  { nNV , V , VNV } { p , T , F , TF }
\cs_new:Npn \__file_compare_timestamp:nnN #1#2#3
  {
    \tl_if_blank:nTF {#1}
      {
        \if_charcode:w #3 <
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      }
```

```
      {
        \tl_if_blank:nTF {#2}
          {
            \if_charcode:w #3 >
              \prg_return_true:
            \else:
              \prg_return_false:
            \fi:
          }
          {
            \if_int_compare:w
              \__file_str_cmp:nn
                { \__file_timestamp:n {#1} }
                { \__file_timestamp:n {#2} }
                #3 \c_zero_int
              \prg_return_true:
            \else:
              \prg_return_false:
            \fi:
          }
      }
  }
\cs_new_eq:NN \__file_timestamp:n \tex_filemoddate:D
```

(*End of definition for* `\file_compare_timestamp:nNnTF`*,* `\__file_compare_timestamp:nnN`*, and* `\__file_timestamp:n`*. This function is documented on page* 12.)

`\file_if_exist_p:n`
`\file_if_exist_p:V`
`\file_if_exist:nTF`
`\file_if_exist:VTF`

The test for the existence of a file is a wrapper around the function to add a path to a file. If the file was found, the path contains something, whereas if the file was not located then the return value is empty.

```
\prg_new_conditional:Npnn \file_if_exist:n #1 { p , T , F , TF }
  {
    \tl_if_blank:eTF { \file_full_name:n {#1} }
      { \prg_return_false: }
      { \prg_return_true: }
  }
\prg_generate_conditional_variant:Nnn \file_if_exist:n { V } { p , T , F , TF }
```

(*End of definition for* `\file_if_exist:nTF`*. This function is documented on page* 10.)

`\file_if_exist_input:n`
`\file_if_exist_input:V`
`\file_if_exist_input:nF`
`\file_if_exist_input:VF`

Input of a file with a test for existence. We do not define the `T` or `TF` variants because the most useful place to place the ⟨*true code*⟩ would be inconsistent with other conditionals.

```
\cs_new_protected:Npn \file_if_exist_input:n #1
  {
    \file_get_full_name:nNT {#1} \l__file_full_name_tl
      { \__file_input:V \l__file_full_name_tl }
  }
\cs_generate_variant:Nn \file_if_exist_input:n { V }
\cs_new_protected:Npn \file_if_exist_input:nF #1#2
  {
    \file_get_full_name:nNTF {#1} \l__file_full_name_tl
      { \__file_input:V \l__file_full_name_tl }
      {#2}
  }
\cs_generate_variant:Nn \file_if_exist_input:nF { V }
```

(*End of definition for* `\file_if_exist_input:n` *and* `\file_if_exist_input:nF`. *These functions are documented on page 13.*)

`\file_input_stop:` A simple rename.

```
\cs_new_protected:Npn \file_input_stop: { \tex_endinput:D }
```

(*End of definition for* `\file_input_stop:`. *This function is documented on page 14.*)

`\__kernel_file_missing:n` An error message for a missing file, also used in `\ior_open:Nn`.

```
\cs_new_protected:Npn \__kernel_file_missing:n #1
  {
    \msg_error:nne { kernel } { file-not-found }
      { \__kernel_file_name_sanitize:n {#1} }
  }
```

(*End of definition for* `\__kernel_file_missing:n`.)

`\file_input:n` `\file_input:V` `\__file_input:n` `\__file_input:V` `\__file_input_push:n` `\__kernel_file_input_push:n` `\__file_input_pop:` `\__kernel_file_input_pop:` `\__file_input_pop:nnn`

Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the `\g__file_stack_seq`, and add it to the file list, either `\g__file_record_seq`, or `\@filelist` in package mode.

```
\cs_new_protected:Npn \file_input:n #1
  {
    \file_get_full_name:nNTF {#1} \l__file_full_name_tl
      { \__file_input:V \l__file_full_name_tl }
      { \__kernel_file_missing:n {#1} }
  }
\cs_generate_variant:Nn \file_input:n { V }
\cs_new_protected:Npe \__file_input:n #1
  {
    \exp_not:N \clist_if_exist:NTF \exp_not:N \@filelist
      { \exp_not:N \@addtofilelist {#1} }
      { \seq_gput_right:Nn \exp_not:N \g__file_record_seq {#1} }
    \exp_not:N \__file_input_push:n {#1}
    \exp_not:N \tex_input:D
    \sys_if_engine_luatex:TF
      { {#1} }
      { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
    \exp_not:N \__file_input_pop:
  }
\cs_generate_variant:Nn \__file_input:n { V }
```

Keeping a track of the file data is easy enough: we store the separated parts so we do not need to parse them twice.

```
\cs_new_protected:Npn \__file_input_push:n #1
  {
    \seq_gpush:Ne \g__file_stack_seq
      {
        { \g_file_curr_dir_str }
        { \g_file_curr_name_str }
        { \g_file_curr_ext_str }
      }
    \file_parse_full_name:nNNN {#1}
      \l__file_dir_str \l__file_name_str \l__file_ext_str
    \str_gset_eq:NN \g_file_curr_dir_str  \l__file_dir_str
    \str_gset_eq:NN \g_file_curr_name_str \l__file_name_str
```

```
    \str_gset_eq:NN \g_file_curr_ext_str  \l__file_ext_str
  }
\cs_new_eq:NN \__kernel_file_input_push:n \__file_input_push:n
\cs_new_protected:Npn \__file_input_pop:
  {
    \seq_gpop:NN \g__file_stack_seq \l__file_internal_tl
    \exp_after:wN \__file_input_pop:nnn \l__file_internal_tl
  }
\cs_new_eq:NN \__kernel_file_input_pop: \__file_input_pop:
\cs_new_protected:Npn \__file_input_pop:nnn #1#2#3
  {
    \str_gset:Nn \g_file_curr_dir_str  {#1}
    \str_gset:Nn \g_file_curr_name_str {#2}
    \str_gset:Nn \g_file_curr_ext_str  {#3}
  }
```

(*End of definition for* `\file_input:n` *and others. This function is documented on page* 13.)

`\file_input_raw:n`
`\file_input_raw:V`
`\__file_input_raw:nn`

No error checking, no tracking.

```
\cs_new:Npn \file_input_raw:n #1
  { \exp_args:Ne \__file_input_raw:nn { \file_full_name:n {#1} } {#1} }
\cs_generate_variant:Nn \file_input_raw:n { V }
\cs_new:Npe \__file_input_raw:nn #1#2
  {
    \exp_not:N \tl_if_blank:nTF {#1}
      {
        \exp_not:N \exp_args:Nnne \exp_not:N \msg_expandable_error:nnn
          { kernel } { file-not-found }
          { \exp_not:N \__kernel_file_name_sanitize:n {#2} }
      }
      {
        \exp_not:N \tex_input:D
          \sys_if_engine_luatex:TF
            { {#1} }
            { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
        }
  }
\exp_args_generate:n { nne }
```

(*End of definition for* `\file_input_raw:n` *and* `\__file_input_raw:nn`. *This function is documented on page* 13.)

`\file_parse_full_name:n`
`\file_parse_full_name:V`
`\file_parse_full_name_apply:nN`
`\file_parse_full_name_apply:VN`

The main parsing macro `\file_parse_full_name_apply:nN` passes the file name `#1` through `\__kernel_file_name_sanitize:n` so that we have a single normalised way to treat files internally. `\file_parse_full_name:n` uses the former, with `\prg_do_nothing:` to leave each part of the name within a pair of braces.

```
\cs_new:Npn \file_parse_full_name:n #1
  {
    \file_parse_full_name_apply:nN {#1}
      \prg_do_nothing:
  }
\cs_generate_variant:Nn \file_parse_full_name:n { V }
\cs_new:Npn \file_parse_full_name_apply:nN #1
  {
```

```
    \exp_args:Ne \__file_parse_full_name_auxi:nN
      { \__kernel_file_name_sanitize:n {#1} }
  }
\cs_generate_variant:Nn \file_parse_full_name_apply:nN { V }
```

`\__file_parse_full_name_area:nw` splits the file name into chunks separated by `/`, until the last one is reached. The last chunk is the file name plus the extension, and everything before that is the path. When `\__file_parse_full_name_area:nw` is done, it leaves the path within braces after the scan mark `\s__file_stop` and proceeds parsing the actual file name.

__file_parse_full_name_auxi:nN
__file_parse_full_name_area:nw

```
\cs_new:Npn \__file_parse_full_name_auxi:nN #1
  {
    \__file_parse_full_name_area:nw { } #1
      / \s__file_stop
  }
\cs_new:Npn \__file_parse_full_name_area:nw #1 #2 / #3 \s__file_stop
  {
    \tl_if_empty:nTF {#3}
      { \__file_parse_full_name_base:nw { } #2 . \s__file_stop {#1} }
      { \__file_parse_full_name_area:nw { #1 / #2 } #3 \s__file_stop }
  }
```

`\__file_parse_full_name_base:nw` does roughly the same as above, but it separates the chunks at each period. However here there's some extra complications: In case `#1` is empty, it is assumed that the extension is actually empty, and the file name is `#2`. Besides, an extra `.` has to be added to `#2` because it is later removed in `\__file_parse_full_name_tidy:nnnN`. In any case, if there's an extension, it is returned with a leading `..`

__file_parse_full_name_base:nw

```
\cs_new:Npn \__file_parse_full_name_base:nw #1 #2 . #3 \s__file_stop
  {
    \tl_if_empty:nTF {#3}
      {
        \tl_if_empty:nTF {#1}
          {
            \tl_if_empty:nTF {#2}
              { \__file_parse_full_name_tidy:nnnN { } { } }
              { \__file_parse_full_name_tidy:nnnN { .#2 } { } }
          }
          { \__file_parse_full_name_tidy:nnnN {#1} { .#2 } }
      }
      { \__file_parse_full_name_base:nw { #1 . #2 } #3 \s__file_stop }
  }
```

Now we just need to tidy some bits left loose before. The loop used in the two macros above start with a leading `/` and `.` in the file path an name, so here we need to remove them, except in the path, if it is a single `/`, in which case it's left as is. After all's done, pass to `#4`.

__file_parse_full_name_tidy:nnnN

```
\cs_new:Npn \__file_parse_full_name_tidy:nnnN #1 #2 #3 #4
  {
    \exp_args:Nee #4
      {
        \str_if_eq:nnF {#3} { / } { \use_none:n }
        #3 \prg_do_nothing:
```

```
      }
      { \use_none:n #1 \prg_do_nothing: }
      {#2}
  }
```

(*End of definition for* `\file_parse_full_name:n` *and others. These functions are documented on page* 13.)

`\file_parse_full_name:nNNN`
`\file_parse_full_name:VNNN`

```
\cs_new_protected:Npn \file_parse_full_name:nNNN #1 #2 #3 #4
  {
    \file_parse_full_name_apply:nN {#1}
      \__file_full_name_assign:nnnNNN #2 #3 #4
  }
\cs_new_protected:Npn \__file_full_name_assign:nnnNNN #1 #2 #3 #4 #5 #6
  {
    \str_set:Nn #4 {#1}
    \str_set:Nn #5 {#2}
    \str_set:Nn #6 {#3}
  }
\cs_generate_variant:Nn \file_parse_full_name:nNNN { V }
```

(*End of definition for* `\file_parse_full_name:nNNN`. *This function is documented on page* 12.)

`\file_show_list:`
`\file_log_list:`
`\__file_list:N`
`\__file_list_aux:n`

A function to list all files used to the log, without duplicates. In package mode, if `\@filelist` is still defined, we need to take this list of file names into account (we capture it `\AtBeginDocument` into `\g__file_record_seq`), turning it to a string (this does not affect the commas of this comma list).

```
\cs_new_protected:Npn \file_show_list: { \__file_list:N \msg_show:nneeee }
\cs_new_protected:Npn \file_log_list: { \__file_list:N \msg_log:nneeee }
\cs_new_protected:Npn \__file_list:N #1
  {
    \seq_clear:N \l__file_tmp_seq
    \clist_if_exist:NT \@filelist
      {
        \exp_args:NNe \seq_set_from_clist:Nn \l__file_tmp_seq
          { \tl_to_str:N \@filelist }
      }
    \seq_concat:NNN \l__file_tmp_seq \l__file_tmp_seq \g__file_record_seq
    \seq_remove_duplicates:N \l__file_tmp_seq
    #1 { kernel } { file-list }
      { \seq_map_function:NN \l__file_tmp_seq \__file_list_aux:n }
        { } { } { }
  }
\cs_new:Npn \__file_list_aux:n #1 { \iow_newline: #1 }
```

(*End of definition for* `\file_show_list:` *and others. These functions are documented on page* 14.)

When used as a package, there is a need to hold onto the standard file list as well as the new one here. File names recorded in `\@filelist` must be turned to strings before being added to `\g__file_record_seq`.

```
\cs_if_exist:NT \@filelist
  {
    \AtBeginDocument
      {
```

```
      \exp_args:NNe \seq_set_from_clist:Nn \l__file_tmp_seq
        { \tl_to_str:N \@filelist }
      \seq_gconcat:NNN
        \g__file_record_seq
        \g__file_record_seq
        \l__file_tmp_seq
    }
}
```

3.5 GetIdInfo

\GetIdInfo
__file_id_info_auxi:w
__file_id_info_auxii:w
__file_id_info_auxiii:w

As documented in expl3.dtx this function extracts file name etc from an SVN `Id` line. This used to be how we got version number and so on in all modules, so it had to be defined in l3bootstrap. Now it's more convenient to define it after we have set up quite a lot of tools, and l3file seems the least unreasonable place for it.

The idea here is to extract out the information needed from a standard SVN `Id` line, but to avoid a line that would get changed when the file is checked in. Hence the fact that none of the lines here include both a dollar sign and the `Id` keyword!

```
\cs_new_protected:Npn \GetIdInfo
  {
    \tl_clear_new:N \ExplFileDescription
    \tl_clear_new:N \ExplFileDate
    \tl_clear_new:N \ExplFileName
    \tl_clear_new:N \ExplFileExtension
    \tl_clear_new:N \ExplFileVersion
    \group_begin:
    \char_set_catcode_space:n { 32 }
    \exp_after:wN
    \group_end:
    \__file_id_info_auxi:w
  }
```

A first check for a completely empty SVN field. If that is not the case, there is a second case when a file created using `svn cp` but has not been checked in. That leaves a special marker `-1` version, which has no further data. Dealing correctly with that is the reason for the space in the line to use `\__file_id_info_auxii:w`.

```
\cs_new_protected:Npn \__file_id_info_auxi:w $ #1 $ #2
  {
    \tl_set:Nn \ExplFileDescription {#2}
    \str_if_eq:nnTF {#1} { Id }
      {
        \tl_set:Nn \ExplFileDate { 0000/00/00 }
        \tl_set:Nn \ExplFileName { [unknown] }
        \tl_set:Nn \ExplFileExtension { [unknown~extension] }
        \tl_set:Nn \ExplFileVersion {-1}
      }
      { \__file_id_info_auxii:w #1 ~ \s__file_stop }
  }
```

Here, `#1` is `Id`, `#2` is the file name, `#3` is the extension, `#4` is the version, `#5` is the check in date and `#6` is the check in time and user, plus some trailing spaces. If `#4` is the marker `-1` value then `#5` and `#6` are empty.

```
\cs_new_protected:Npn \__file_id_info_auxii:w
```

```
    #1 ~ #2.#3 ~ #4 ~ #5 ~ #6 \s__file_stop
  {
    \tl_set:Nn \ExplFileName {#2}
    \tl_set:Nn \ExplFileExtension {#3}
    \tl_set:Nn \ExplFileVersion {#4}
    \str_if_eq:nnTF {#4} {-1}
      { \tl_set:Nn \ExplFileDate { 0000/00/00 } }
      { \__file_id_info_auxiii:w #5 - 0 - 0 - \s__file_stop }
  }
```

Convert an SVN-style date into a LaTeX-style one.

```
\cs_new_protected:Npn \__file_id_info_auxiii:w #1 - #2 - #3 - #4 \s__file_stop
  { \tl_set:Nn \ExplFileDate { #1/#2/#3 } }
```

(*End of definition for* `\GetIdInfo` *and others. This function is documented on page* ??.)

3.6 Checking the version of kernel dependencies

`\__kernel_dependency_version_check:Nn`
`\__kernel_dependency_version_check:nn`
`\__file_kernel_dependency_compare:nnn`
`\__file_parse_version:w`

This function is responsible for checking if dependencies of the LaTeX3 kernel match the version preloaded in the LaTeX 2_ε kernel. If versions don't match, the function attempts to tell why by searching for a possible stray format file.

The function starts by checking that the kernel date is defined, and if not zero is used to force the error route. The kernel date is then compared with the argument requested date (ususally the packaging date of the dependency). If the kernel date is less than the required date, it's an error and the loading should abort.

```
\cs_new_protected:Npn \__kernel_dependency_version_check:Nn #1
  { \exp_args:NV \__kernel_dependency_version_check:nn #1 }
\cs_new_protected:Npn \__kernel_dependency_version_check:nn #1
  {
    \cs_if_exist:NTF \c__kernel_expl_date_tl
      {
        \exp_args:NV \__file_kernel_dependency_compare:nnn
          \c__kernel_expl_date_tl {#1}
      }
      { \__file_kernel_dependency_compare:nnn { 0000-00-00 } {#1} }
  }
\cs_new_protected:Npn \__file_kernel_dependency_compare:nnn #1 #2 #3
  {
    \int_compare:nNnT
        { \__file_parse_version:w #1 \s__file_stop } <
        { \__file_parse_version:w #2 \s__file_stop }
      { \__file_mismatched_dependency_error:nn {#2} {#3} }
  }
\cs_new:Npn \__file_parse_version:w #1 - #2 - #3 \s__file_stop {#1#2#3}
```

If the versions differ, then we try to give the user some guidance. This function starts by taking the engine name `\c_sys_engine_str` and replacing `tex` by `latex`, then building a command of the form: `kpsewhich` –all –engine=⟨*engine*⟩ ⟨*format*⟩[-dev].fmt to query the format files available. A shell is opened and each line is read into a sequence.

`\__file_mismatched_dependency_error:nn`

```
\cs_new_protected:Npn \__file_mismatched_dependency_error:nn #1 #2
  {
    \exp_args:NNe \ior_shell_open:Nn \g__file_internal_ior
      {
```

```
        kpsewhich ~ --all ~
          --engine = \c_sys_engine_exec_str
          \c_space_tl \c_sys_engine_format_str
            \bool_lazy_and:nnT
                { \tl_if_exist_p:N \development@branch@name }
                { ! \tl_if_empty_p:N \development@branch@name }
              { -dev } .fmt
      }
    \seq_clear:N \l__file_tmp_seq
    \ior_map_inline:Nn \g__file_internal_ior
      { \seq_put_right:Nn \l__file_tmp_seq {##1} }
    \ior_close:N \g__file_internal_ior
    \msg_error:nnnn { kernel } { mismatched-support-file }
      {#1} {#2}
```

And finish by ending the current file.

```
    \tex_endinput:D
  }
```

Now define the actual error message:

```
\msg_new:nnnn { kernel } { mismatched-support-file }
  {
    Mismatched~LaTeX~support~files~detected. \\
    Loading~'#2'~aborted!
```

`\c__kernel_expl_date_tl` may not exist, due to an older format, so only print the dates when the sentinel token list exists:

```
    \tl_if_exist:NT \c__kernel_expl_date_tl
      {
        \\ \\
        The~L3~programming~layer~in~the~LaTeX~format \\
        is~dated~\c__kernel_expl_date_tl,~but~in~your~TeX~
        tree~the~files~require \\ at~least~#1.
      }
  }
  {
```

The sequence containing the format files should have exactly one item: the format file currently being run. If that's the case, the cause of the error is not that, so print a generic help with some possible causes. If more than one format file was found, then print the list to the user, with appropriate indications of what's in the system and what's in the user tree.

```
    \int_compare:nNnTF { \seq_count:N \l__file_tmp_seq } > 1
      {
        The~cause~seems~to~be~an~old~format~file~in~the~user~tree. \\
        LaTeX~found~these~files:
        \seq_map_tokens:Nn \l__file_tmp_seq { \\~--~\use:n } \\
        Try~deleting~the~file~in~the~user~tree~then~run~LaTeX~again.
      }
      {
        The~most~likely~causes~are:
        \\~--~A~recent~format~generation~failed;
        \\~--~A~stray~format~file~in~the~user~tree~which~needs~
             to~be~removed~or~rebuilt;
        \\~--~You~are~running~a~manually~installed~version~of~#2 \\
```

```
        \ \ \ which~is~incompatible~with~the~version~in~LaTeX. \\
      }
    \\
    LaTeX~will~abort~loading~the~incompatible~support~files~
    but~this~may~lead~to \\ later~errors.~Please~ensure~that~
    your~LaTeX~format~is~correctly~regenerated.
  }
```

(*End of definition for* `\__kernel_dependency_version_check:Nn` *and others.*)

3.7 Messages

```
\msg_new:nnnn { kernel } { file-not-found }
  { File~'#1'~not~found. }
  {
    The~requested~file~could~not~be~found~in~the~current~directory,~
    in~the~TeX~search~path~or~in~the~LaTeX~search~path.
  }
\msg_new:nnn { kernel } { file-list }
  {
    >~File~List~<
    #1 \\
    ..............
  }
\msg_new:nnnn { kernel } { filename-chars-lost }
  { #1~invalid~in~file~name.~Lost:~#2. }
  {
    There~was~an~invalid~token~in~the~file~name~that~caused~
    the~characters~following~it~to~be~lost.
  }
\msg_new:nnnn { kernel } { filename-missing-endcsname }
  { Missing~\iow_char:N\\endcsname~inserted~in~filename. }
  {
    The~file~name~had~more~\iow_char:N\\csname~commands~than~
    \iow_char:N\\endcsname~ones.~LaTeX~will~add~the~missing~
    \iow_char:N\\endcsname~and~try~to~continue~as~best~as~it~can.
  }
\msg_new:nnnn { kernel } { unbalanced-quote-in-filename }
  { Unbalanced~quotes~in~file~name~'#1'. }
  {
    File~names~must~contain~balanced~numbers~of~quotes~(").
  }
\msg_new:nnnn { kernel } { iow-indent }
  { Only~#1 allows~#2 }
  {
    The~command~#2 can~only~be~used~in~messages~
    which~will~be~wrapped~using~#1.
    \tl_if_empty:nF {#3} { ~ It~was~called~with~argument~'#3'. }
  }
```

3.8 Functions delayed from earlier modules

<@@=sys>

\c_sys_platform_str

Detecting the platform on LuaTEX is easy: for other engines, we use the fact that the two common cases have special null files. It is possible to probe further (see package platform), but that requires shell escape and seems unlikely to be useful. This is set up here as it requires file searching.

```
\sys_if_engine_luatex:TF
  {
    \str_const:Ne \c_sys_platform_str
      { \tex_directlua:D { tex.print(os.type) } }
  }
  {
    \file_if_exist:nTF { nul: }
      {
        \file_if_exist:nF { /dev/null }
          { \str_const:Nn \c_sys_platform_str { windows } }
      }
      {
        \file_if_exist:nT { /dev/null }
          { \str_const:Nn \c_sys_platform_str { unix } }
      }
  }
\cs_if_exist:NF \c_sys_platform_str
  { \str_const:Nn \c_sys_platform_str { unknown }  }
```

(*End of definition for* `\c_sys_platform_str`. *This variable is documented on page* ??.)

\sys_if_platform_unix_p:
\sys_if_platform_unix:*TF*
\sys_if_platform_windows_p:
\sys_if_platform_windows:*TF*

We can now set up the tests.

```
\clist_map_inline:nn { unix , windows }
  {
    \__file_const:nn { sys_if_platform_ #1 }
      { \str_if_eq_p:Vn \c_sys_platform_str { #1 } }
  }
```

(*End of definition for* `\sys_if_platform_unix:TF` *and* `\sys_if_platform_windows:TF`. *These functions are documented on page* ??.)

⟨/package⟩

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.

C

E

F

G

group commands:

I

if commands:

int commands:

ior commands:

ior internal commands:

K

L

M

O

P

Q

S

T

U