NAME

File::Slurp − Simple and Efficient Reading/Writing/Modifying of Complete Files

SYNOPSIS

```
  use File::Slurp;

  # read in a whole file into a scalar
  my $text = read_file('/path/file');

  # read in a whole file into an array of lines
  my @lines = read_file('/path/file');

  # write out a whole file from a scalar
  write_file('/path/file', $text);

  # write out a whole file from an array of lines
  write_file('/path/file', @lines);

  # Here is a simple and fast way to load and save a simple config file
  # made of key=value lines.
  my %conf = read_file('/path/file') =~ /^(\w+)=(.*)$/mg;
  write_file('/path/file', {atomic => 1}, map "$_=$conf{$_}\n", keys %conf);

  # insert text at the beginning of a file
  prepend_file('/path/file', $text);

  # in-place edit to replace all 'foo' with 'bar' in file
  edit_file { s/foo/bar/g } '/path/file';

  # in-place edit to delete all lines with 'foo' from file
  edit_file_lines sub { $_ = '' if /foo/ }, '/path/file';

  # read in a whole directory of file names (skipping . and ..)
  my @files = read_dir('/path/to/dir');
```

DESCRIPTION

This module provides subs that allow you to read or write entire files with one simple call. They are designed to be simple to use, have flexible ways to pass in or get the file contents and to be very efficient. There is also a sub to read in all the files in a directory.

WARNING − PENDING DOOM

Although you technically *can*, do NOT use this module to work on file handles, pipes, sockets, standard IO, or the `DATA` handle. These are features implemented long ago that just really shouldn't be abused here.

Be warned: this activity will lead to inaccurate encoding/decoding of data.

All further mentions of actions on the above have been removed from this documentation and that feature set will likely be deprecated in the future.

In other words, if you don't have a filename to pass, consider using the standard `do { local $/; <$fh> }`, or Data::Section/Data::Section::Simple for working with `__DATA__`.

FUNCTIONS

File::Slurp implements the following functions.

append_file

```
    use File::Slurp qw(append_file write_file);
    my $res = append_file('/path/file', "Some text");
    # same as
    my $res = write_file('/path/file', {append => 1}, "Some text");
```

The `append_file` function is simply a synonym for the "write_file" in File::Slurp function, but ensures that the `append` option is set.

edit_file

```
    use File::Slurp qw(edit_file);
    # perl -0777 -pi -e 's/foo/bar/g' /path/file
    edit_file { s/foo/bar/g } '/path/file';
    edit_file sub { s/foo/bar/g }, '/path/file';
    sub replace_foo { s/foo/bar/g }
    edit_file \&replace_foo, '/path/file';
```

The `edit_file` function reads in a file into `$_`, executes a code block that should modify `$_`, and then writes `$_` back to the file. The `edit_file` function reads in the entire file and calls the code block one time. It is equivalent to the `-pi` command line options of Perl but you can call it from inside your program and not have to fork out a process.

The first argument to `edit_file` is a code block or a code reference. The code block is not followed by a comma (as with `grep` and `map`) but a code reference is followed by a comma.

The next argument is the filename.

The next argument(s) is either a hash reference or a flattened hash, `key => value` pairs. The options are passed through to the "write_file" in File::Slurp function. All options are described there. Only the `binmode` and `err_mode` options are supported. The call to "write_file" in File::Slurp has the `atomic` option set so you will always have a consistent file.

edit_file_lines

```
    use File::Slurp qw(edit_file_lines);
    # perl -pi -e '$_ = "" if /foo/' /path/file
    edit_file_lines { $_ = '' if /foo/ } '/path/file';
    edit_file_lines sub { $_ = '' if /foo/ }, '/path/file';
    sub delete_foo { $_ = '' if /foo/ }
    edit_file \&delete_foo, '/path/file';
```

The `edit_file_lines` function reads each line of a file into `$_`, and executes a code block that should modify `$_`. It will then write `$_` back to the file. It is equivalent to the `-pi` command line options of Perl but you can call it from inside your program and not have to fork out a process.

The first argument to `edit_file_lines` is a code block or a code reference. The code block is not followed by a comma (as with `grep` and `map`) but a code reference is followed by a comma.

The next argument is the filename.

The next argument(s) is either a hash reference or a flattened hash, `key => value` pairs. The options are passed through to the "write_file" in File::Slurp function. All options are described there. Only the `binmode` and `err_mode` options are supported. The call to "write_file" in File::Slurp has the `atomic` option set so you will always have a consistent file.

ef

```
    use File::Slurp qw(ef);
    # perl -0777 -pi -e 's/foo/bar/g' /path/file
    ef { s/foo/bar/g } '/path/file';
    ef sub { s/foo/bar/g }, '/path/file';
    sub replace_foo { s/foo/bar/g }
    ef \&replace_foo, '/path/file';
```

The `ef` function is simply a synonym for the "edit_file" in File::Slurp function.

efl

```
use File::Slurp qw(efl);
# perl -pi -e '$_ = "" if /foo/' /path/file
efl { $_ = '' if /foo/ } '/path/file';
efl sub { $_ = '' if /foo/ }, '/path/file';
sub delete_foo { $_ = '' if /foo/ }
efl \&delete_foo, '/path/file';
```

The `efl` function is simply a synonym for the "edit_file_lines" in File::Slurp function.

overwrite_file

```
use File::Slurp qw(overwrite_file);
my $res = overwrite_file('/path/file', "Some text");
```

The `overwrite_file` function is simply a synonym for the "write_file" in File::Slurp function.

prepend_file

```
use File::Slurp qw(prepend_file);
prepend_file('/path/file', $header);
prepend_file('/path/file', \@lines);
prepend_file('/path/file', { binmode => ':raw'}, $bin_data);

# equivalent to:
use File::Slurp qw(read_file write_file);
my $content = read_file('/path/file');
my $new_content = "hahahaha";
write_file('/path/file', $new_content . $content);
```

The `prepend_file` function is the opposite of "append_file" in File::Slurp as it writes new contents to the beginning of the file instead of the end. It is a combination of "read_file" in File::Slurp and "write_file" in File::Slurp. It works by first using `read_file` to slurp in the file and then calling `write_file` with the new data and the existing file data.

The first argument to `prepend_file` is the filename.

The next argument(s) is either a hash reference or a flattened hash, `key => value` pairs. The options are passed through to the "write_file" in File::Slurp function. All options are described there.

Only the `binmode` and `err_mode` options are supported. The `write_file` call has the `atomic` option set so you will always have a consistent file.

read_dir

```
use File::Slurp qw(read_dir);
my @files = read_dir('/path/to/dir');
# all files, even the dots
my @files = read_dir('/path/to/dir', keep_dot_dot => 1);
# keep the full file path
my @paths = read_dir('/path/to/dir', prefix => 1);
# scalar context
my $files_ref = read_dir('/path/to/dir');
```

This function returns a list of the filenames in the supplied directory. In list context, an array is returned, in scalar context, an array reference is returned.

The first argument is the path to the directory to read.

The next argument(s) is either a hash reference or a flattened hash, `key => value` pairs. The following options are available:

- err_mode

  The `err_mode` option has three possible values: `quiet`, `carp`, or the default, `croak`. In `quiet` mode, all errors will be silent. In `carp` mode, all errors will be emitted as warnings. And, in `croak` mode, all errors will be emitted as exceptions. Take a look at Try::Tiny or Syntax::Keyword::Try to see how to catch exceptions.

- keep_dot_dot

  The `keep_dot_dot` option is a boolean option, defaulted to false (`0`). Setting this option to true (`1`) will also return the `.` and `..` files that are removed from the file list by default.

- prefix

  The `prefix` option is a boolean option, defaulted to false (`0`). Setting this option to true (`1`) add the directory as a prefix to the file. The directory and the filename are joined using `File::Spec->catfile()` to ensure the proper directory separator is used for your OS. See File::Spec.

read_file

```
use File::Slurp qw(read_file);
my $text = read_file('/path/file');
my $bin = read_file('/path/file', { binmode => ':raw' });
my @lines = read_file('/path/file');
my $lines_ref = read_file('/path/file', array_ref => 1);
my $lines_ref = [ read_file('/path/file') ];

# or we can read into a buffer:
my $buffer;
read_file('/path/file', buf_ref => \$buffer);

# or we can set the block size for the read
my $text_ref = read_file('/path/file', blk_size => 10_000_000, array_ref

# or we can get a scalar reference
my $text_ref = read_file('/path/file', scalar_ref => 1);
```

This function reads in an entire file and returns its contents to the caller. In scalar context it returns the entire file as a single scalar. In list context it will return a list of lines (using the current value of `$/` as the separator, including support for paragraph mode when it is set to `''`).

The first argument is the path to the file to be slurped in.

The next argument(s) is either a hash reference or a flattened hash, `key => value` pairs. The following options are available:

- array_ref

  The `array_ref` option is a boolean option, defaulted to false (`0`). Setting this option to true (`1`) will only have relevance if the `read_file` function is called in scalar context. When true, the `read_file` function will return a reference to an array of the lines in the file.

- binmode

  The `binmode` option is a string option, defaulted to empty (`''`). If you set the `binmode` option, then its value is passed to a call to `binmode` on the opened handle. You can use this to set the file to be read in binary mode, utf8, etc. See `perldoc -f binmode` for more.

  Please note that using binmode :utf8 with sysread (and thus read_file) has been deprecated in recent versions of perl.

- blk_size

  You can use this option to set the block size used when slurping from an already open handle (like `\*STDIN`). It defaults to 1MB.

- buf_ref

  The `buf_ref` option can be used in conjunction with any of the other options. You can use this option to pass in a scalar reference and the slurped file contents will be stored in the scalar. This saves an extra copy of the slurped file and can lower RAM usage vs returning the file. It is usually the fastest way to read a file into a scalar.

- chomp

  The `chomp` option is a boolean option, defaulted to false (`0`). Setting this option to true (`1`) will cause each line to have its contents `chomp`ed. This option works in list context or in scalar context with the `array_ref` option.

- err_mode

  The `err_mode` option has three possible values: `quiet`, `carp`, or the default, `croak`. In `quiet` mode, all errors will be silent. In `carp` mode, all errors will be emitted as warnings. And, in `croak` mode, all errors will be emitted as exceptions. Take a look at Try::Tiny or Syntax::Keyword::Try to see how to catch exceptions.

- scalar_ref

  The `scalar_ref` option is a boolean option, defaulted to false (`0`). It only has meaning in scalar context. The return value will be a scalar reference to a string which is the contents of the slurped file. This will usually be faster than returning the plain scalar. It will also save memory as it will not make a copy of the file to return.

rf

```
use File::Slurp qw(rf);
my $text = rf('/path/file');
```

The `rf` function is simply a synonym for the "read_file" in File::Slurp function.

slurp

```
use File::Slurp qw(slurp);
my $text = slurp('/path/file');
```

The `slurp` function is simply a synonym for the "read_file" in File::Slurp function.

wf

```
use File::Slurp qw(wf);
my $res = wf('/path/file', "Some text");
```

The `wf` function is simply a synonym for the "write_file" in File::Slurp function.

write_file

```
use File::Slurp qw(write_file);
write_file('/path/file', @data);
write_file('/path/file', {append => 1}, @data);
write_file('/path/file', {binmode => ':raw'}, $buffer);
write_file('/path/file', \$buffer);
write_file('/path/file', $buffer);
write_file('/path/file', \@lines);
write_file('/path/file', @lines);

# binmode
write_file('/path/file', {binmode => ':raw'}, @data);
write_file('/path/file', {binmode => ':utf8'}, $utf_text);

# buffered
write_file('/path/file', {buf_ref => \$buffer});
write_file('/path/file', \$buffer);
write_file('/path/file', $buffer);

# append
write_file('/path/file', {append => 1}, @data);

# no clobbering
write_file('/path/file', {no_clobber => 1}, @data);
```

This function writes out an entire file in one call. By default `write_file` returns `1` upon successfully writing the file or `undef` if it encountered an error. You can change how errors are handled with the `err_mode` option.

The first argument to `write_file` is the filename.

The next argument(s) is either a hash reference or a flattened hash, `key => value` pairs. The following options are available:

- append

  The `append` option is a boolean option, defaulted to false (`0`). Setting this option to true (`1`) will cause the data to be be written at the end of the current file. Internally this sets the `sysopen` mode flag `O_APPEND`.

  The ''append_file'' in File::Slurp function sets this option by default.

- atomic

  The `atomic` option is a boolean option, defaulted to false (`0`). Setting this option to true (`1`) will cause the file to be be written to in an atomic fashion. A temporary file name is created using ''tempfile'' in File::Temp. After the file is closed it is renamed to the original file name (and`rename` is an atomic operation on most OSes). If the program using this were to crash in the middle of this, then the temporary file could be left behind.

- binmode

  The `binmode` option is a string option, defaulted to empty (`''`). If you set the `binmode` option, then its value is passed to a call to `binmode` on the opened handle. You can use this to set the file to be read in binary mode, utf8, etc. See `perldoc -f binmode` for more.

- buf_ref

  The `buf_ref` option is used to pass in a scalar reference which has the data to be written. If this is set then any data arguments (including the scalar reference shortcut) in `@_` will be ignored.

- err_mode

  The `err_mode` option has three possible values: `quiet`, `carp`, or the default, `croak`. In `quiet` mode, all errors will be silent. In `carp` mode, all errors will be emitted as warnings. And, in `croak` mode, all errors will be emitted as exceptions. Take a look at Try::Tiny or Syntax::Keyword::Try to see how to catch exceptions.

- no_clobber

  The `no_clobber` option is a boolean option, defaulted to false (`0`). Setting this option to true (`1`) will ensure an that existing file will not be overwritten.

- perms

  The `perms` option sets the permissions of newly-created files. This value is modified by your process's `umask` and defaults to `0666` (same as `sysopen`).

  NOTE: this option is new as of File::Slurp version 9999.14.

EXPORT

These are exported by default or with

```
        use File::Slurp qw(:std);
        # read_file write_file overwrite_file append_file read_dir
```

These are exported with

```
        use File::Slurp qw(:edit);
        # edit_file edit_file_lines
```

You can get all subs in the module exported with

```
        use File::Slurp qw(:all);
```

SEE ALSO

- File::Slurper − Provides a straightforward set of functions for the most common tasks of reading/writing text and binary files.
- Path::Tiny − Lightweight and comprehensive file handling, including simple methods for reading, writing, and editing text and binary files.
- Mojo::File − Similar to Path::Tiny for the Mojo toolkit, always works in bytes.

AUTHOR

Uri Guttman, <*uri@stemsystems.com*>

COPYRIGHT & LICENSE