

ReNamer

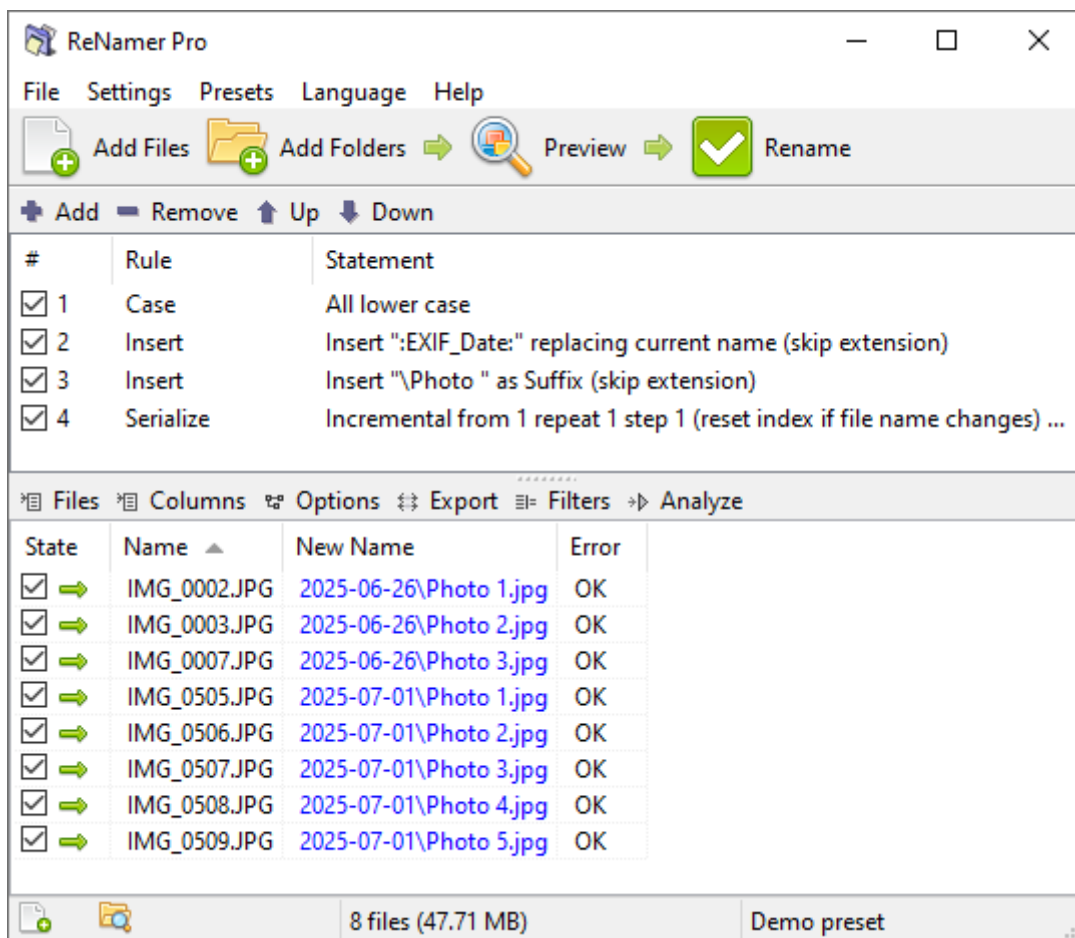
Documentation for [ReNamer](#), a versatile file and folder renaming utility.

- [Overview](#)
- [Quick Guide](#)
- [Getting Started](#)
 - [Adding files and folders](#)
 - [Managing rules](#)
 - [Previewing files](#)
 - [Renaming files](#)
- [Rules](#)
 - [Overview of Rules](#)
 - [Insert rule](#)
 - [Delete rule](#)
 - [Remove rule](#)
 - [Replace rule](#)
 - [Rearrange rule](#)
 - [Extension rule](#)
 - [Strip rule](#)
 - [Case rule](#)
 - [Serialize rule](#)
 - [Randomize rule](#)
 - [Padding rule](#)
 - [Clean Up rule](#)
 - [Translit rule](#)
 - [Reformat Date rule](#)
 - [Regular Expressions rule](#)
 - [Pascal Script rule](#)
 - [User Input rule](#)

- [Mapping rule](#)
- [Operational Guides](#)
 - [Using presets](#)
 - [Presets Manager](#)
 - [Analyze tool](#)
 - [Meta Tags](#)
 - [Date and Time format](#)
 - [Sorting files](#)
 - [Renaming folders](#)
 - [Moving files to another folder](#)
 - [Validation](#)
 - [Failed renaming](#)
 - [Manual editing](#)
 - [Command line](#)
 - [Long paths](#)
 - [Wildcard masks](#)
 - [Regular Expressions reference](#)
- [Settings and Menus](#)
 - [Main settings](#)
 - [Main menu](#)
 - [Rules menu](#)
 - [Files and Tools](#)
 - [Files context menu](#)
 - [Files header menu](#)
 - [Filter settings](#)
 - [Export menu](#)
 - [Options menu](#)
- [Pascal Script](#)
 - [Pascal Script: Overview](#)
 - [Pascal Script: Basics](#)
 - [Pascal Script: Types](#)
 - [Pascal Script: Functions](#)

- [Pascal Script: Examples](#)
- [Pascal Script: Library](#)

Overview



[ReNameR](#) is a powerful and flexible file renaming utility. It goes well beyond simple find-and-replace, letting you build a precise sequence of [rules](#) that are applied to your files in order, giving you full control over the final result.

Standard operations cover all common renaming needs: inserting or removing text, replacing content, changing case, adjusting file extensions, adding sequential numbers, cleaning up filenames, and much more. Rules can be combined freely and saved as named [presets](#) to reuse at any time, making repetitive renaming tasks effortless.

For renaming based on file content, a rich set of [meta tags](#) is supported, covering music tags (artist, album, track number), photo metadata (camera model, date taken), image dimensions, video information, document properties (title, author, page count), and file checksums. This allows you to build filenames directly from the content of your files.

Advanced users can take things further with [Regular Expressions](#) for pattern-based renaming, or the [Pascal Script rule](#) for fully custom renaming logic and integration with external tools. Full Unicode support ensures reliable handling of non-English filenames, and folders can be renamed

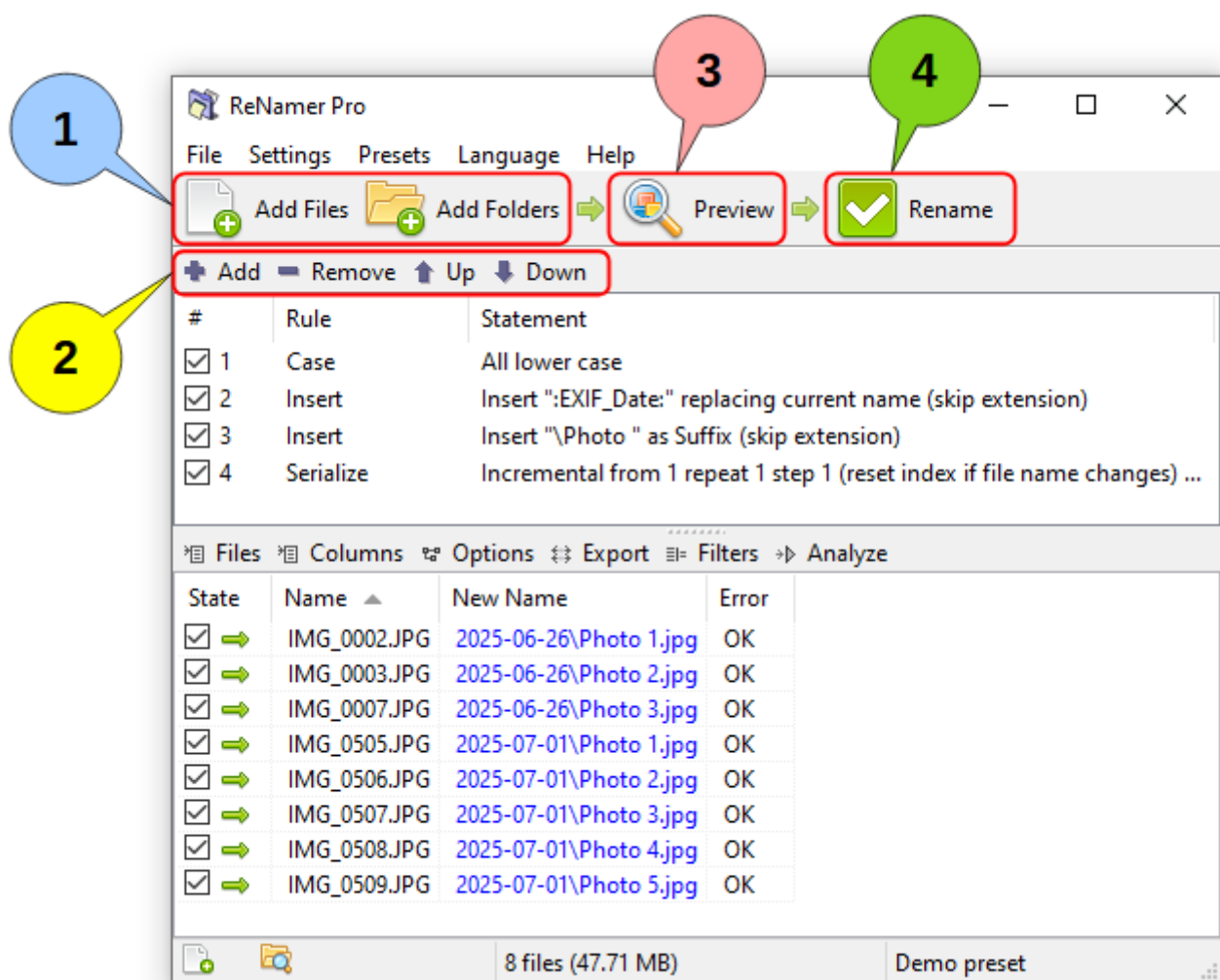
alongside files.

Quick Guide

The workflow in ReNamer is a simple 4-step process:

1. [Add files or folders](#) for renaming.
2. [Add renaming rules](#) for generating new names.
3. [Preview](#) the results and address any warnings.
4. [Rename](#).

Let's briefly go through the main steps below.



Step 1: Add files

Add individual files or whole folders for renaming using the *Add Files* and *Add Folders* buttons, or by dragging and dropping from Windows Explorer (or any other application) into the ReNamer

window.

Optionally, adjust the [filter settings](#) to control which files and folders get added, including depth criteria and filter masks.

The toolbar above the **Files pane** and the right-click context menu offer additional options for managing files, including sorting, marking, deleting, importing, and exporting.

Step 2: Add rules

[Add and arrange rules](#) to build a sequence of operations for generating new names. Rules can be managed via the toolbar, context menu, or keyboard shortcuts.

Rule sets can be saved as [presets](#) for future reuse.

Step 3: Preview

Generate the new names and review the results before renaming. Address any [validation](#) warnings.

By default, the [auto preview](#) option is enabled, which automatically regenerates new names whenever files or rules change, so pressing the *Preview* button manually is optional. Additional [preview settings](#) offer features like highlighting changed names and resolving conflicts.

Step 4: Rename

Press the *Rename* button to apply the new names to your files and folders. Files are renamed exactly as shown in the preview.

If needed, the operation can be undone via *File » Undo Renaming*.

Getting Started

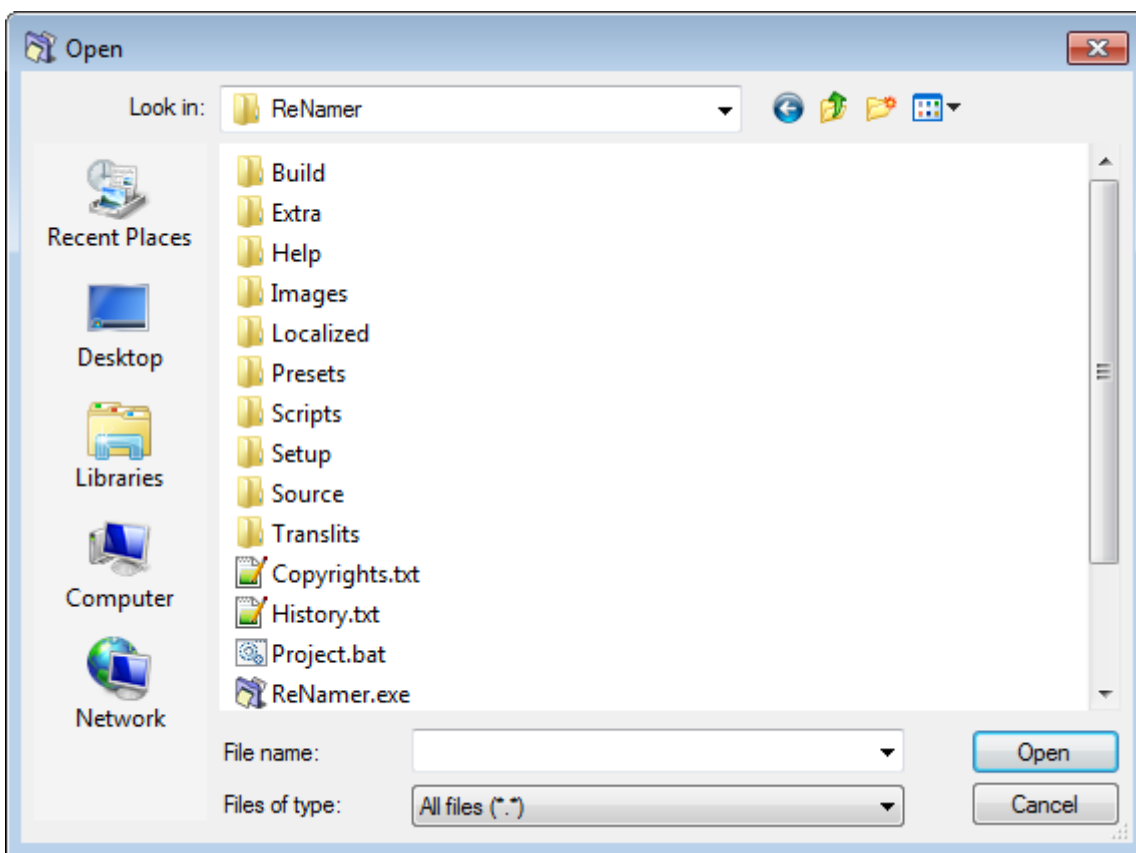
Learn the core workflow: adding files, setting up rules, previewing, and renaming.

Adding files and folders

This page explains how to populate and manage the **Files pane**, the central working area where files and folders are loaded for renaming.

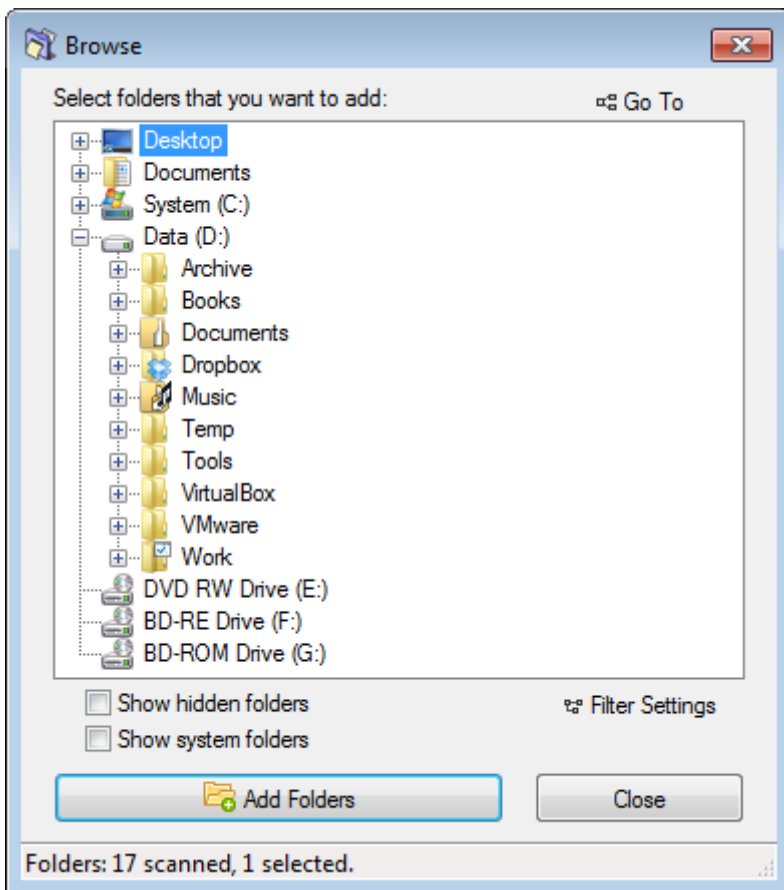
The *Add Files* and *Add Folders* buttons on the main toolbar are the most common ways to load items, though several other methods are available and described below.

Adding files using the Add Files dialog



1. Click the *Add Files* button. The file open dialog appears.
2. Navigate to the required folder and select the desired files.
3. Click *OK*. The selected files are added to the **Files pane**.
4. Repeat to load files from additional folders as needed.

Adding files using the Add Folders dialog



1. Click the *Add Folders* button. The folder browse dialog appears.
 - You can select multiple folders by holding `Ctrl` and clicking each one.
2. By default, the contents of each selected folder are added, not the folder itself.
 - To change what gets added, such as including folder entries or limiting by file type, adjust the [Filter settings](#) before confirming.
3. Select the desired folders and click *Add Folders* to confirm.

Adding files using the Add Paths dialog

The *Add Paths* option in the [main menu](#) opens a dialog where file and folder paths can be entered or pasted directly as text.

Which items are added from each path follows the same [Filter settings](#) as the *Add Folders* dialog. Unlike the standard dialogs, *Add Paths* also supports the long path specification (`\\?\`), making it the appropriate method when working with paths that exceed the Windows 260-character limit.

See [Long paths](#) for more information.

Adding files using drag and drop

Select files in any application and drag them into the **Files pane**.

If the window is obscured, drag the selection onto its taskbar button and pause briefly until the window comes to the foreground. You can also enable the [Stay on top](#) option to keep the window above all other applications at all times.

Adding files using copy and paste

Select files in any application and press `Ctrl+C`, then switch to ReNamer and press `Ctrl+Shift+V`.

Note that the paste shortcut is `Ctrl+Shift+V`, not the standard `Ctrl+V`, and the **Files pane** does not need to be focused for it to work.

Adding files by importing a list

The [Export menu](#) in the **Files pane** toolbar offers two options for loading files from a previously saved list:

- *Import files from text-list or play-list* loads file paths from a plain text list (one path per line) or a playlist file in M3U or PLS format.
- *Import file paths and new names* loads file paths together with predefined new names from a CSV or tab-separated TXT file.

Removing files from the pane

Select the items to remove and press `Del`. This only removes them from the **Files pane**. Files and folders on disk are not affected.

Ordering files in the pane

Some rules, such as the [Serialize rule](#), process files in top-to-bottom order, so position in the list matters. By default, files appear in the order they were added, with newer additions at the bottom.

To reposition files manually, click and drag a file to the desired location. Multiple files can be dragged as a group. To sort by a column automatically, click its column header. A small triangle indicates the active sort direction. See [Sorting files](#) for more detail.

Marking and unmarking files

Marking is independent of selection. Only *marked* files are previewed and renamed. *Unmarked* files are skipped entirely.

Unmarking a file is a convenient way to exclude it from renaming without removing it from the pane.

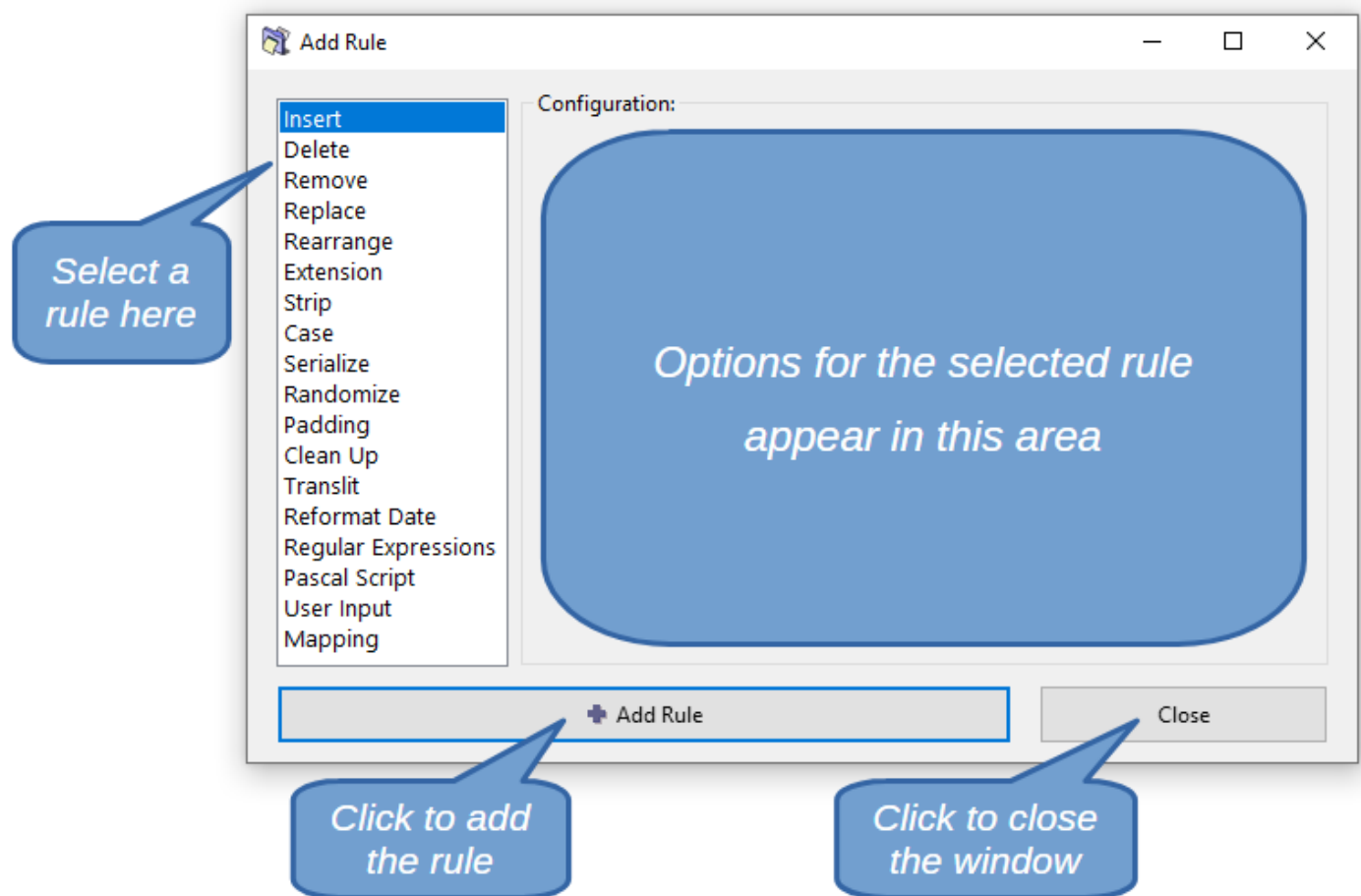
To toggle the *marked* state of a file, click its checkbox, or select the file and press Spacebar.

Managing rules

This page explains how to add, remove, edit, and reorder rules in the **Rules pane**.

Most actions for managing rules are available via the toolbar above the pane, the right-click context menu, or keyboard shortcuts. See the [Rules menu](#) article for a full reference.

Adding rules



To open the *Add Rule* dialog, click the *Add Rule* button in the toolbar or press the `Insert` key. You can also right-click in the **Rules pane** and select *Add Rule*, *Add Rule (above)*, or *Add Rule (below)* to control where the new rule is inserted.

Select the desired rule type from the list on the left. Its options appear immediately in the configuration area on the right. Set the parameters as needed, then click *Add Rule* to add it to the stack. To add another rule, repeat the process. Click *Close* to dismiss the dialog without adding a rule.

To duplicate an existing rule, select it and press `Shift+Ins`, or use *Duplicate Rule* from the context menu. The duplicate is inserted immediately below the original.

Removing rules

Select a rule and press `Del`, or use *Remove Rule* from the context menu.

To remove all rules at once, press `Shift+Del` or use *Remove All Rules*.

Reordering rules

Rules are applied to each file in the order they appear in the pane. The same set of rules can produce very different results depending on their order.

To reposition a rule, use any of the following:

- Click the *Move Up* or *Move Down* buttons in the toolbar.
- Press `Ctrl+Up` or `Ctrl+Down`.
- Drag the rule to the desired position.

Editing rules

To edit an existing rule, use any of the following:

- Double-click it.
- Select it and press `Enter`.
- Right-click it and select *Edit Rule*.

The rule's configuration dialog opens, pre-populated with the current settings. Adjust the parameters and click *Save Rule* to apply the changes.

Marking and unmarking rules

Marking is independent of selection. Only *marked* rules are applied during preview and renaming. *Unmarked* rules are skipped entirely.

Unmarking a rule is a convenient way to temporarily disable it without removing it from the pane.

To toggle the *marked* state of a rule, click its checkbox or select the rule and press `Spacebar`. To mark or unmark all rules at once, use *Mark All* (`Shift+M`) or *Unmark All* (`Shift+U`) from the context menu.

Annotating rules

Any rule can be annotated with a custom comment, visible in the *Comment* column of the **Rules pane**. This is particularly useful for documenting complex rules such as [Pascal Script](#) or [Regular Expressions](#).

To add or edit a comment, select the rule and press `Shift+Enter`, or use *Comment* from the context menu.

Saving rules as presets

A set of rules can be saved as a **preset** and reloaded on demand, making it easy to reuse common rule configurations across sessions. See [Using presets](#) for more information.

Previewing files

Preview applies the active rules and shows the resulting names, without making any changes to files on disk. It is the essential step between setting up rules and committing to a rename.

Workflow

When a preview is triggered, each *marked* file is passed through the rule stack in order. Each rule transforms the name produced by the previous one, so the final result in the *New Name* column reflects all active rules applied cumulatively. This is why the order of rules matters — the same rules in a different sequence can produce a different outcome.

Only *marked* files and *marked* rules are included. *Unmarked* files are skipped and their *New Name* column remains empty. *Unmarked* rules are ignored, which is a convenient way to test the effect of individual rules without removing them from the stack. See [Managing rules](#) for more detail.

Preview results

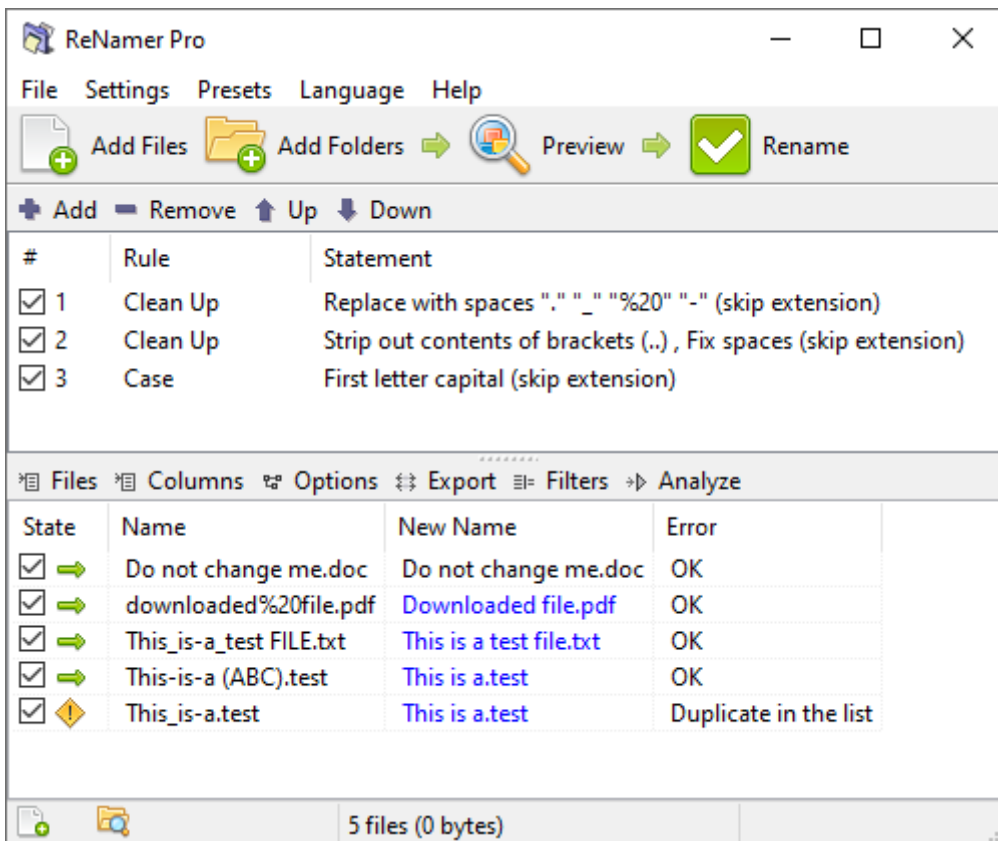
After a preview, the **Files pane** reflects the following:

- The *State* column indicates whether each file is ready to rename or has a potential issue. A green arrow means the file is ready. A warning sign flags a problem such as a duplicate name or an invalid character.
- The *New Name* column shows the resulting name. If the *Highlight changed names* option is enabled, names that differ from the original are shown in a distinct colour, making unchanged files easy to spot.
- The *Error* column describes any detected issue for that file.

Note that preview validation can flag common problems in advance, but not every possible failure can be detected before the actual rename. See [Validation](#) for more detail on how warnings are generated.

The columns displayed in the **Files pane** can be customised. Many users prefer to show the *Path* and *New Path* columns to see full paths alongside names. To toggle columns, right-click the column header row. See [Files header menu](#) for more detail.

Example



The example above has three rules in the stack: two [Clean Up](#) rules that replace punctuation characters and strip bracket contents, followed by a [Case](#) rule that capitalises the first letter of the name.

- `Do not change me.doc` is unaffected by the rules and its new name is shown in the same colour as the original.
- The remaining files are transformed and their new names are highlighted.
- The last file produces `This is a.test`, the same result as the file above it. The *Error* column flags this as *Duplicate in the list*, which would cause a conflict at rename time.

Automatic and manual preview

By default, preview updates automatically whenever files or rules change. This behaviour is controlled by the *Auto preview* option in [Preview settings](#).

When *Auto preview* is disabled, preview must be triggered manually by clicking the *Preview* button or pressing **F5**. Manual mode is useful in two situations:

- When working with a large number of files, where automatic preview on every change would be slow.
- When using [manual editing](#) to fine-tune names after a preview, as triggering a new preview would overwrite those edits.

In both modes, files are renamed exactly as shown in the preview.

Renaming files

Once the [preview](#) looks correct, clicking the *Rename* button or pressing **F6** commits the operation.

Files are renamed exactly as shown in the *New Name* column — what you see is what you get.

How renaming works

When the *Rename* button is pressed, all *marked* files are renamed according to the *Name* and *New Name* columns in the **Files pane**. The *New Path* column, when visible, reflects the full destination path derived from combining the original folder path with the new name.

After renaming:

- The *Name* column is updated to reflect the new filename.
- The *New Name* column is cleared, since there is no longer a proposed name pending.
- The previous names are stored internally to allow the operation to be undone.

If the results are not what you expected, the most likely cause is that the preview was not refreshed after the last rule change. This can happen in manual preview mode. Press the *Preview* button or **F5** to update the preview before renaming again.

Renaming outcomes

Each file in the **Files pane** will have one of the following outcomes after a rename:

- **Renamed:** The file was successfully renamed to a new name.
- **Unchanged:** The new name was identical to the original, so no change was applied. The operation is still considered successful.
- **Skipped:** The file was *unmarked* and excluded from the operation entirely.
- **Failed:** An error prevented the file from being renamed. See [Failed renaming](#) for common causes and how to resolve them.

Note that even an *Unchanged* outcome can fail, for example if the source file no longer exists at its original path.

Undoing a rename

The most recent renaming operation can be reversed using *Undo Renaming* from the [File menu](#), or by pressing `Shift+Ctrl+Z`. This restores the original filenames using the stored previous names, which are also visible in the *Old Path* column of the **Files pane**.

Post-rename behaviour

Several automatic actions can be configured to run after a successful rename. For example, files that were renamed can be cleared from the **Files pane** automatically, or the application can close.

These options are available in [Rename settings](#).

Rules

Description of individual renaming rules, their options, and example uses.

Overview of Rules

ReNamer has an extensive set of rules for renaming files. These rules can be combined together, in a logical sequence, to perform nearly any thinkable operation with the filename. You can also [manually edit](#) the new name of any file to apply fine tuning.

You can create a stack of rules to achieve a complex renaming algorithm. The rules will be applied in the order in which they appear in the stack, from top to bottom. A rule modifies the name and then passes it to the next rule, which acts on the *modified* name (not the original name). Any rule can be used multiple times and with different settings.

If you are going to use the same set of rules frequently, you can save them as a [preset](#) and later reload the entire set for immediate application.

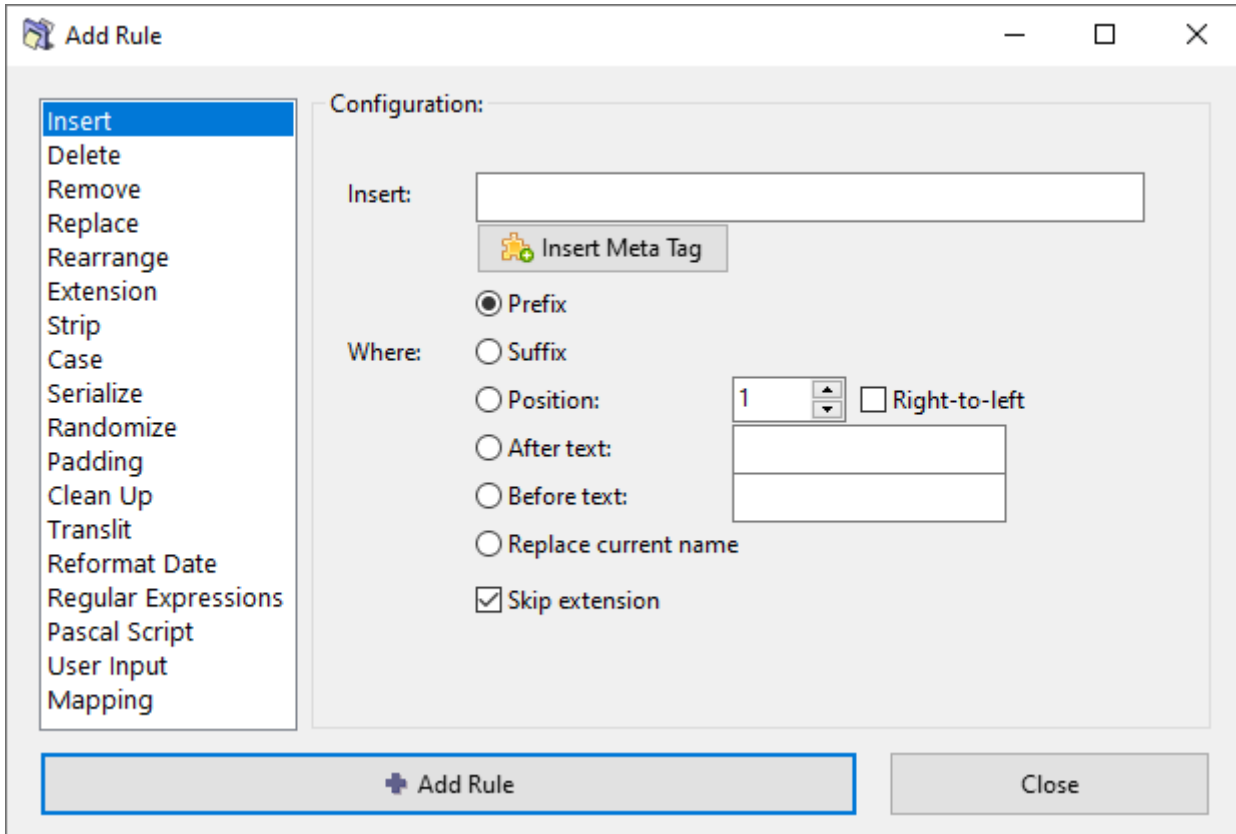
The table below lists all rules, with a brief description of each rule. The subsequent chapters provide more details for each rule.

Rule	Description
Insert	Insert the specified text into the filename: as prefix, as suffix, at the specified position, before or after certain text. There is also an option to insert meta tags into the filename.
Delete	Delete a portion of the filename, usually defined by character positions: from the specified position, from the occurrence of the specified delimiter, until the specified number of characters, until occurrence of the specified delimiter or till the end. This rule can be set to process the filename in a right-to-left manner.
Remove	Remove the specified text from the filename: first, last or all occurrences. Optionally, wildcards can be used within this rule, to remove masked text fragments.
Replace	This rule is very much like the <i>Remove</i> rule (above). It has similar options, except that instead of removing the matching text, it will replace it with alternative text.
Rearrange	Chop up the existing file name using any delimiter or position and reuse any/all of the parts in any order to compose a new name. Add extra text, or use the meta tags extracted from the file to compose the new name.
Extension	Change extension of files to the specified extension, or to the extension automatically detected through the internal database of binary signatures.

Rule	Description
Strip	Strip all occurrences of the specified characters from the filename. This rule has predefined character sets, like digits, symbols, brackets, but you can also define your own character set.
Case	Change the case of the filename: capitalize each word, to lower case, to upper case, invert case, or capitalize only the first letter and force the rest to lowercase (as in a sentence). There is also an option to force case for the manually entered fragments, for example: CD, DVD, SMS, ReNamer, etc.
Serialize	Add incremental numbers to put filenames into an order.
Randomize	Add randomly generated sequences into filenames.
Padding	Apply or remove zero padding to/from number sequences, or add text padding using custom characters.
Clean Up	Clean up filenames from (or for) commonly used naming conventions for Internet, peer-to-peer networks, and other applications.
Translit	Transliterate Non-English characters from different languages into their English/Latin representation. Useful for preparing files for network storage and transfer. Several transliteration maps are built in, and you can define your own maps.
Reformat Date	Change the format of date and time values in the filename. Also, it can applied adjustments to the date and time components, e.g. add hours, subtract days, etc.
Regular Expressions	Regular Expressions (RegEx) are used for complex pattern/expression matching and replacing operations. Although it may look complex at first, you can learn it quite easily using the provided guide.
Pascal Script	Scripting allows programming-aware users to code their own renaming rule using predefined set of functions. This rule uses Pascal/Delphi programming syntax and conventions. Extremely powerful feature in the right hands.
User Input	Set the new names of the files to the names entered in a list, one name per line.
Mapping	Map old filenames to new filenames, which can be externally managed and loaded into the rule via clipboard or CSV file. It also allows you to capture the current set of files and their generated new names as a rule set, which could then also be applied to a different set of files.

Insert rule

This rule inserts the specified text at the beginning of the name, at the end of the name, at a specified position, or before or after certain text.



The screenshot shows the 'Add Rule' dialog box with the 'Insert' rule selected. The 'Configuration' section includes the following options:

- Insert:** A text input field.
- Insert Meta Tag:** A button with a puzzle piece icon.
- Where:**
 - ☒ Prefix
 - ☐ Suffix
 - ☐ Position: A numeric input field set to '1' and a 'Right-to-left' checkbox.
 - ☐ After text: A text input field.
 - ☐ Before text: A text input field.
 - ☐ Replace current name
 - ☒ Skip extension

At the bottom of the dialog are two buttons: 'Add Rule' and 'Close'.

Options

Insert

Enter the subject text that will be inserted.

Insert meta tag

Click the button to see a list of [meta tags](#), which extract meta information associated with files and use it for renaming.

Where

Where to insert the subject text. Select one of the *Where* options described in the section below.

Skip extension

If checked, the file extension will be excluded from processing and will remain unaffected.

Where options

Prefix

Adds the subject text at the beginning of the existing name.

Suffix

Adds the subject text at the end of the existing name.

If the *Skip extension* option is checked, the subject text will be inserted before the file extension. Otherwise, it will be inserted after the file extension.

Position

Insert the subject text at the specified position.

- The position index starts from 1.
- Account for spaces and special characters when counting the position.
- You can count in the *right-to-left* direction if the corresponding option is enabled.
- You can use the [Analyze tool](#) to check the position by pointing to the character with mouse or keyboard instead of counting it by eye. You can also select sample files from the table, right-click to open the context menu and choose the *Analyze name* option.

After text

Insert the main subject text after an occurrence of text entered in the "after text" field.

If the "after text" is not found in the name, the main subject text will not be inserted.

Before text

Insert the main subject text before an occurrence of text entered in the "before text" field.

If the "before text" is not found in the name, the main subject text will not be inserted.

Replace current name

The current name is deleted and replaced by the subject text.

Delete rule

This rule deletes all characters located between the *From* and the *Until* positions.

Options

From

From which character-position you want to start the deletion.

Select from the following options:

- The starting position (the position starts from 1).
- The delimiter from where the deletion starts.
 - The delimiter can be a single character or any arbitrary text.

Until

Until which character-position you want to delete.

Select from the following options:

- Delete an exact number of characters.
- Delete until the specified delimiter is reached.
 - The delimiter can be a single character or any arbitrary text.
- Delete all characters until the end.

Delete current name

A shortcut option for deleting the whole current name.

Skip extension

If checked, the file extension will be excluded from processing and will remain unaffected.

Right-to-left

Inverts the normal direction for counting a position, so that the count goes from right and towards left.

When this option is selected, the "till the end" option in *Until* deletes all characters on the left, what we normally regard as the *beginning* of the name.

For example, to keep only the four characters on the right side of the name and delete everything else, use:

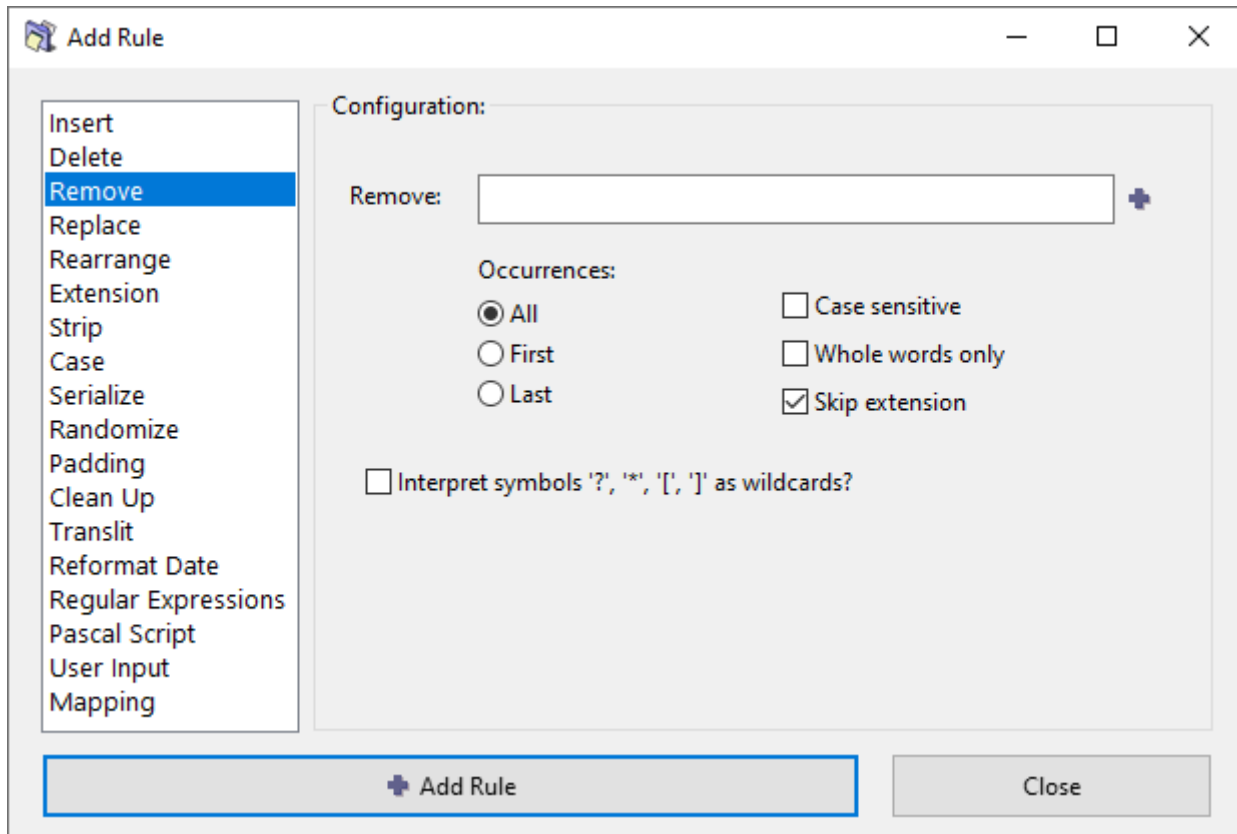
- **From:** Position 5
- **Until:** Till the end

Do not remove delimiters

If you select this option, the delimiters themselves will be retained. Otherwise, delimiters will be removed together with the deletion range.

Remove rule

This rule removes the specified text from the file name, optionally using positional criteria and wildcard patterns.



Options

Remove

Enter the text to be removed.

You can add multiple pieces of text to remove in one rule, by separating them with `*|*` delimiter. Click the  button to add the delimiter, or type it out manually.

Occurrences

When the text to be removed occurs multiple times in the filename, you can choose to remove all occurrences, only the first occurrence, or only the last occurrence.

Case sensitive

Match case exactly when searching for text to remove. When enabled, "Apple" will not match "apple". When disabled, uppercase and lowercase letters are treated as equivalent.

Whole words only

Match whole words only. When enabled, the text must appear as a complete word, not as part of a longer word. For example, searching for "bar" will not match "foobar" or "bars".

Skip extension

If checked, the file extension will be excluded from processing and will remain unaffected.

Interpret wildcards

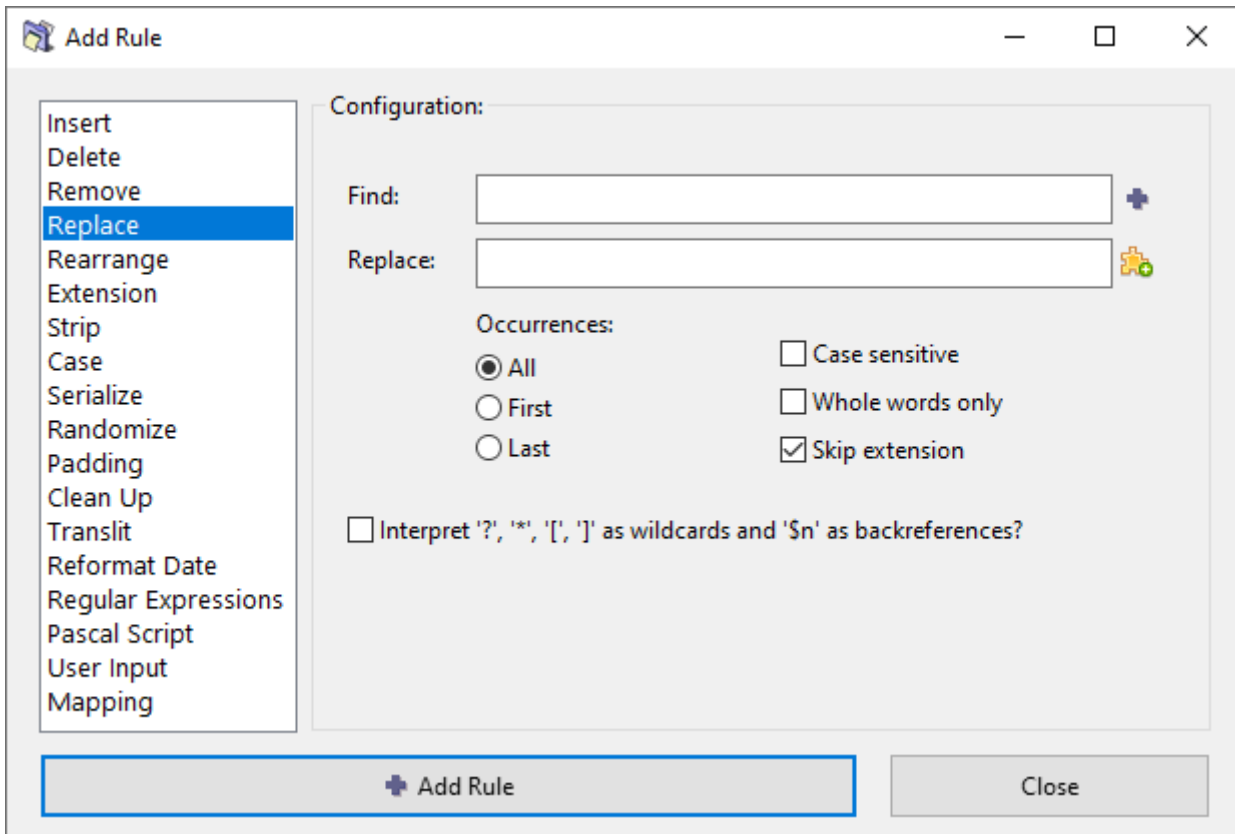
Enable wildcard interpretation for pattern matching.

Wildcard	Description	Example
*	Matches zero or more characters.	<code>abc*</code> matches <code>abc</code> , <code>abcd</code> , <code>abc123</code> , etc.
?	Matches exactly one character.	<code>ab?d</code> matches <code>abcd</code> , <code>ab1d</code> , <code>ab d</code> , <code>ab_d</code> , etc.
[]	Brackets enclose a set of characters, any one of which must match a single character at that position.	<code>foo[ab]ar</code> matches <code>fooaar</code> and <code>foobar</code> .
-	Within brackets, denotes a range of characters.	<code>foo[a-z]ar</code> matches <code>fooaar</code> , <code>foobar</code> , <code>foocar</code> , <code>foodar</code> , etc.

For more advanced pattern matching, consider using the [Regular Expressions rule](#).

Replace rule

This rule finds the specified text in the file name and replaces it with another text, optionally using positional criteria and wildcard patterns.



Options

Find

Enter the text to be replaced.

You can add multiple pieces of text to be replaced in one rule, by separating them with `*|*` delimiter. Click the  button to add the delimiter, or type it out manually.

Replace

Enter the text which will replace the *Find* text.

When using multiple *Find* text pieces, separated by the `*|*` delimiter, every n^{th} piece in the *Find* field will be replaced by the corresponding n^{th} piece in the *Replace* field.

Click the  button to insert [Meta Tags](#) as the replacement text.

Occurrences

When the text to be replaced occurs multiple times in the filename, you can choose to replace all occurrences, only the first occurrence, or only the last occurrence.

Case sensitive

Match case exactly when searching for text to replace. When enabled, "Apple" will not match "apple". When disabled, uppercase and lowercase letters are treated as equivalent.

Whole words only

Match whole words only. When enabled, the text must appear as a complete word, not as part of a longer word. For example, searching for "bar" will not match "foobar" or "bars".

Skip extension

If checked, the file extension will be excluded from processing and will remain unaffected.

Interpret wildcards

Enable wildcard interpretation for pattern matching.

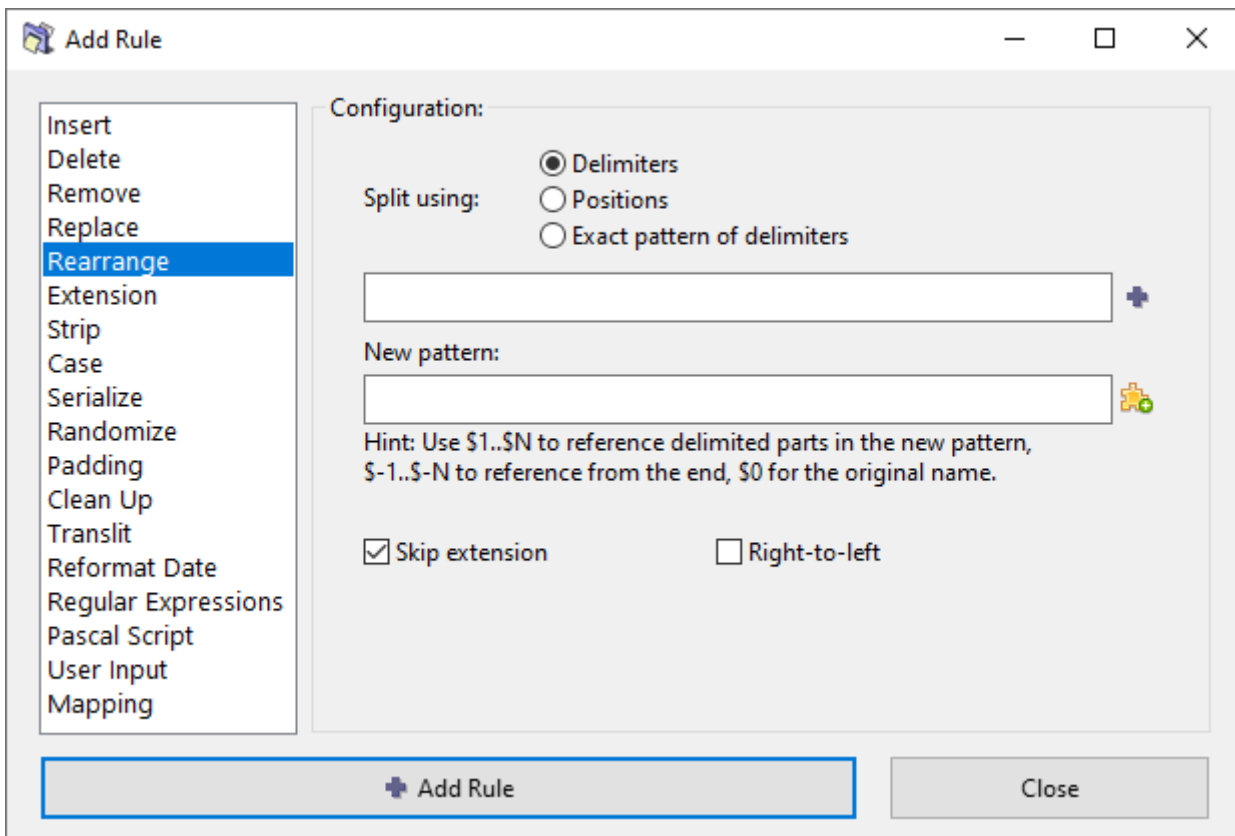
Wildcard	Description	Example
*	Matches zero or more characters.	<code>abc*</code> matches <code>abc</code> , <code>abcd</code> , <code>abc123</code> , etc.
?	Matches exactly one character.	<code>ab?d</code> matches <code>abcd</code> , <code>ab1d</code> , <code>ab d</code> , <code>ab_d</code> , etc.
[]	Brackets enclose a set of characters, any one of which must match a single character at that position.	<code>foo[ab]ar</code> matches <code>fooaar</code> and <code>foobar</code> .
-	Within brackets, denotes a range of characters.	<code>foo[a-z]ar</code> matches <code>fooaar</code> , <code>foobar</code> , <code>foocar</code> , <code>foodar</code> , etc.

For more advanced pattern matching, consider using the [Regular Expressions rule](#).

Rearrange rule

This rule allows you to chop up the existing file name and reuse any/all of the parts in any order to compose a new name.

- You can also add your own text, or use meta tags while composing the new name.
- You can also use the whole original name, and insert literal text (or meta tags) around it.



The screenshot shows the 'Add Rule' dialog box with the 'Rearrange' rule selected in the left-hand menu. The main configuration area is titled 'Configuration:' and contains the following elements:

- Split using:** Three radio buttons are present: 'Delimiters' (selected), 'Positions', and 'Exact pattern of delimiters'.
- Input field:** A text box for entering delimiters or positions, with a plus icon to its right for adding more.
- New pattern:** A text box for entering the new file name pattern, with a puzzle piece icon to its right.
- Hint:** Text below the pattern field stating: 'Hint: Use \$1..\$N to reference delimited parts in the new pattern, \$-1..\$-N to reference from the end, \$0 for the original name.'
- Options:** Two checkboxes are shown: 'Skip extension' (checked) and 'Right-to-left' (unchecked).

At the bottom of the dialog, there are two buttons: 'Add Rule' (highlighted with a blue border) and 'Close'.

Options

Split using

Specify how to split the existing name into parts, using either a set of delimiters, an exact pattern of delimiters, or exact positions.

Enter a list of delimiters/positions, separating them with `|` (vertical pipe) character. Alternatively, click the  button to add another delimiter/position used for splitting.

A more detailed explanation of different split options can be found in the *Split options* section below.

New pattern

Compose the new name from the parts created from the original name.

All the parts resulting from splitting the original name can be accessed via short codes `$1`, `$2`, `$3`, and so on. You can also reference the parts using indexing from the opposite end via negative short codes `$-1`, `$-2`, `$-3`, and so on. The full original name is accessible via short code `$0`.

Using a mix of short codes and any other text you can rearrange the name and add additional text around the rearranged parts.

Click the  button to insert [Meta Tags](#) into the new pattern.

Skip extension

If checked, the file extension will be excluded from processing and will remain unaffected.

Right-to-left

Inverts the normal direction for splitting the subject text, so that it goes from right and towards left.

Split options

Delimiters

Chop the name where the delimiters occurs.

- The delimiter can be a single character or a piece of text.
- The chopped parts will not contain the delimiters (they are omitted totally).
- Several different delimiters can be used at a time. Use the `|` character to separate them.
- The chopped parts are numbered from the start as `$1`, `$2`, `$3`, and so on, or from the end as `$-1`, `$-2`, `$-3`, and so on.
- If the delimiter occurs at the very beginning of the name, the resultant `$1` part will be empty, because there is nothing on the left side of the delimiter.

The number of parts into which the filename is broken down depends solely on the number of delimiters in the filename. If you reference fewer parts in the output pattern than the number of available parts, not referenced parts will be discarded.

For example, take filename "Artist - Title" and to swap it around one would use " - " as a delimiter and "\$2 - \$1" as a new pattern which will result in "Title - Artist". However, if some filename appears with more dashes like "Artist - Title - Album" the result will also be "Title - Artist" and the trailing part "Album" will be discarded. To avoid silently discarding parts of the filename, consider using the *Exact pattern of delimiters* option instead.

Positions

Chop the name at the indicated position.

- The position count begins with 1.
- Every position marks the start of a chopped part, including the character at that position.
- No part of the original name is omitted during chopping.
- You can enter multiple positions, separating them with the `|` (vertical pipe) character.
- The chopped parts are numbered from the start as `$1`, `$2`, `$3`, and so on, or from the end as `$-1`, `$-2`, `$-3`, and so on.

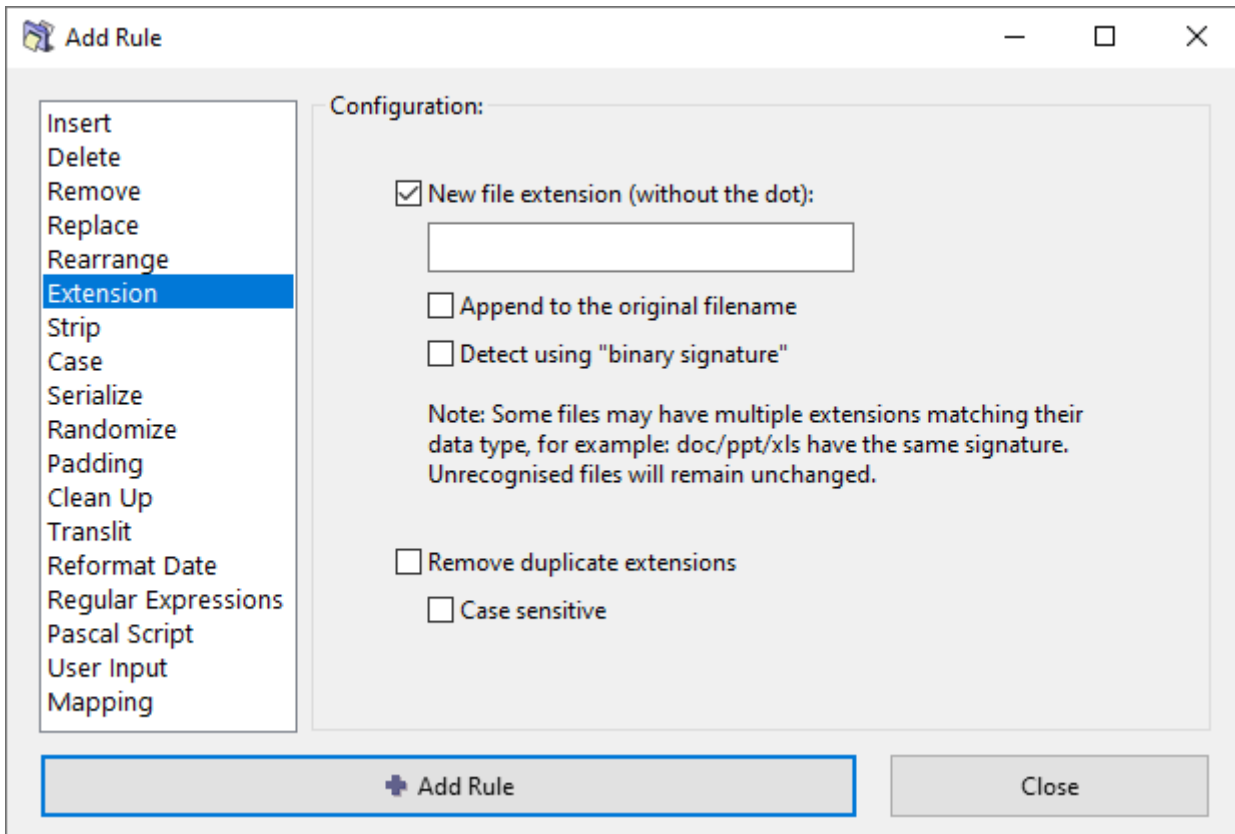
Exact pattern of delimiters

Chop the name using an exact pattern (sequence) of delimiters.

With this option you basically define how many parts you want the filename to be split into and the order in which the delimiters must occur. If you specify 1 delimiter then you will end up with exactly 2 parts, if you specify 2 delimiters you'll get 3 parts, and so on.

Extension rule

This rule lets you change the file extension, with the option to automatically detect the appropriate extension based on the file's content.



Options

New file extension

Specify the new file extension, without a leading dot. For example: `txt`, `mp3`, `jpg`.

Append to the original filename

Rather than changing the original file extension, appened the new extension to the end of the filename.

Detect using binary signature

Attempt to detect the appropriate file extension based on the file's content, using the so called binary signatures (see the section below).

Remove duplicate extensions

If a file has consecutive duplicate extensions, remove the duplication. For example: `file.jpg.jpg` becomes `file.jpg`.

Case sensitive

When removing duplicate extensions, extensions must match exactly to be considered duplicates. For example: `.jpg.jpg` will be corrected, but `.jpg.JPG` will not.

Binary signatures

Sometimes the extension of a file is missing. At other times it is simply wrong, e.g. some downloaded files get the `aspx` extension, although they may *actually* be `zip` or `pdf` files. One way to identify the file extension is by trial-and-error: Attach different extensions and try to open the file with its associated application. This is very tedious.

A far more efficient way is to compare the file's *digital signature* with the signatures of known file types to identify the correct file type. This is done internally within ReNamer, so you do not have to know what *digital signature* means, or the actual value of the signature for a given file.

Note that each extension has a range of signatures, and these ranges overlap. This means a given file's signature may match with the signature of several different extensions. In such cases, ReNamer shows the new filename with all matching extensions. For example, `file.wma|wmv|asf`. ReNamer also shows a warning message, because the combined extension is invalid. Just read the suggested extensions and try them out one by one. This method is still better compared to making wild guesses.

For more accurate results, use [ReNamer with TrID library](#), a specialized utility for identifying the file's real extension. Be aware that even TrID tool often suggests multiple extensions, and you may still have to try them out.

Filename starts with a dot

Many operating systems treat the dot at the start of the filename as an indicator that the file should be hidden from the user. For that reason, the dot at the start of the filename is not considered as a file extension delimiter.

This behavior can be toggled via an option in the settings file:

```
FirstDotAtFileNameStartIsExtension=1
```

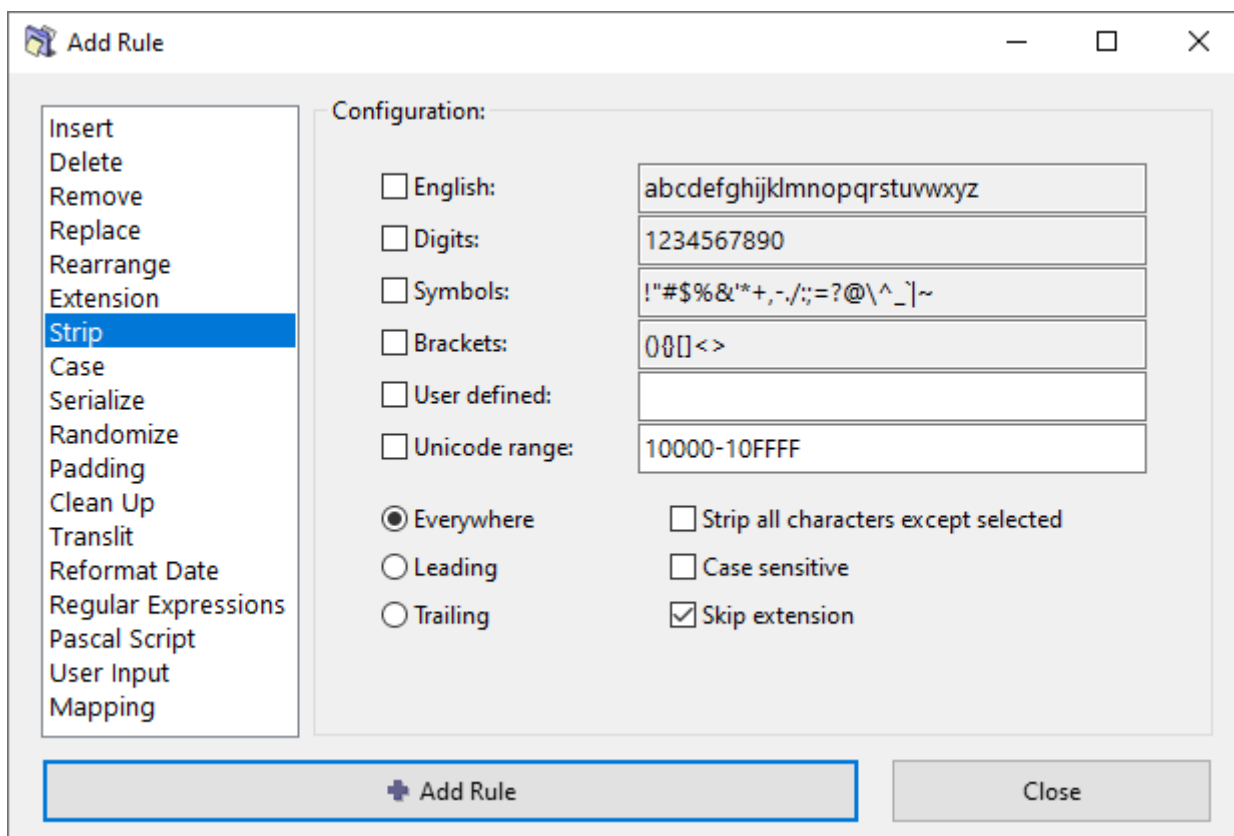
The settings file is normally located in the [Application Data Storage](#).

Examples of filenames and their considered extension depending on the value of the option:

Filename	Extension if 1	Extension if 0
file.ext	.ext	.ext
.file.ext	.ext	.ext
.file	.file	

Strip rule

This rule strips a list of characters from the filename. It offers predefined character sets, like digits, symbols and brackets, but you can also define your own character set. All occurrences of the specified characters will be removed from the filename.



Character set options

English

Strip all English characters: "a" through "z".

Characters used in other languages, including those with diacritical marks, will not be stripped.

Digits

Strip all digits: "0" through "9".

Symbols

Strip common symbols: `!"#$%&'*+,-./:;=?@^_`|~.`

Brackets

Strip common brackets:

The content between the brackets will remain unaffected. If you want to delete the content as well, use the [Clean Up rule](#) instead.

User defined

Specify any characters that need to be removed.

Unicode range

Strip Unicode codepoints by their hexadecimal codes.

You can specify a range of characters using the (dash) between two codes.

Multiple codes and ranges can be separated using the (comma) character.

For example:

- for [supplementary planes](#),
- for [zero width characters](#).

Positioning options

- *Everywhere* – The characters will be stripped no matter where they occur in the filename.
- *Leading* – Strip characters only at the beginning of the filename.
- *Trailing* – Strip characters only at the end of the filename.

General options

Strip all characters except selected

Invert the logic by keeping only the specified characters, and stripping all other characters.

For example, if you want to strip all non-English characters, select this option along with the *English* character set option above.

Case sensitive

Match character case exactly when searching for characters to remove.

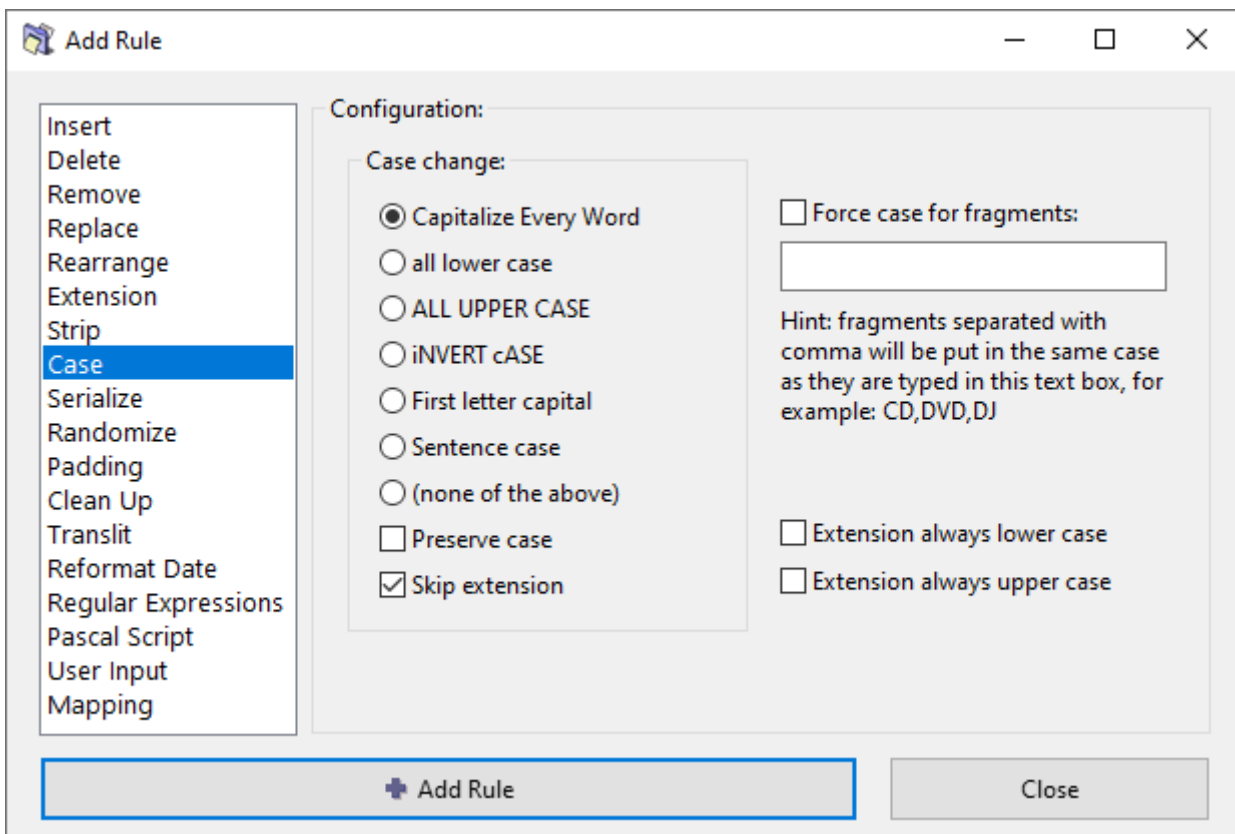
When enabled, "A" will not match "a". When disabled, uppercase and lowercase letters are treated as equivalent.

Skip extension

If checked, the file extension will be excluded from processing and will remain unaffected.

Case rule

This rule changes the case of the filename, including capitalizing every word, converting to lower/upper case, inverting the case, and enforcing a specific case for certain words, e.g. CD, DVD, SMS, PhD, iPad.



Case change options

Select one of the available text case transformations:

- **Capitalize Every Word** – Convert all letters to lower case (*general normalization*), and then convert the first letter of each word to upper case.
- **all lower case** – Convert all letters to lower case.
- **ALL UPPER CASE** – Convert all letters to upper case.
- **iNVERT cASE** – Invert the case of all letters, convert lower case to upper case, and vice versa.
- **First letter capital** – Convert only the first letter to upper case, and the rest to lower case (*general normalization*).
- **Sentence case** – Convert the first letter of every sentence to upper case, and the rest to lower case (*general normalization*).

- **(none of the above)** – Do not apply any of these general transformations.

Preserve case

Preserve the original case where applicable by not performing the *general normalization*. See transformation options above.

Skip extension

If checked, the file extension will be excluded from processing and will remain unaffected.

Force case for fragments

This option enforces a specific case for the specified words (text fragments).

Enter the text fragments with the desired case in the text field provided. Separate multiple fragments with (comma).

For example: .

Extension options

Convert the file extension to either upper case or lower case, regardless of other options and case transformations.

Select one of the following options:

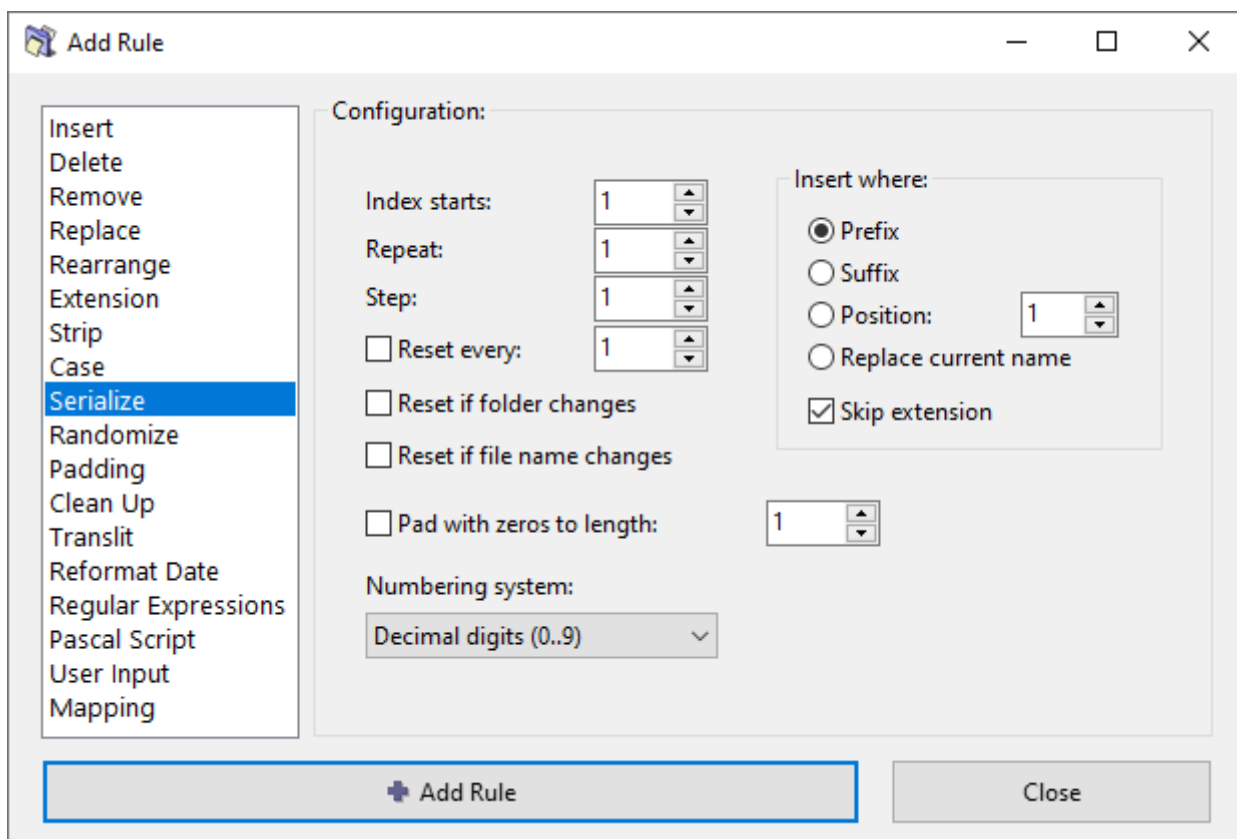
- *Extension always upper case*
- *Extension always lower case*

Serialize rule

This rule allows you to index (serialize) a list of files by inserting numbers in increasing or decreasing order, using several different numbering systems, including Decimal digits, English letters, Roman numerals, Music notes, and more.

Example use cases:

1. You have a bunch of log files, and you want to make them look like "log0001", "log0002", "log0003", etc.
2. You want to force a specific sorting for your music collection: "01 - Song A", "02 - Song B", "03 - Song C", etc.



Indexing options

Index starts

Specify the starting number to be used for the first file.

For example, if the destination folder already has some files with serialized numbers, you can start with the next available number.

Repeat

How many times to repeat (reuse) the same index before incrementing it.

For example, if the repeat value is 2 and the index starts at 1, you will get these indexes 1, 1, 2, 2, 3, 3, and so on.

Step

Increment the index by this value after each processed file.

Usually the step is 1, but you may like to enter a higher number here if files with intermediate numbers are expected later. Also, negative numbers can be used to make decremental indexes, e.g. -1, -2, -3, etc.

Reset every

Reset index to the initial value after processing this many files.

Reset if folder changes

Reset index to the initial value if the folder path of the current file is different from the folder path of the previously processed file.

If you files are sorted by the folder path, you will get a fresh sequence of indexes used for each folder.

Reset if file name changes

Reset index to the initial value if the name of the current file is different from the name of the previously processed file.

Pad with zero to length

Pad the inserted number with leading zeros.

For example, index "123" becomes "000123" if it is padded to 6 digits, and "0123" if padded to 4 digits.

Insert where options

Specify where to insert the number. You have the following options:

- **Prefix** – Before the original filename.
- **Suffix** – After the original filename.
- **Position** – Insert the number at the specified position.
- **Replace current name** – Inserted number replaces the entire filename.
- **Skip extension** – Exclude file extension when calculating the position for insertion.

Numbering system

The choice of numbering system determines the symbols and enumeration techniques used for serialization.

The examples below illustrate each numbering system.

Decimal digits

Index starts at 0 with repeat 1 and step 1:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ...

English letters

Index starts at 1 with repeat 1 and step 1:

a, b, c, ..., x, y, z, ba, bb, bc, ..., bx, by, bz, ca, cb, cc, ..., zx, zy, zz, baa, bab, bac, ...

Index starts at 1 with repeat 1 and step 1, pad to length 3:

aaa, aab, aac, ..., aax, aay, aaz, aba, abb, abc, ..., abx, aby, abz, aca, acb, acc, ..., azx, azy, azz, baa, bab, bac, ...

Roman numerals

Index starts at 1 with repeat 1 and step 1:

I, II, III, IV, V, VI, VII, VIII, IX, X, XI, XII, ...

Music notes

Index starts at 1 with repeat 1 and step 1:

C0, C#0, D0, D#0, E0, F0, F#0, G0, G#0, A0, A#0, B0, C1, C#1, ...

Added in ReNamer 7.3.0.2 Beta.

Simplified Chinese

Index starts at 1 with repeat 1 and step 1:

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 00, 000, 000, 000, ...

Custom alphabetic

Index starts at 1 with repeat 1 and step 1, using "ABC" symbols:

A, B, C, BA, BB, BC, CA, CB, CC, BAA, BAB, BAC, BBA, ...

Index starts at 1 with repeat 1 and step 1, using "01" symbols:

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, ...

Custom numeric

Index starts at 0 with repeat 1 and step 1, using "ABC" symbols:

A, B, C, BA, BB, BC, CA, CB, CC, BAA, BAB, BAC, BBA, ...

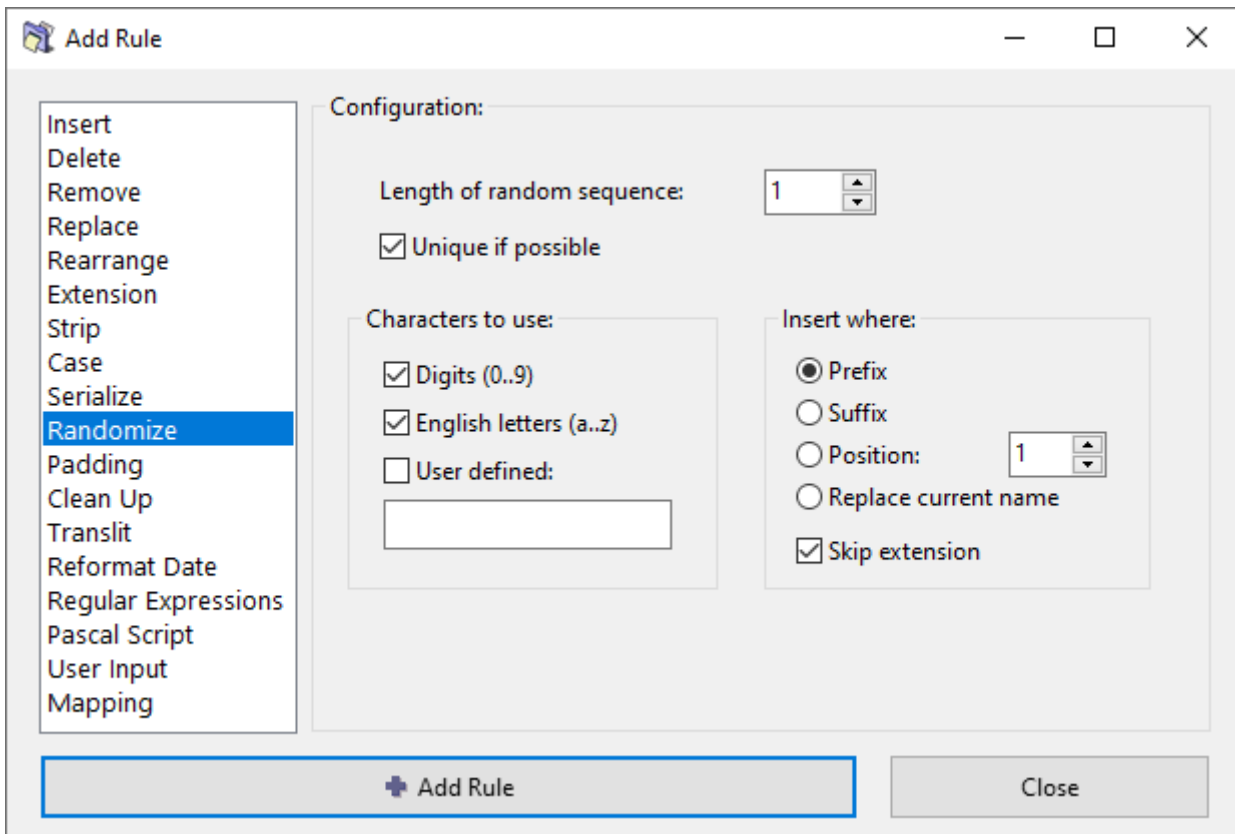
Index starts at 0 with repeat 1 and step 1, using "01" symbols:

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, ...

Randomize rule

This rule inserts randomly generated sequences of specified length and using a selection of characters (digits, letters, or a custom set).

Common practical uses include randomizing the order files and destroying existing filenames.



Options

Length of random sequence

Specify how many characters your random sequence should have.

For example, if you are using only digits and set length to 4, you will will get random numbers from 0000 to 9999, inclusive.

Unique if possible

Normally, randomly generated values can repeat themselves. If this option is enabled, ReNamer will attempt to generate unique values (without repetition) when possible.

Characters to use

The set of characters which will be used for generating random sequences.

You can choose between using digits (0..9), English letters (a..z), or specify your custom set of characters.

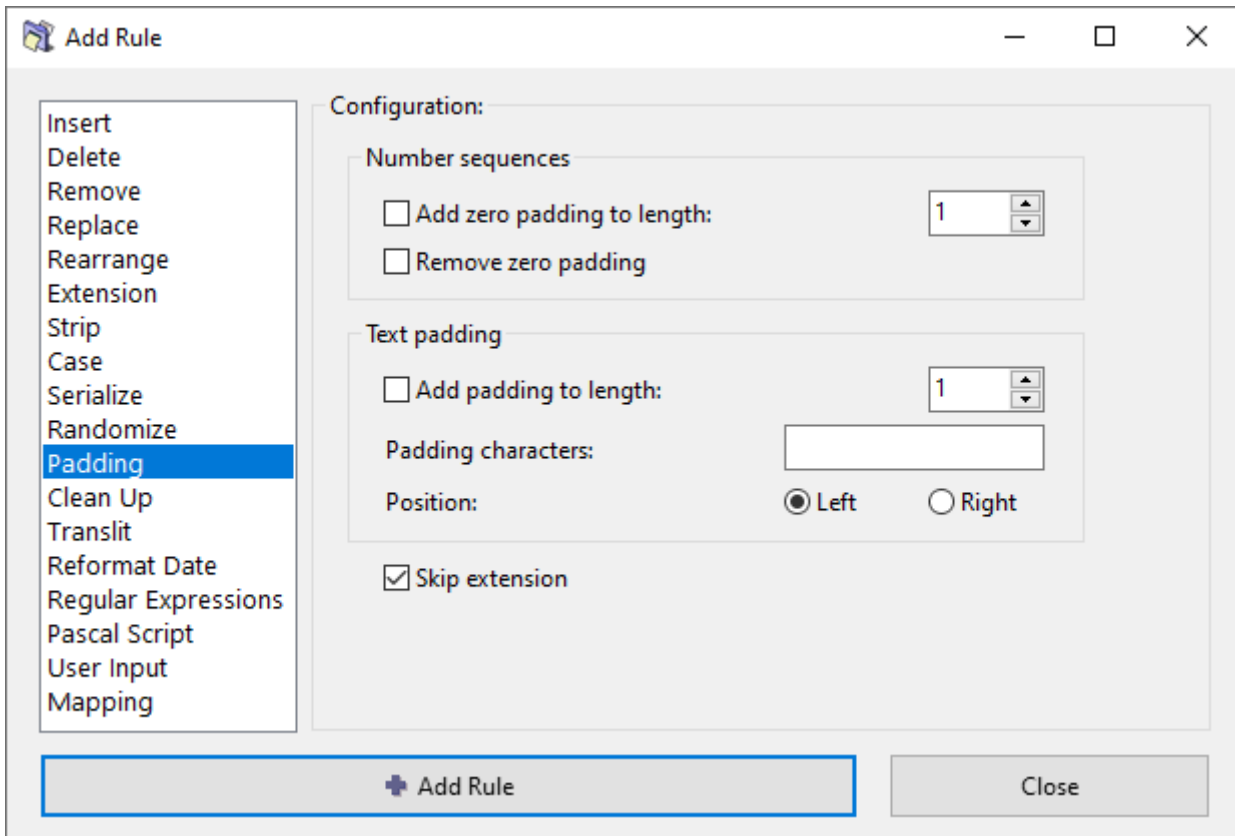
Insert where options

Specify where to insert the number. You have the following options:

- **Prefix** – Before the original filename.
- **Suffix** – After the original filename.
- **Position** – Insert the number at the specified position.
- **Replace current name** – Inserted number replaces the entire filename.
- **Skip extension** – Exclude file extension when calculating the position for insertion.

Padding rule

This rule allows you to apply or remove zero padding on number sequences, or add text padding using custom characters.



Number sequences options

Add zero padding to length

Apply zero padding to number sequences until the number of digits reaches a specified count (length).

Remove zero padding

Strip away leading zeros from all number sequences.

Text padding options

Add padding to length

Add custom text padding either form the left or the right side of the subject.

Padding characters

A sequence of custom characters which will be used as the padding material.

Position

Choose either the left or the right side for adding the text padding.

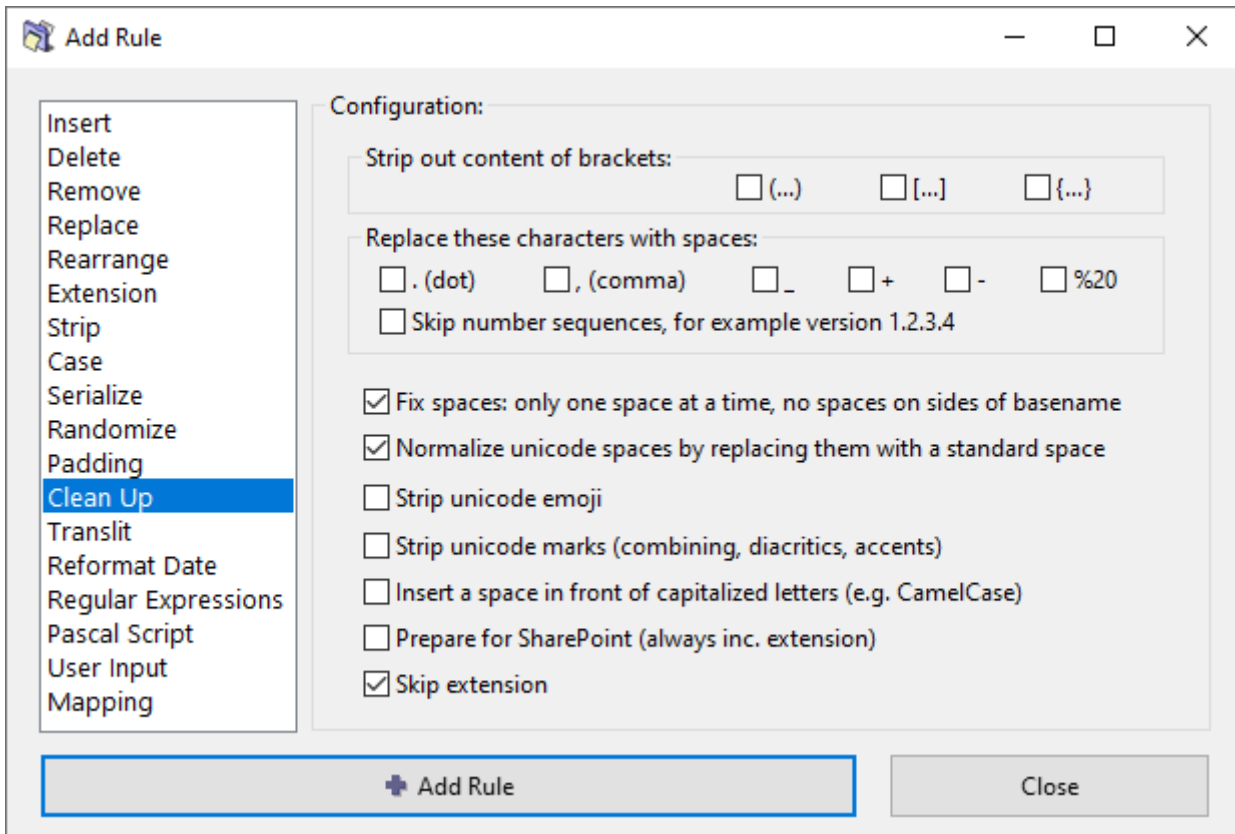
General options

Skip extension

If checked, the file extension will be excluded from processing and will remain unaffected.

Clean Up rule

This rule cleans up the filenames from (or for) commonly used naming conventions for Internet, peer-to-peer networks, and other applications. Multiple issues can be resolved at once.



Options

Strip out content of brackets

Remove the content of various brackets, including the brackets.

You can choose the types of brackets: `(...)`, `[...]`, `{...}`.

A typical use of this option is to remove the needless comments attached to filenames.

Replace these characters with spaces

Replace various characters that are often used in place of spaces, such as `.` (dot), `_` (underscore), and others.

Optionally, skip number sequences which may denote versions numbers or phone numbers, for example "1.2.3.4".

Fix spaces

Replace multiple consecutive spaces with a single space character.

It also removes spaces from the beginning and the end of the filename. If the *skip extension* option is enabled, removes spaces around the base name (before the extension).

Normalize unicode spaces

Replace all Unicode white space characters with a standard space bar character code.

[Unicode](#) character set contains a number of different characters (`C1_SPACE` type) that represent a white space with slightly different properties (e.g. wider, narrower, non-breaking, etc).

Strip unicode emoji

Strip all [Emoji](#) code points from the filename.

For example, "Happy😊Face.mp3" becomes "HappyFace.mp3".

Strip unicode marks

Strip unicode marks, including combining marks, diacritics and accents.

For example, it will convert "á" to "a", "Ñ" to "N", "X̄" to "X", and so on.

Insert a space in front of capitalized letters

Sometimes words in a filename are joined together without spaces or underscores, with each new word starting with a capital letter to improve readability. This naming pattern is also known as *CamelCase*.

This option separates such words by inserting a space before each capital letter (except the first character, to avoid a leading space).

For example, "SeparateTheseWords.pdf" becomes "Separate These Words.pdf".

Prepare for SharePoint

Prepares the filename for hosting it on Microsoft SharePoint.

1. Trim leading and trailing spaces.
2. Replace curly brackets `{ }` with round brackets `( )`.
3. Delete consecutive dots.
4. Delete forbidden characters:

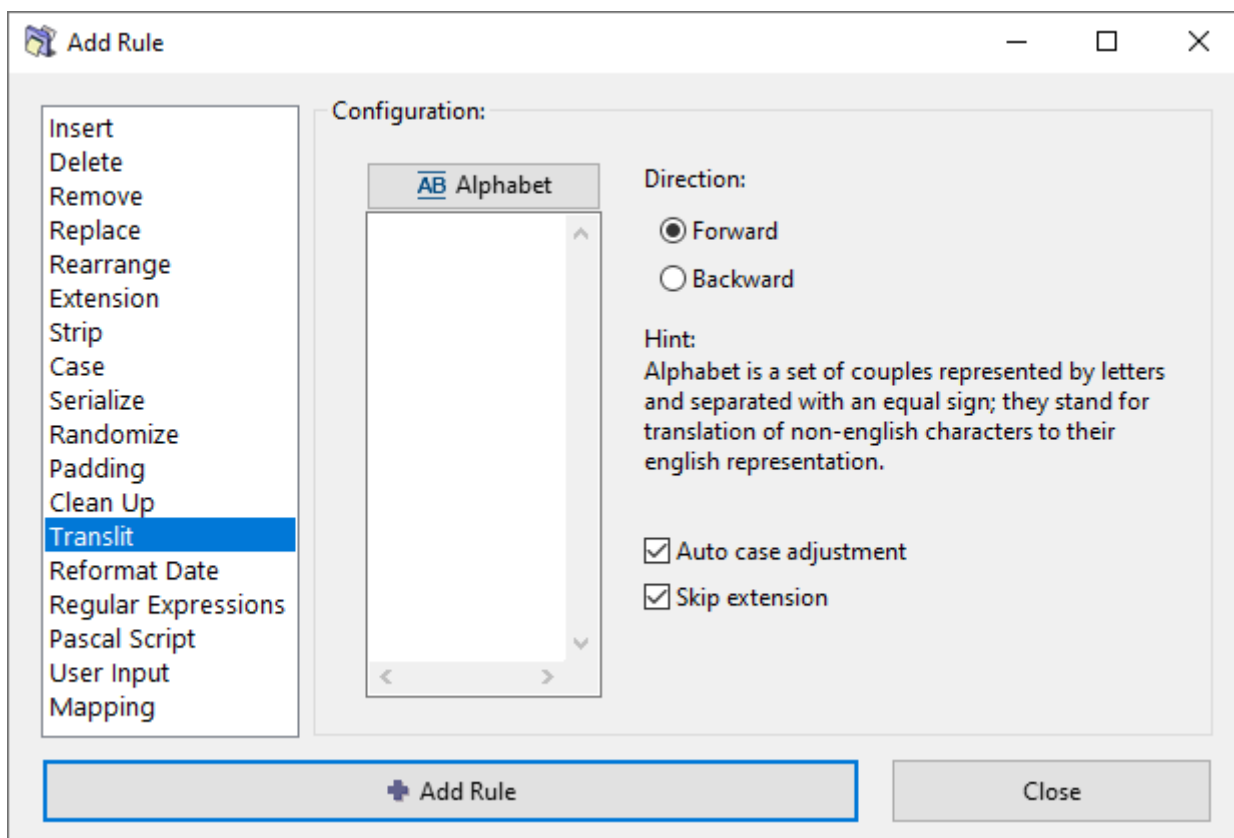
- ASCII control characters
- Standard forbidden filename characters: `/\*?"<>|:`
- Special characters forbidden in SharePoint: `#%~&`

Skip extension

If checked, the file extension will be excluded from processing and will remain unaffected.

Translit rule

This rule transliterates one alphabet into another. Its main goal is to transliterate Non-English characters from different languages into their English/Latin representation. The rule comes with built-in transliteration maps for many different languages, but you can also define your own custom mapping.



Options

Alphabet

Define a transliteration map or load an existing map from the menu. See how to work with transliteration maps in the sections below.

Direction

Select the direction for applying the transliteration: **Forward** (left to right) or **Backward** (right to left).

Auto case adjustment

Automatically adjust the text case of the output based on the case of the input. The exact algorithm is described in a section below.

Skip extension

If checked, the file extension will be excluded from processing and will remain unaffected.

Transliteration maps

To transliterate, we define pairs of equivalent characters or combinations of characters. For example, "ü" character in German language can be transliterated to "ue", so the name "Müller" becomes "Mueller" in English alphabet.

We need several such equivalent pairs to convert one language into another. The entire set is called a *transliteration map*. You can think of it as a kind of a find-and-replace rule.

The rule comes with built-in transliteration maps for many different languages, but you can also define your own custom mapping. Each map can be used in both directions (*forward* or *reverse*), for example the French map can be used as French-to-English or as English-to-French.

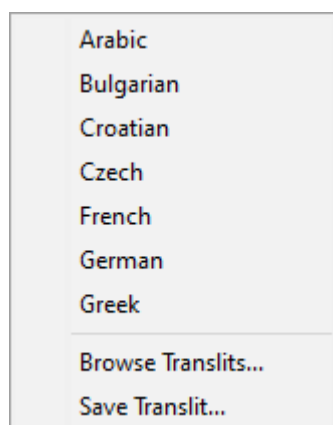
When you create a new rule, its window does not show any maps. You can now do one of the following:

1. Use any of the built-in maps.
2. Create your own map and use it.
3. Edit a built-in map first, and then use it.

Let us see how to do this.

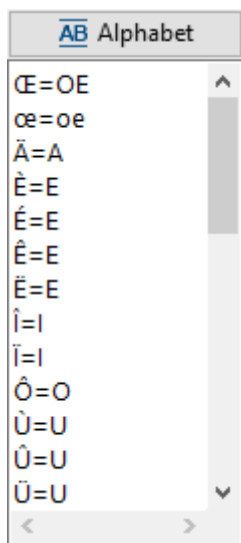
Use a built-in transliteration map

Press the  button to see the list of available transliteration maps for different languages.



Click on the desired transliteration map to load it.

For an example, let's say we loaded the French transliteration map.



You can edit any of the entries in this list, add new entries, or delete any of the entries.

Note that such editing does not alter the saved version of the map, as it only applies to the current rule configuration. You will see how to alter an existing transliteration map and create a new map in a section below.

Next, select the direction for applying the transliteration: **Forward** (left to right) or **Backward** (right to left).

Make your own transliteration map

Transliteration map (or an "alphabet") consists of sets of equivalence pairs, which are entered one per line, and two parts of a pair separated with (equal sign). The mapping should not contain spaces and should have case discarded (case is adjusted automatically).

Make sure to put couples which contain greater number of characters at the top, so they will get processed first and will not get processed partially by shorter representations.

Below is a simple example which can be used to transliterate German words:

```
ä=ae
ö=oe
ü=ue
ß=ss
```


Save a transliteration map

To save a newly composed or an edited transliteration map:

1. Press the  button.
2. Select the "Save Translit..." option.
3. Enter the name of your mapping in the dialog, and click OK.
 - When editing an existing map, the dialog will show current map's name.

Transliteration maps are saved as individual files in a folder which can be located via the "Browse Translits..." menu option. If you delete any map files in that folder, they will no longer be available for loading in the rule configuration.

Automatic case adjustment algorithm

The transliteration process performs automatic case conversion with an algorithm adopted specifically for transliteration. The rule discards the case of the map definition, i.e. "A=B" is same as "a=b". The output case is determined based on the case of each input fragment. Multiple character fragments are treated as part of words, with their case determined based on the case of letters around them.

Let's consider a mapping pair *INPUT=OUTPUT*, where the *INPUT* token takes the case of a match in the original text. The logic for determining the case of the *OUTPUT* token is as follows:

```
set OUTPUT to lower case
if first letter in INPUT is upper case then
  if length of OUTPUT greater than 1 then
    if next letter in original name is upper case then
      convert whole OUTPUT to upper case
    else
      convert only first letter in OUTPUT to upper case
  else
    convert whole OUTPUT to upper case
```

Unicode character forms

Have you encounter a case where some characters don't get converted, despite having a visually identical character defined in the Translit alphabet?

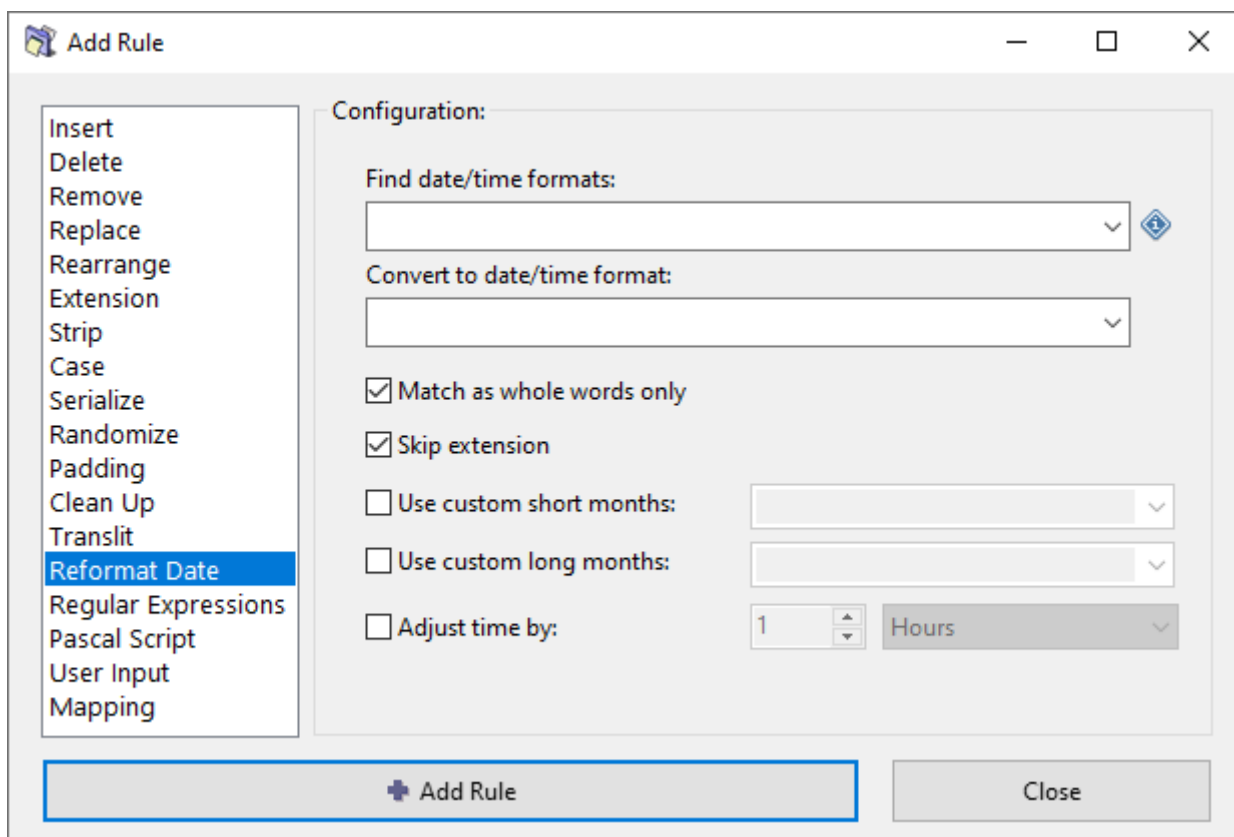
Unicode characters can be defined using exact character codes or using [combining characters](#). The displayed characters will look identical, but their binary content is different. The conversion process between these forms is covered by the [Unicode Normalization](#) standard.

Transliteration maps are defined using exact character codes, so variations that use alternative forms, including combining characters, will not be affected unless these forms are also defined in the map. You can put a piece of text through a *Unicode analyzer* tool to see exactly how each character is defined and to identify the use of combining characters.

To handle all possible forms of the same visual character, one could define all possible forms in an alphabet or one can simply strip away those combining characters, which can be accomplished by using the "*Strip unicode marks*" option found in the [Clean Up rule](#).

Reformat Date rule

This rule allows finding and reformatting various date/time values in the filename, including the ability to adjust date and time components.



Options

Find date/time formats

Find date and/or time matching a specified format. A dropdown menu provides a quick access to commonly used formats.

Supported date/time format variables are described in the [Date and Time format](#) article. For example, "yyyy-mm-dd hh:nn:ss" format works for "2025-12-31 20:55:00".

You can specify multiple date/time formats in a single rule, separating them with `|` (vertical pipe). Alternatively, you can click the button menu located on the right side of the dropdown field.

Convert to date/time format

Convert found date and/or time values to the specified format. A dropdown menu provides a quick access to commonly used formats.

Match as whole words only

When searching for a matching date/time pattern, match only if the found pattern is a whole word, i.e. surrounded by word boundaries.

For example, the 4 digit year pattern (YYYY) will be found in "foo2025bar" if this option is disabled, but not found if this option is enabled.

Skip extension

Exclude the file extension from processing.

Use custom short months

Use a custom list of short month names when searching or formatting the short month name pattern (MMM). Month names are separated by a comma. By default, month names associated with your system locale will be used.

Use custom long months

Use a custom list of long month names when searching or formatting the long month name pattern (MMMM). Month names are separated by a comma. By default, month names associated with your system locale will be used.

Adjust time by

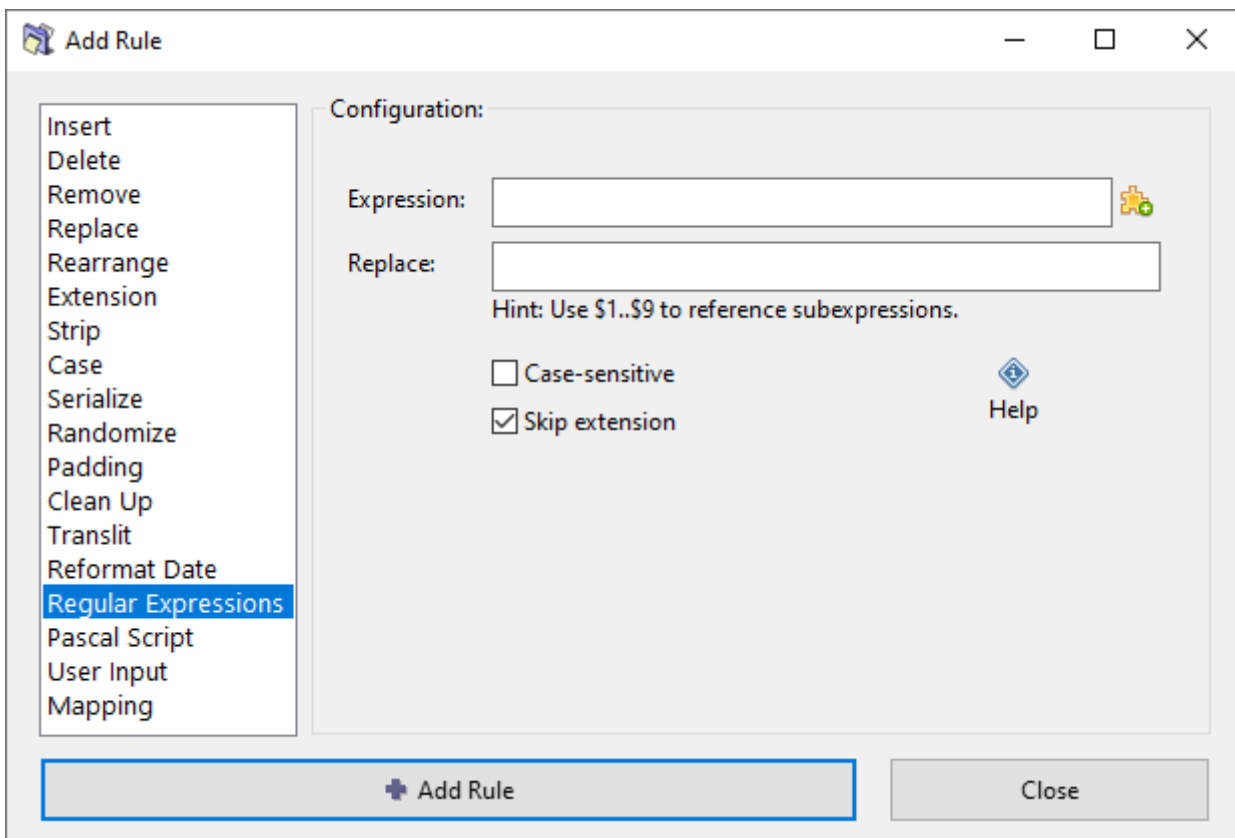
This option lets you adjust date and time by the specified number of years, months, days, hours, minutes, seconds, and milliseconds.

Regular Expressions rule

This rule allows finding and replacing patterns, and performing sophisticated manipulations with filenames using [Regular Expressions](#).

The Regular Expressions processor uses a special syntax for describing search and replace patterns, which are very powerful and well worth learning. If you are already familiar with them, please note that the syntax and features available in ReNamer are a little different from those in mainstream processors like [Perl](#).

You can find many use cases and examples in the [User Forum](#), where you might find your particular case already solved.



Options

Expression

The expression (pattern) you want to find.

See the [Regular Expressions reference](#) to learn about the syntax and features.

The menu button on the right side of the expression field offers a few examples:

.	Any single character
.*	Zero or more of any characters
.*+	One or more of any characters
.*+?	One or more of any characters, but fewer as possible
.{3}	A set of 3 of any characters
.{3,}	A set of 3 or more of any characters
.{2,4}	A set of 2 to 4 of any characters
(.+)	One or more of any characters, grouped into subexpression
[abc]	Any single character from the list (e.g. "a" or "b" or "c")
[^abc]	Any single character except the ones from the list
[a-z]	Any single character from the range (e.g. "a" to "z")
aa bb cc	Any one occurrence of the alternatives (e.g. "aa" or "bb" or "cc")
\\	Use an escape character to match an actual backslash "\"
\w	Any single alphanumeric character (including an underscore)
\s	Any single space character (space, tab, new line)
\b	Word boundary
\A	Start of the input text
\Z	End of the input text

Replace

The pattern that replaces the found expression.

Captured subexpressions (parenthesized groups in the pattern) can be referenced via short codes `$1`, `$2`, `$3`, and so on. The entire match is accessible via short code `$0`.

Case-sensitive

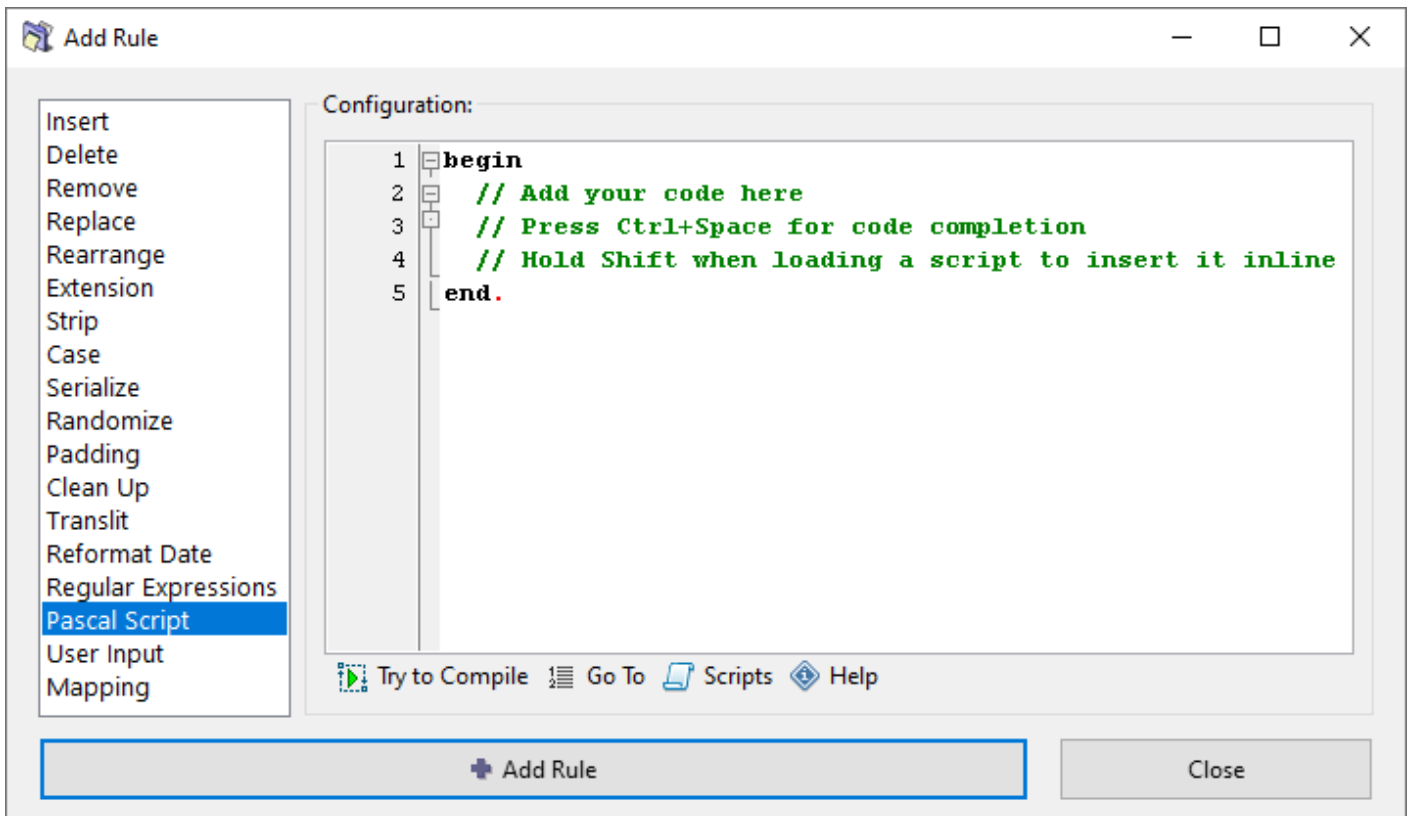
Match case exactly when searching for the expression. When enabled, "Apple" will not match "apple". When disabled, uppercase and lowercase letters are treated as equivalent.

Skip extension

If checked, the file extension will be excluded from processing and will remain unaffected.

Pascal Script rule

This rule allows you define your own renaming logic and integrate external tools. It uses the [Pascal Script programming engine](#) with syntax and conventions similar to Delphi/Pascal language. The rule comes with a number of preloaded example scripts. We will see how to use them, and how to write your own script.



Writing your own script

To write your own script, you need to have knowledge of Pascal Script syntax and conventions, and be familiar with the list of built-in types and functions. All that you can learn using a dedicated [Pascal Script reference](#) section.

Let's assume that you already know how to write scripts.

The general workflow is as follows:

1. Write your script, or load one from an external source.
 - You can also use the built-in examples via the *Scripts* menu, or take a script from the [Examples](#) and the [Library](#) reference sections. More on that in the section below.

2. Compile the script by pressing the *Try to Compile* button.
 - If an error message comes up, troubleshoot the script. The error message usually includes the line number of the problematic statement in the script. Press the *Go To* button and enter that line number to locate the faulty statement quickly. Once you correct the fault, press the *Try to Compile* button again. Repeat until you see a *Compiled successfully!* message.
3. Add the rule to your renaming stack by clicking the *Add Rule* button.
 - Optionally, you can save the script to the built-in repository via the *Scripts* menu, for quick access and reuse later.

The editor provides basic syntax highlighting and auto-completion for built-in functions, via the `CTRL+Space` shortcut.

To get you started, let's try this example script, which will simply prefix each filename with "Example" text:

```
begin
  FileName := 'Example ' + FileName;
end.
```

Loading a script

You can access your local scripts repository via the the *Scripts* menu. The menu will show you a list of preloaded example scripts and any scripts which you have saved previously. Click on any of the script names in the menu to load it into the editor.

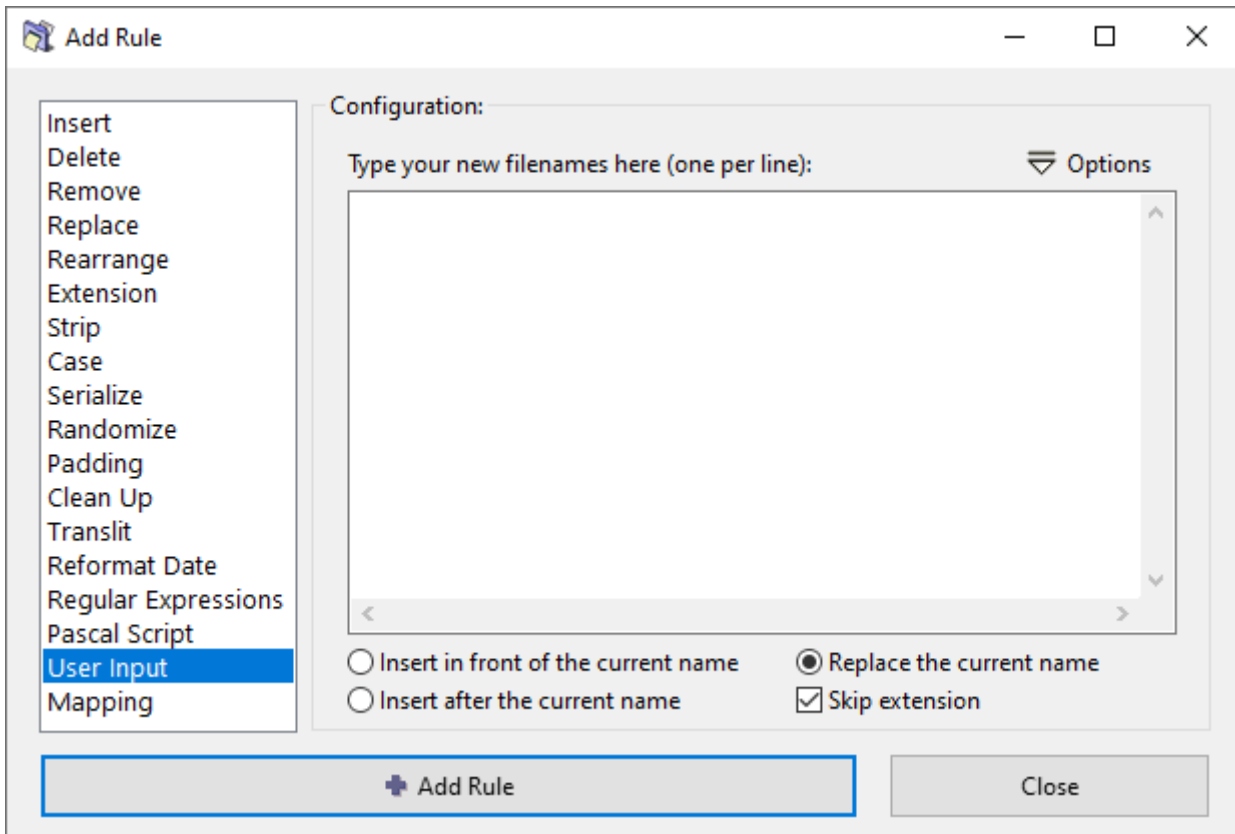
You can assemble your script using multiple scripts by holding down the `SHIFT` key when loading a script, which will insert it inline instead of replacing the whole script, just make sure to combine them correctly.

In addition to the preloaded example scripts, you can find more examples and user contributed scripts at:

- [Examples repository](#) for simple how-to examples.
- [Library repository](#) for advanced examples and 3rd party tool integrations.
- [User Forum](#) for user contributions.

User Input rule

This rule replaces the original filenames with the names taken from a fixed list.



The n -th line in the list serves as the new name for the n -th file in the *Files* pane.

Naturally, the list should contain names for all the files loaded in the *Files* pane.

- If the list is shorter, then some of the files will not get a new name and won't get renamed.
- If the list is longer, some of the names will remain unused, but all files in the *Files* pane will be renamed.

There are three ways to create the list:

1. Manually type the list of new names, one name per line.
2. Copy the list of names from another application via the clipboard, using either the `CTRL+V` shortcut or the *Options* menu.
3. Load a list from a text file, using the *Options* menu.

See also the [Export menu](#) for additional options on export and import of files and their new names.

General options

Insert in front of the current name

Insert the new name in front of the original name.

For example, if the original name is "Dog" and new name is "Hot", the result will be "HotDog".

Insert after the current name

Insert the new name after the the original name.

For example, if the original name is "Dog" and new name is "Hot", the result will be "DogHot".

Replace the current name

Replaces the existing filename with the new name.

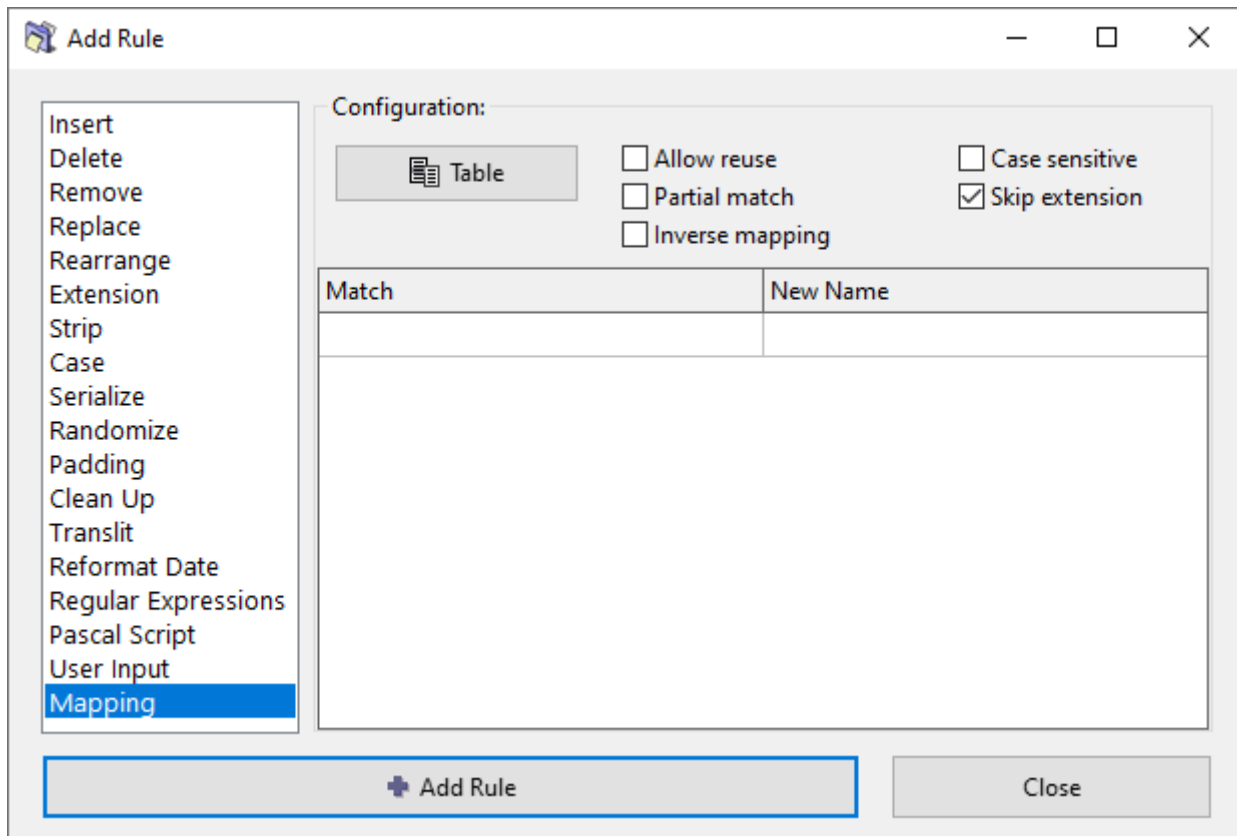
The effect on the file extension is determined by the *Skip extension* option.

Skip extension

If checked, the file extension will be excluded from processing and will remain unaffected.

Mapping rule

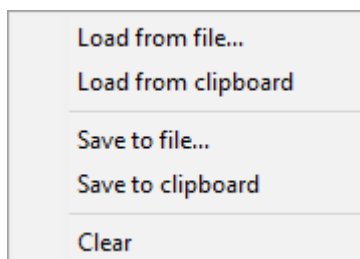
This rule allows you to map a list of old filenames to a list of new filenames, which can be externally managed and loaded into the rule via clipboard or CSV file. It also allows you to capture the current set of files and their generated new names as a rule set, and then applied to a different set of files via this rule.



Options

Table menu

Click the *Table* button to access the menu for managing the mappings table.



- *Load from file...* – Load a list of mappings from a comma separated (CSV) file or a tab separated (TXT) file.
- *Load from clipboard* – Load a list of mappings from clipboard, in a tab separated format.
- *Save to file...* – Save the current list of mappings to a comma separated (CSV) file or a tab separated (TXT) file.
- *Save to clipboard* – Save the current list of mappings to clipboard, in a tab separated format.
- *Clear* – Clear the current list of mappings.

The loading and saving of mappings via clipboard allows a quick exchange of data between the rule configuration and an external application such as *Microsoft Excel*, *LibreOffice Calc*, and other spreadsheet editors.

You can also use the [Export menu](#) to convert the current list of filenames and their generated new names to a Mapping rule, by using the *Export names and new names to clipboard* option and then loading them into a Mapping rule.

Allow reuse

Normally, once a mapping has been applied to a file, that mapping is removed from the list of available mappings, so the remaining files in the same iteration of the preview process will not be allowed to reuse the same mapping. In other words, the mappings have a strict *one-to-one* application.

If this option is enabled, all mappings can be reused multiple times, for all matching files. In other words, the mappings have a relaxed *one-to-many* application.

Partial match

Apply the mapping if a match is found against a part of a filename.

If this option disabled, the whole filename must match for the mapping to be applied.

Inverse mapping

Swap the *Match* and *New Name* entries in the table, inverting the mappings.

Case sensitive

Match case exactly.

When enabled, "Apple" will not match "apple". When disabled, uppercase and lowercase letters are treated as equivalent.

Skip extension

Exclude the file extension from processing so it remains unaffected.

If the *Match* entries in the table include file extensions, leave this option unchecked so that the full filename is used for matching.

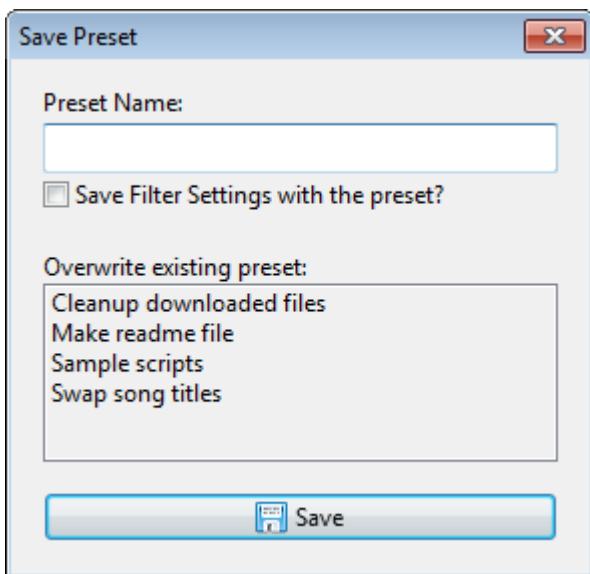
Operational Guides

In-depth guides and reference material for features, tools, and specific renaming scenarios.

Using presets

A **preset** is a named collection of [rules](#) that can be saved and reloaded on demand. Presets can also store [filter settings](#), making it easy to reuse complete renaming configurations across sessions. Presets can also be used for unattended renaming via [command line](#).

Saving a preset



1. Set up the desired rules in the **Rules pane**. Optionally configure [filter settings](#) to include with the preset.
2. Press `Ctrl+S`, or use *Presets » Save As* from the main menu.
3. Enter a name for the preset, or select an existing preset from the list to overwrite it.
4. Check the option to include filter settings if needed.
5. Click Save.

Press `Esc` or close the dialog to cancel without saving.

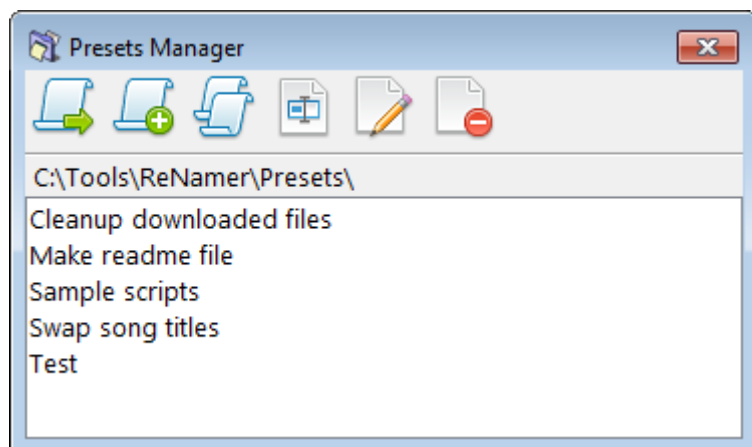
Loading a preset

Presets can be loaded via *Presets » Load* in the main menu or via the [Presets Manager](#).

Loading a preset replaces the current set of rules. To add rules from a preset to the existing stack instead of replacing it, hold `Shift` while selecting the preset from *Presets » Load*.

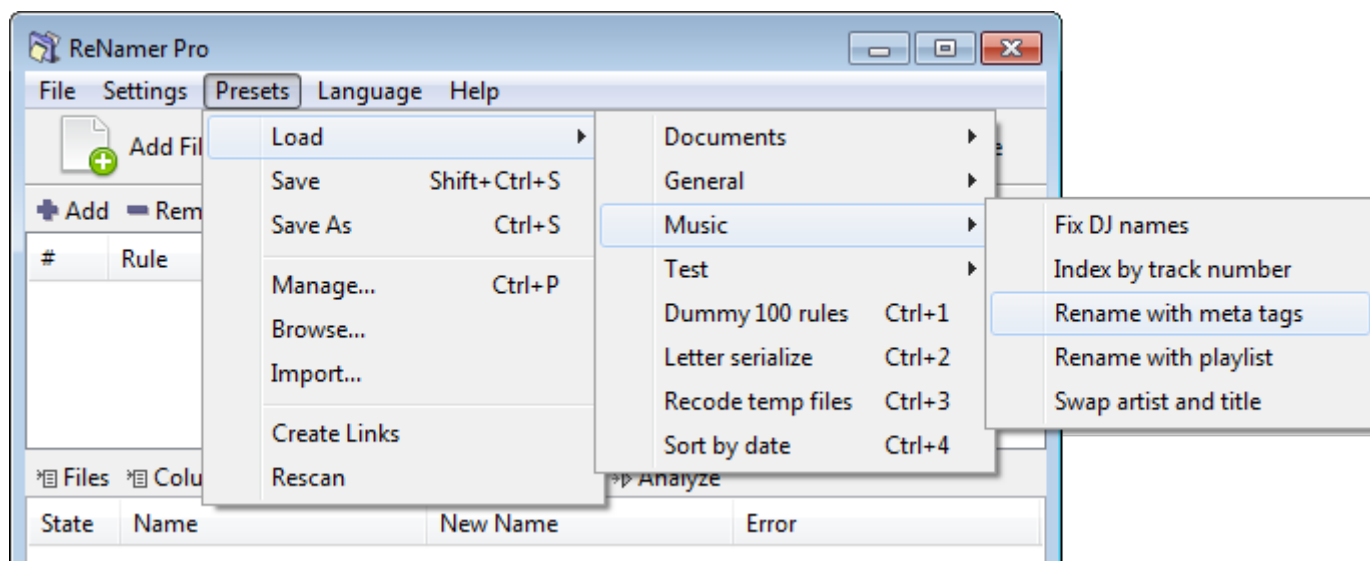
For quick access, the first nine presets can be loaded directly using `Ctrl+1` through `Ctrl+9`. Presets are ordered alphabetically, so their shortcut assignment can be controlled by renaming them.

Managing presets



The [Presets Manager](#) provides a dedicated interface for loading, appending, copying, renaming, editing, and deleting presets. Open it via *Presets » Manage* or by pressing `Ctrl+P`.

Presets directory structure



Preset files can be organised into subfolders, which is useful when managing a large collection. To save a preset into a subfolder, include the folder name in the preset name when saving, using a backslash as a separator. For example, entering `Music\Cleanup` saves a preset named `Cleanup` inside a `Music` subfolder.

Presets can also be rearranged manually by browsing to the presets folder via *Presets » Browse* and moving the files in Windows Explorer.

Locating and transferring presets

Presets are stored as `*.rnp` files in a `Presets` folder on disk. The exact location depends on the installation type:

- In a portable installation, the `Presets` folder is located in the same directory as `ReNamer.exe`.
- In a standard installation, it is located in the user profile folder.

The full path is shown at the top of the [Presets Manager](#) window. The *Presets » Browse* menu option opens that folder directly in Windows Explorer.

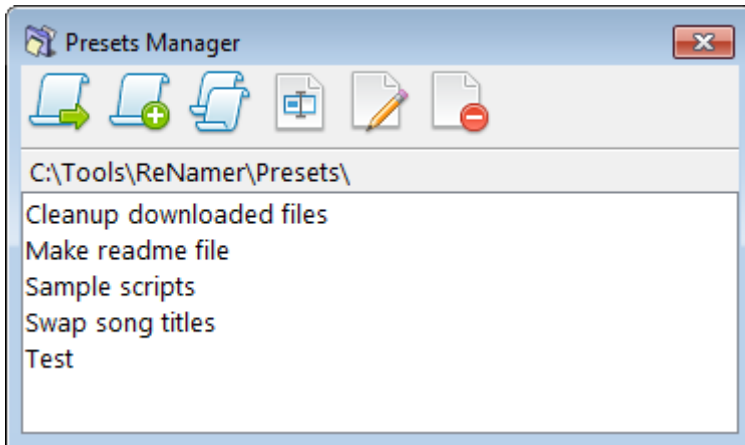
To transfer presets to another computer or installation, copy the `*.rnp` files from the source presets folder and place them in the presets folder on the target, or import them via *Presets » Import*.

See also [Application Data Storage](#) for more information on locating application data.

Presets Manager

The **Presets Manager** provides options for managing your collection of presets, including loading, appending, duplicating, renaming, and deleting.

Open it via *Main Menu » Presets » Manage*, or press **Ctrl+P** from the main window.



Load a preset

Load the selected preset, replacing the current set of rules.

You can also press **Enter** to load the currently selected preset.

- Hold **Ctrl** when loading a preset to automatically close the window.
- Hold **Shift** when loading a preset to append rules from the selected preset to the current set of rules.

Append a preset

Append rules from the selected preset to the current set of rules.

This is a useful way to merge rules from multiple presets.

Copy a preset

Create a duplicate preset file with same rules.

The duplicate will have the same base name with **COPY** appended to the end.

This is useful as a starting point when creating a variation of an existing preset.



Rename a preset

Rename the selected preset. A prompt will appear to enter the new name.

You can use path delimiters in the new name to create a [directory structure for presets](#).



Edit a preset

Manually edit a preset file using a text editor (e.g. Notepad). This is for advanced users only.

Warning: Avoid manual editing unless you know exactly what you are doing. Small mistakes can produce a corrupted preset. The normal approach is to load a preset through the main interface, [modify the rules](#), and save the preset again.

Example preset file content:

```
[Rule0]
ID=Insert
Config=TEXT:Example;WHERE:1;POSITION:1;INSERTAFTERTEXT:;INSERTBEFORETEXT:;RIGHTTOLEFT:0;SKIPEXTENSION:1
Marked=1
Comment=This is an example rule
```

Syntax:

- Rule definitions are grouped in sections headed by `[Rule0]`, `[Rule1]`, and so on.
- `ID=` identifies the type of rule.
- `Config=` contains the encoded rule configuration.
 - Parameters are separated by `;`.
 - Each parameter follows `Name:Value` format.
 - Parameter values are [URL-encoded](#).
- `Marked=` stores the marked/unmarked state of the rule.
- `Comment=` stores an annotation for the rule.



Delete a preset

Delete the selected preset.

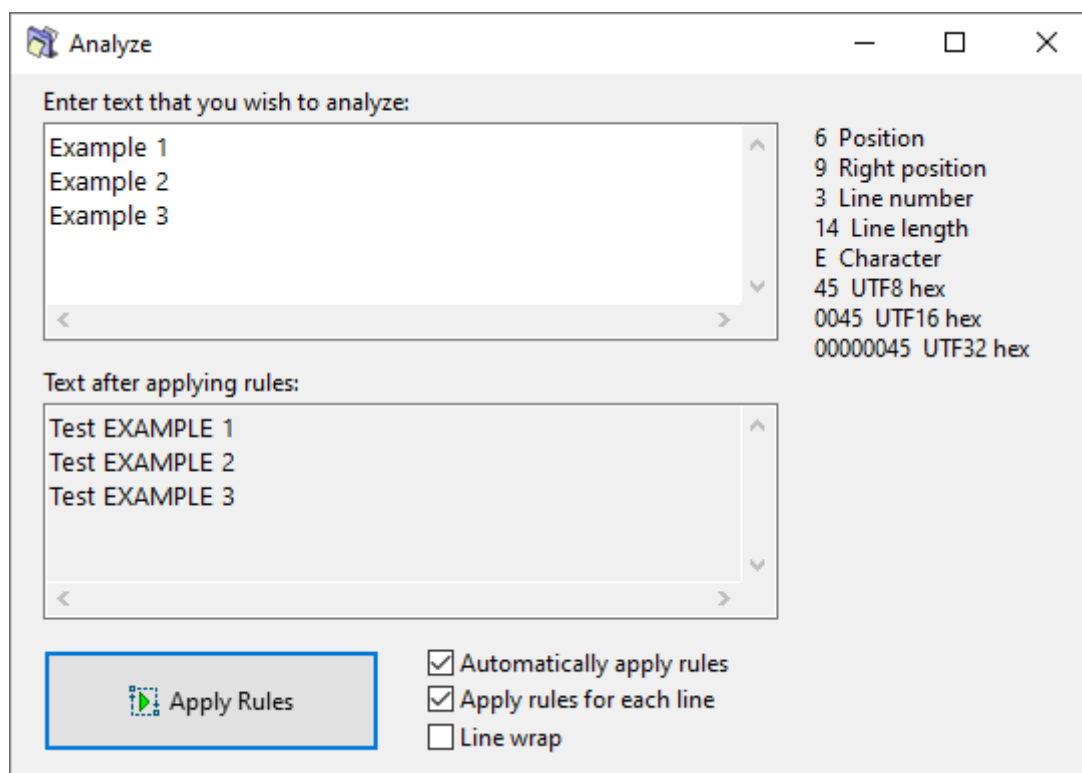
This action permanently deletes the preset file from disk.

Analyze tool

The **Analyze tool** opens a window where you can enter arbitrary text and test your current rules against it, without renaming any actual files. Open it from the [Files and Tools](#) toolbar or press **Shift+A** in the main window.

The input text can be typed manually or loaded from file names in the *Files* pane using the *Analyze Name* option in the [Files context menu](#). Rules from the *Rules* pane are applied to the input and the result is displayed alongside it. This is particularly useful for verifying rule behaviour before committing to a rename — for example, when setting up an [Insert rule](#) to confirm the exact insertion position.

Navigating the cursor over the input and output text with the keyboard or mouse displays the cursor position and selection information in the panel to the right.



The available options are:

Option	Description
<i>Automatically apply rules</i>	Apply rules whenever the input text changes, so you do not need to press the <i>Apply Rules</i> button after each edit.

Option	Description
<i>Apply rules for each line</i>	Apply the rules to each line individually rather than to the whole text at once.
<i>Line wrap</i>	Wrap long lines to fit within the width of the input field.

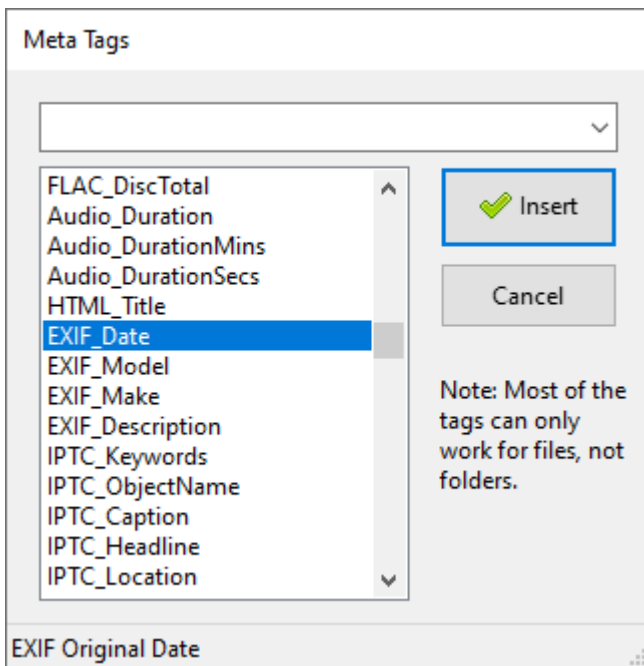
Note: Using *Line wrap* together with *Apply rules for each line* can produce unexpected results. Line wrapping introduces artificial line breaks, and rules will be applied to each of those wrapped fragments as if they were separate lines.

Meta Tags

Meta Tags allow users to extract meta information associated with files and use it for renaming. ReNamer supports a wide range of built-in meta tags, while 3rd party tools can be used for extracting tags which are currently not supported directly.

There are two different ways for using the built-in meta tags:

- Using the [Insert rule](#), the [Replace rule](#), or the [Rearrange rule](#) to insert meta data directly into the filename,
- Using [CalculateMetaTag](#) function in the [Pascal Script rule](#) to perform more complex manipulations with the meta data.



List of built-in Meta Tags

There are more than 100 built-in meta tags of various kinds:

- File information (size, dates, name, path, extension, and more).
- Audio tags (title, artist, album, genre, track number, duration, and more) in MP3, WMA, FLAC files.
- Photo tags (date, camera make and model, description, and more) in EXIF/IPTC/TIFF formats.
- Graphics properties (width and height, aspect ratio, number of pixels).
- Document summary information (title, subject, author, page count).

- File hashes (CRC32, MD5, SHA1, SHA256, SHA512).
- Email properties (date sent, subject, sender) for EML and MSG (Outlook) formats.
- Video properties (width and height, frames, duration, and more) for AVI format.
- Font properties (name, family, revision).
- Torrent file properties (name, hash, size, file count, and more).
- Automatically detected file extension using built-in [Binary Signatures](#).
- File version information (product, company, version number, and more).
- Miscellaneous (HTML title, current date, clipboard contents).

Editing Meta Tags

ReNamer is a file renaming tool, not a meta tag editing tool. ReNamer does not edit meta tags and this feature will never be supported.

Most of meta tags (e.g. ID3, EXIF) are not part of the names of the files, instead they are contained inside certain files. To edit meta tags, please use a suitable meta tag editor such as [mp3BookHelper](#), [mp3Tag](#) or [TagScanner](#) (for various tags in digital audio files) or [PhotoME](#) (for EXIF tags).

Checksum-based tags (MD5, SHA1, CRC32) are non-editable: They are calculated by the system from the contents of the file. You can use them to check the integrity of any file using software like [Exactfile](#).

Extracting Meta Tags with 3rd party tools

There are 3rd party tools which let extract meta data from various file formats for which ReNamer does not have a built-in support. In such cases the output of these tools can be integrated and used within ReNamer with the help of the [Pascal Script rule](#).

Sample scripts which demonstrate the use of this method can be found in the [Pascal Script Library](#) section.

Here is list of some 3rd party tools which can be used for extracting meta data: [ExifTool](#), [FFmpeg](#), [Exiv2](#), [MediaInfo](#).

Date and Time format

Date and time format is used primarily in [Meta Tags](#) and the [Pascal Script rule](#) to format date and time values.

The format consists of a series of placeholder characters (or variables) corresponding to year, month, day, hour, and so on. You can change the format in the [Settings](#).

Format variables

Below is a list of variables which you can use.

Variable	Description
d	Displays the day as a number without a leading zero (1-31).
dd	Displays the day as a number with a leading zero (01-31).
ddd	Displays the day as an abbreviation (Sun-Sat).
dddd	Displays the day as a full name (Sunday-Saturday).
e	Displays the year in the current period/era as a number without a leading zero (Japanese, Korean and Taiwanese locales only).
ee	Displays the year in the current period/era as a number with a leading zero (Japanese, Korean and Taiwanese locales only).
g	Displays the period/era as an abbreviation (Japanese and Taiwanese locales only).
gg	Displays the period/era as a full name. (Japanese and Taiwanese locales only).
m	Displays the month as a number without a leading zero (1-12). If the "m" specifier immediately follows an "h" or "hh" specifier then minute is displayed.
mm	Displays the month as a number with a leading zero (01-12). If the "mm" specifier immediately follows an "h" or "hh" specifier then minute is displayed.
mmm	Displays the month as an abbreviation (Jan-Dec).
mmmm	Displays the month as a full name (January-December).

Variable	Description
yy	Displays the year as a two-digit number (00-99).
yyyy	Displays the year as a four-digit number (0000-9999).
h	Displays the hour without a leading zero (0-23).
hh	Displays the hour with a leading zero (00-23).
n	Displays the minute without a leading zero (0-59).
nn	Displays the minute with a leading zero (00-59).
s	Displays the second without a leading zero (0-59).
ss	Displays the second with a leading zero (00-59).
z	Displays the millisecond without a leading zero (0-999).
zzz	Displays the millisecond with a leading zero (000-999).
am/pm	Use the 12-hour clock notation for "h" and "hh" specifiers, and display "am" or "pm" accordingly. † The "am/pm" specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
a/p	Use the 12-hour clock notation for "h" and "hh" specifiers, and display "a" or "p" accordingly. † The "a/p" specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
"xx"	Characters enclosed in single or double quotes are displayed as-is, and the formatting is not applied to them.

Examples

For example, if we assume that the date is *25 October 2007* and the time is *16:59:00*, then sample formats and their outputs would be:

Format	Output
<code>dd-mm-yyyy hh.nn.ss</code>	Produces 25-10-2007 16.59.00 , which is an easily readable format for the date and time.
<code>"It's" dddd, hh:nn:ss AM/PM</code>	Produces It's Thursday, 4:59:00 PM , which is unsuitable for filenames because of : forbidden character.
<code>yyyymmddhhnnss</code>	Produces 20071025165900 , which is ideal for serializing files because the filename is relatively short, most probably unique, contains only digits, and also makes files automatically sorted in chronological order.

† Prior to *ReNamer v6.0*, **am/pm** and **a/p** specifiers affected only the immediately preceding **h** and **hh** specifiers.

Sorting files

The order of items in the *Files* pane can affect the renaming outcome. This is most notable in two situations:

1. When [renaming folders](#) and their contents together.
2. When using the [Serialize rule](#) to assign sequential numbers to file names.

ReNamer supports two sorting modes:

- **Manual sorting** — a one-time reorder applied to the current list.
- **Automatic sorting** — items are always kept sorted, and newly added items are sorted automatically.

Manual sorting

To sort items manually, select a group of items and drag them to a new position using the mouse. Alternatively, use the `Ctrl+Up` and `Ctrl+Down` shortcuts, or the move commands in the [Files context menu](#).

With manual sorting, newly added items are always placed at the end of the list, in the order they were added.

Automatic sorting

Click a column header in the *Files* pane to enable automatic sorting by that column. Clicking the same column header again toggles between ascending and descending order. The active sort direction is indicated by a small up/down arrow in the column header.

Sorted by Name (ascending)				Sorted by Created Date (descending)			
State	Name ▲	New Name	Created	State	Name	New Name	Created ▼
✓ →	File 1	File 1	09/08/2009 12:39:26	✓ →	File 2	File 2	09/08/2009 12:39:27
✓ →	File 2	File 2	09/08/2009 12:39:27	✓ →	File 1	File 1	09/08/2009 12:39:26
✓ →	File 3	File 3	09/08/2009 12:38:47	✓ →	File 4	File 4	09/08/2009 12:39:23
✓ →	File 4	File 4	09/08/2009 12:39:23	✓ →	File 5	File 5	09/08/2009 12:39:21
✓ →	File 5	File 5	09/08/2009 12:39:21	✓ →	File 3	File 3	09/08/2009 12:38:47

Not all columns are visible by default. Right-click on the column headers to open the [Files header menu](#), where you can toggle additional columns on or off.

To retain the same sorting order in future sessions, enable the *Remember sorting options* setting in [Settings](#).

Canceling automatic sorting

To cancel automatic sorting, right-click on the column headers and select *Cancel sorting* from the context menu. Once canceled, newly added items will appear at the end of the list in the order they are added.

Sorting by multiple columns

To sort by more than one column, apply sorting in multiple steps (one column at a time) starting with the least dominant column and ending with the most dominant.

For example, to sort files by folder name, with files within each folder sorted by modification date: first sort by the modification date column (least dominant), then by the folder name column (most dominant). The order established in the first step is preserved in the second step for items that are equal in the dominant column — that is, items within the same folder.

Preserving the original order

If you have already arranged items in a specific order (for example, in Windows Explorer) and want ReNamer to preserve that order, disable automatic sorting *before* adding items.

Note that due to a quirk of Windows, the order in which a selection of files is added may depend on the focused item. To preserve your intended order, always ensure the first item in the selection has focus. For example, when using drag-and-drop, drag the selection from the very first file.

When adding folders, their contents are added in the order determined by the file system, since folders are automatically scanned for contained items.

Renaming folders

By default, ReNamer is configured to rename files only. When a folder is added, it is scanned recursively and the files it contains are added to the renaming list — the folder itself is not.

To rename folders themselves, open [Filter settings](#) and enable the *Add folders as files* option. To rename only folders without their contained files, also disable the *Add files within folders* option.

When both folders and their contents are included in the same rename operation, the order in which items are processed becomes critical. See the *Conflicting renaming order* section below.

Extensions in folder names

The *Skip extension* option found in many [renaming rules](#) can also affect folders. A separate setting, *Folders also have extensions* found in [Settings](#), controls whether ReNamer treats periods in folder names the same way as it does in file names. When enabled, the portion to the right of the last period in a folder name is treated as the extension.

For example, consider folders named `01.Introduction`, `02.Basics`, and `03.Advanced`. With *Folders also have extensions* enabled, `01`, `02`, and `03` are treated as base names, while `Introduction`, `Basics`, and `Advanced` are treated as their extensions.

Conflicting renaming order

Renaming a folder immediately changes the file system path of everything it contains. If both a folder and its contents are listed for renaming in a single operation, the order of items matters.

Items are processed from top to bottom. If a parent folder is renamed before its contained items, its path changes immediately in the file system — but ReNamer still holds the old paths for those items. When ReNamer then attempts to rename them at their original (now non-existent) paths, it cannot find them and the renames fail.

To avoid this, parent folders must always appear below their contained items in the list.

This is most easily achieved using *Sort by path for renaming folders* in the [Options menu](#), or manually by sorting the list in descending order by the *Folder* or *Path* column, see [Sorting files](#) for details.

Example

Suppose `C:\Folder` contains `C:\Folder\Document.txt`, and both are listed for renaming with `Folder` appearing above `Document.txt` in the list. When `Folder` is renamed to `NewFolder`, the document's path in the file system immediately becomes `C:\NewFolder\Document.txt`. However, ReNamer still has the document listed at its original path `C:\Folder\Document.txt`, which no longer exists, so the rename fails.

Problem occurred

Items as they appear after being added, without sorting.

Preview				Rename			
State	Folder	Name	New Name	State	Folder	Name	New Name
☒ →	C:\	Folder	NEW_Folder	☒ ✓	C:\	NEW_Folder	
☒ →	C:\Folder\	Subfolder	NEW_Subfolder	☒ ✗	C:\Folder\	Subfolder	NEW_Subfolder
☒ →	C:\Folder\	File1	NEW_File1	☒ ✗	C:\Folder\	File1	NEW_File1
☒ →	C:\Folder\	File2	NEW_File2	☒ ✗	C:\Folder\	File2	NEW_File2
☒ →	C:\Folder\Subfolder\	File3	NEW_File3	☒ ✗	C:\Folder\Subfolder\	File3	NEW_File3

Problem solved

Items after sorting in descending order by the *Folder* column.

Preview				Rename			
State	Folder ▼	Name	New Name	State	Folder ▼	Name	New Name
☒ →	C:\Folder\Subfolder\	File3	NEW_File3	☒ ✓	C:\Folder\Subfolder\	NEW_File3	
☒ →	C:\Folder\	Subfolder	NEW_Subfolder	☒ ✓	C:\Folder\	NEW_Subfolder	
☒ →	C:\Folder\	File1	NEW_File1	☒ ✓	C:\Folder\	NEW_File1	
☒ →	C:\Folder\	File2	NEW_File2	☒ ✓	C:\Folder\	NEW_File2	
☒ →	C:\	Folder	NEW_Folder	☒ ✓	C:\	NEW_Folder	

Moving files to another folder

ReNamer can move files and folders to a different location as part of the renaming process. You can move everything to a single target folder, or distribute files into multiple folders based on their properties.

Moving to a folder

To move items to a different folder, include the target folder path in the new name. You can use either an absolute path (e.g. `C:\Example\`) or a relative path (e.g. `..\Example\`).

For example, to move a set of files to `C:\New Folder`, add an [Insert rule](#) with `C:\New Folder\` as the prefix. The preview will show:

Name	New Name
Text.txt	<code>C:\New Folder\Text.txt</code>
Song.mp3	<code>C:\New Folder\Song.mp3</code>
Document.doc	<code>C:\New Folder\Document.doc</code>

Proceed with renaming as usual. If the target folder does not already exist, ReNamer will create it automatically.

Tip: Make the *New Path* column visible in the files table to see the resolved destination for each item. This is especially useful with relative paths, since ReNamer will display the fully resolved absolute path.

Sorting files into multiple folders

Instead of moving all files to a single folder, you can distribute them into different folders based on their properties — sometimes called *binning*. To do this, use a [meta tag](#) in the path instead of a hard-coded folder name.

You can build a folder hierarchy in a single operation by using multiple meta tags in the path. Folders that do not exist will be created automatically.

For example, inserting `:EXIF_Date:\` as a prefix will group photos into subfolders named by their capture date. Inserting `:MP3_Artist:\:MP3_Album:\` will group music files into nested subfolders by artist and album.

Copying files

A normal renaming operation moves files by modifying their path. Copying is different, it involves creating a duplicate of the file, and is not supported directly.

However, to achieve the same effect, duplicate your files first using an external tool such as Windows Explorer and then rename the copies in ReNamer as usual.

Validation

Validation is a process that analyzes the list of files and their target destinations to identify common problems before renaming begins. When a potential problem is detected, a warning is raised so it can be resolved during the preview stage rather than resulting in a failed rename.

By default, validation runs automatically during the preview process. It can also be triggered on demand via *Validate new names* in the [Options menu](#). Automatic validation can be disabled in [Preview settings](#), which may be worth considering when processing a large number of files, as it may take a noticeable amount of time.

Users should generally eliminate all warnings before proceeding. Items that still have warnings at the time of renaming may [fail to rename](#).

Duplicated destination paths

Two or more files in the list are set to be renamed to the same destination path, which would result in a conflict. Adjust the renaming rules or [manually edit](#) the affected new names to ensure all destination paths are unique.

New path contains forbidden characters

The new filename includes one or more characters that are not permitted in Windows file paths. Review and correct the renaming rules to avoid producing forbidden characters in the output.

New path is already taken

A file already exists at the destination path, which would cause a name conflict. See [Failed Renaming](#) for options on how to resolve this.

New path exceeds maximum length

The resulting file path would exceed the maximum length permitted by Windows. See [Long paths](#) for information on how to work around this limit.

Failed renaming

The renaming operation can fail for a variety of reasons. The built-in [validation process](#) attempts to identify common problems before renaming by raising warnings during the preview stage, but it cannot catch every possible issue.

The most common reasons for a failed renaming operation are described below.

File path is too long

The total length of the file path exceeds the system limit. In Windows, the maximum file path length is **260 characters**, as defined by the `MAX_PATH` constant.

As a workaround, you can use the "long path" specification by prepending the file path with `\\?\`, which effectively raises the maximum path length to **32,767 characters**. See [Long paths](#) for more information.

Alternatively, shorten the filename or the name of one of its parent folders to bring the total path length within the limit.

Destination file already exists

A file with the target name already exists at the destination, causing a name conflict. There are several ways to resolve this:

- Enable *Fix conflicting new names on preview* in [Preview settings](#) to automatically resolve conflicts by appending a number suffix during preview.
- Use *Fix Conflicting New Names* from the [Options menu](#) to resolve conflicts on demand.
- [Manually edit the new name](#) of the conflicting file.
- Rename or move the existing file at the destination to eliminate the conflict.

Source file does not exist

The file listed in the *Files* pane no longer exists at its original path. This can happen if:

- The file was moved or renamed outside of ReNamer.
- The file's parent folder was [renamed first](#), making the stored path stale.

To fix this, remove the affected items from the *Files* pane and re-add them using their current path.

File is in use by another program

The file is currently locked by another process and cannot be renamed. Use a utility such as *Windows Task Manager* or *Process Explorer* to identify and close the program holding the file. If the file is still being downloaded, simply wait for the download to complete before renaming.

Insufficient privileges

You do not have the necessary permissions to rename the file. If you are not the administrator, contact the owner or admin. If you are the administrator, check the permissions of the containing folder.

Invalid destination path

The new filename contains one or more characters that are **not permitted** in Windows file paths:

\ / : * ? " < > |

Remove or replace any of these characters in the new name.

Parent path changed earlier in the list

A parent folder of the affected file was renamed earlier in the same renaming operation, which made the file's stored path stale. This can occur when renaming both folders and their contents at the same time.

To avoid this, always rename the contents of a folder before renaming the folder itself. This can be achieved by sorting the *Files* pane by the *Folder* or *Path* column in descending order before running the operation.

For more information, see [Renaming folders](#).

Manual editing

In addition to combining rules to generate new names automatically, you can also make manual changes to the new names directly in the *Files* pane. This is useful for minor tweaks before proceeding to [renaming](#).

Note: ReNamer is designed primarily for automatic, rule-based renaming. Manual editing is a convenience feature for small final adjustments. Every time new names are regenerated by a [preview operation](#), all manual changes will be lost.

How to edit a name

Step 1

In the *Files* pane, select the row of the file whose name you want to edit.

State	Name	New Name
☒ →	Name1	Name1
☒ →	Name2	Name2

Step 2

Enter edit mode using any of the following methods:

- Press **F2**
- Right-click the file name and select *Edit New Name*
- Slow-click (single click with a short pause) on the file name

The *New Name* column switches to an editable field with the current name selected. You can type a completely new name, or use the arrow keys to position the cursor and make targeted changes.

State	Name	New Name
☒ →	Name1	Name1
☒ →	Name2	Name2

Step 3

Press **Enter** to confirm the change (or click anywhere outside the edit field). The edited name will be shown as the new name, although the file has not been renamed yet. If unhappy with the changes, press **Esc** to cancel and discard the edit.

State	Name	New Name
☒ →	Name1	Name1
☒ →	Name2	Name 3

Step 4

Click the *Rename* button to perform the actual renaming.

State	Name	New Name
☒ ✓	Name1	
☒ ✓	Name 3	

Manual changes can get lost

Manual changes are discarded whenever new names are regenerated by a [preview operation](#). This can be triggered:

- **Manually** — by clicking the *Preview* button
- **Automatically** — when [auto-preview](#) fires due to added files or changed rules

This happens because manual changes are not part of the rules stack. Name generation always starts from the original file name and passes through all rules in sequence, so any manually entered name is simply overwritten. A manual change is preserved only as long as no preview is generated after it was made.

If you need to preserve manual changes, you have the following options:

1. Perform the renaming immediately after making manual changes to commit them, then continue adding rules as needed.
2. After making manual changes, export the new names to the clipboard via the [Export menu](#), then add a [User Input rule](#) and paste the names from the clipboard. This captures the current state of all new names as a rule.
3. After making manual changes, export both the original and new names to the clipboard via the [Export menu](#), then add a [Mapping rule](#) and load the new name mappings from the clipboard. This maps each original name to its intended new name as a rule.

Command line

ReNamer supports several command line parameters for launching the program with files pre-loaded, applying presets automatically, and performing unattended renaming operations.

The general format is:

```
"ReNamer.exe" /parameter [arguments]
```

Note that multiple parameter types cannot be combined together.

The examples below do not include the full path to `ReNamer.exe`, as it can differ between installations. To run these commands, either add the ReNamer installation folder to the Windows `PATH` environment variable, or use the full path to the executable in each command, e.g.

```
"C:\Program Files\ReNamer\ReNamer.exe".
```

Command line parameters can also be appended to any ReNamer shortcut via its *Properties*, so that the shortcut always launches ReNamer with the desired options.

Commands

Add files

```
"ReNamer.exe" <files>
```

Launch ReNamer and add the specified files and folders to the *Files* pane. You will then need to [add rules](#), [preview](#), and [rename](#) the files yourself.

Replace `<files>` with absolute paths to the files and folders you want to rename, separated by spaces. Wildcard paths are also supported, e.g. `"C:\Pictures\*.jpg"`.

- Even if a ReNamer window is already open, this command will always launch a new window.
- When adding folders, [Filter settings](#) determine which files are included.

Example:

```
"ReNamer.exe" "C:\Folder" "C:\Pictures\*.jpg"
```

This launches ReNamer and adds the contents of `C:\Folder` (according to the current [Filter settings](#)) and all `.jpg` files from `C:\Pictures` to the *Files* pane.

Load preset

```
"ReNamer.exe" /preset <preset> [<files>]
```

Launch ReNamer and load a [preset](#) specified by its name or by a full path to the preset file. You will then need to [preview](#) and [rename](#) the files yourself.

Paths to files and folders may optionally be appended to the command, and they will be added to the *Files* pane automatically.

You can use the *Create links* option in the [Presets menu](#) to automatically generate shortcut files for all available presets using this mode.

Example:

```
"ReNamer.exe" /preset "MyRules" "C:\Folder"
```

This launches a new ReNamer window, loads the preset named `MyRules`, and adds the contents of `C:\Folder` to the *Files* pane (according to the current [Filter settings](#)).

Rename automatically

```
"ReNamer.exe" /rename <preset> [<files>]
```

Launch ReNamer, load a [preset](#) specified by its name or by a full path to the preset file, add any specified files and folders, and then automatically proceed with [Preview](#) and [Rename](#) operations.

- If both operations complete successfully, the ReNamer window closes automatically.
- If any errors occur, the ReNamer window remains open and displays an appropriate error message.

You can use the *Create links* option in the [Presets menu](#) to automatically generate shortcut files for all available presets using this mode.

Example:

```
"ReNamer.exe" /rename "MyRules" "C:\Folder"
```

This launches a new ReNamer window, loads the preset named `MyRules`, adds the contents of `C:\Folder`, and executes the preview and rename operations. The window closes automatically on

success.

Silent rename

```
"ReNamer.exe" /silent /rename <preset> [<files>]
```

Perform an unattended, fully automated renaming operation. The behavior is similar to `/rename`, but the graphical user interface is never shown, and any warnings or errors are silently discarded.

A non-zero exit code is returned if any warnings or errors occurred during the operation. *Added in v6.6.0.5 Beta.*

Example:

```
"ReNamer.exe" /silent /rename "MyRules" "C:\Folder"
```

Enqueue files

```
"ReNamer.exe" /enqueue <files>
```

Add the specified files and folders to an already running instance of ReNamer. If no running instance is found, a new one is launched.

Example:

```
"ReNamer.exe" /enqueue "C:\Folder" "C:\Pictures\*.jpg"
```

If ReNamer is already running, this adds the contents of `C:\Folder` and all `.jpg` files from `C:\Pictures` to the existing instance. If ReNamer is not running, a new instance is launched and the files are added there.

Load from list

```
"ReNamer.exe" /list <list files>
```

Load files and folders into ReNamer from one or more *list files*. A list file is a plain text file containing one file path per line. Paths in the list file can be absolute or relative — relative paths are resolved relative to the list file itself, not to the ReNamer executable.

Example:

```
"ReNamer.exe" /list "List1.txt" "List2.txt"
```

This loads all file paths contained in `List1.txt` and `List2.txt` into the *Files* pane.

Uninstall

```
"ReNamer.exe" /uninstall
```

Remove all manually enabled associations with the program, such as the [presets association](#).

This option is intended for advanced users only.

Long paths

Introduction

The maximum length of paths and file/folder names is limited in every mainstream operating system and file system. The exact limits can vary significantly between systems.

For example, in Windows 10 and earlier versions the maximum path length is limited to **260 characters** (`MAX_PATH` constant). In contrast, Linux imposes a limit of **4096 bytes** for full paths and **255 bytes** for file names. Older systems imposed even stricter limits, like [DOS](#) with an **80-character** limit for full paths and only **12 characters** (8.3 format) for file names.

Fortunately, this issue is slowly but surely being addressed, in one way or another. The sections below also discuss ways of working around some of these limitations.

Long paths in Windows

Modern versions of Windows have an alternative file path specification mechanism that overcomes the **260 characters** limitation.

Consider a conventional file path:

```
C:\Very\Long\Path
```

To convert a path to the "long path" specification, prepend it with `\\?\`, as follows:

```
\\?\C:\Very\Long\Path
```

This effectively raises the maximum path length to **32,767 characters**, and also enables paths that end with a dot.

However, there are other potential limitations to be aware of:

- This only raises the limit imposed by the Windows operating system; limits imposed by individual file systems may still apply.
- Support for long paths also depends on the capabilities of individual applications.

Native long path support in Windows

On Windows 10 version 1607 and later, long paths can be enabled natively. This requires two conditions to be met:

1. The application's manifest must have the *Long Path Aware* feature enabled:

```
<ws2:longPathAware>true</ws2:longPathAware>
```

2. The system must have the `LongPathsEnabled` registry entry set to `1` (DWORD):

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem\LongPathsEnabled
```

A reboot, or a log off and log back on, may be required for the changes to take effect.

Once enabled, the maximum path length extends to approximately **32,767 characters**, without the need for the `\\?\` prefix.

Long paths in ReNamer

ReNamer normally operates with conventional file paths, but it can also handle the "long path" specification if one is provided.

When native long path support is enabled system-wide (described in the section above), ReNamer benefits from it automatically with no additional configuration required.

On systems without native long path support, the *Convert paths to long path specification* option in [General settings](#) offers an alternative. When enabled, all added paths are automatically prefixed with `\\?\`, raising the maximum path length enforced by Windows from 260 to 32,767 characters.

The standard *Add Files* and *Add Folders* dialogs cannot handle long paths, as this is a Windows limitation. However, you can use the *Add Paths* dialog from the [main menu](#), or the [Export and Import](#) menu options.

Truncating long paths

Excessively long paths can make files inaccessible to some applications, including Windows File Explorer itself.

A common workaround is to truncate long paths. This can be achieved using the [Pascal Script](#) and [Regular Expressions](#) rules.

References

- [Windows: Naming Files, Paths, and Namespaces](#)
- [Windows: Maximum Path Length Limitation](#)
- [Wikipedia: Comparison of file systems and limits](#)

Wildcard masks

Wildcard masks (also called filename masks) are a simple pattern-matching syntax that uses special placeholder characters to match filenames.

Masks can be used in several places in ReNamer, including the [Replace rule](#) and [Remove rule](#), [Filter settings](#), and the [Command line](#).

Pattern syntax

Masks are built from literal characters and the following special patterns:

Pattern	Description
<code>*</code> (asterisk)	Matches any number of any characters, including none.
<code>?</code> (question mark)	Matches any single character.
<code>[abc]</code>	Matches any one of the specified characters (<code>a</code> , <code>b</code> , or <code>c</code>).
<code>[a-z]</code>	Matches any one character in the specified range (<code>a</code> through <code>z</code>).

Mask lists

Multiple masks can be combined into a single *mask list* by separating them with semicolons (`;`). A file matches the list if it matches any one of the masks.

For example, the mask list `*.txt;*.doc` matches files with either a `.txt` or a `.doc` extension.

Examples

Mask	Description
<code>*.jpg</code>	Matches files with a <code>.jpg</code> extension.
<code>magic*</code>	Matches files whose name starts with <code>magic</code> .
<code>*magic*</code>	Matches files whose name contains <code>magic</code> anywhere.

Mask	Description
magic.	Matches files whose base name (excluding extension) ends with <code>magic</code> .
magic.jpg	Matches <code>.jpg</code> files whose name contains <code>magic</code> anywhere.
b?ll	Matches files containing a <code>b</code> , followed by any single character, followed by <code>ll</code> — for example, <code>ball</code> , <code>bell</code> , or <code>bill</code> .

Regular Expressions reference

Introduction

Regular Expressions (RegEx) is a syntax for specifying patterns of text to search and replace, which can be used for renaming files via the [Regular Expressions renaming rule](#).

Special metacharacters allow you to specify, for instance, that a particular string you are looking for occurs at the beginning or end of a line, or contains N recurrences of a certain character. Metacharacters, such as `$`, `.`, `^`, `{`, `[`, `(`, `|`, `)`, `*`, `+`, `?`, `\` are interpreted according to their individual meaning, instead of finding a literal match for them.

In this document, the subject text which is checked against a pattern for a possible match is shown in **bold**. Parts of the subject text may be color-coded to provide a clue as to why a certain part matches (**green color**), or does not match (**red color**).

Simple matches

When the search string does not contain any metacharacters, the RegEx processor works like "normal" search. It tries to find an exact copy of the search string. This is also known as a literal match.

If you want to find a literal match for a metacharacter, put a backslash `\` before it. The `\` character is called an *escape character*, because it lets the metacharacter escape from its special duty, and lets it act as a normal character. Its combination with a metacharacter is called *escape sequence*.

For example, metacharacter `^` matches the beginning of string, but pattern `\^` matches `^` character literally. Similarly, pattern `\\` matches `\` character literally.

Pattern	Matches	Remarks
<code>foobar</code>	foobar	This pattern does not contain any metacharacters, so all characters are matched literally.
<code>\^FooBar</code>	^FooBar	The <code>\^</code> escape sequence matches the <code>^</code> character literally.

Escape sequences

We already saw one use of escape sequence (above).

Specific escape sequences are interpreted as special conditions, as listed below.

Pattern	Remarks
<code>\xnn</code>	Character represented by the hex code <i>nn</i>
<code>\x{nnnn}</code>	Two bytes char with hex code <i>nnnn</i> (unicode)
<code>\t</code>	Tab (HT/TAB), same as <code>\x09</code> (Hex 09)
<code>\n</code>	New line (NL), same as <code>\x0a</code> (Hex 0a)
<code>\r</code>	Carriage return (CR), same as <code>\x0d</code> (Hex 0d)
<code>\f</code>	Form feed (FF), same as <code>\x0c</code> (Hex 0c)
<code>foo\x20bar</code>	Matches foo bar (note the space in the middle), but does not match foobar
<code>\tfoobar</code>	Matches foobar preceded by a tab (the tab is needed for the match)

Note that the tab, new line, carriage return, and form feed are all known as *white space* characters, because they are normally not visible on display, but the RegEx processor can distinguish between them.

Character classes

A character class is a list of characters surrounded by square brackets `[` and `]`, which will match any one (and only one) character in that class.

Note that:

- The characters are not separated with a comma or a space.
- If you repeat any character in the list, it is considered only once (duplicates are ignored).
- A hyphen character `-` is used to indicate a range of characters.

Pattern	Remarks
<code>[abcdef]</code>	Matches any one of a , b , c , d , e , or f .
<code>[c-m]</code>	Matches any one small alphabetical character from c through m .
<code>[G-J]</code>	Matches any one capital alphabetical character from G through J .

Pattern	Remarks
<code>[a-zA-Z]</code>	Matches any one alphabetical character (small or capital).
<code>[5-8]</code>	Matches any one numerical character from 5 through 8 .
<code>[\x00-\x1F]</code>	Matches any one character with a hexadecimal value in range from <code>0</code> through <code>1F</code>. This range includes all ASCII control characters, most of which are non-printable.

There are some special conditions:

- If you do not want any of the characters in the specified class, then place `^` at the very beginning of the list, which means "none of the characters listed in this class".
- If you want `[` or `]` itself to be a member of a class, put it at the start or end of the list, or use an escape sequence by putting `\` before it.

Pattern	Remarks
<code>[-az]</code>	Matches a, z, and -. Note the <code>-</code> is at the beginning of the pattern, the escape sequence is not needed.
<code>[a\\-z]</code>	Matches a, z, and -. Note the <code>-</code> is not at the beginning/end of the pattern, the escape sequence is needed.
<code>[^0-9]</code>	Matches any non-digit character.
<code>[]-a]</code>	Matches any character from] to a. Note the <code>]</code> is at the beginning of the pattern, the escape sequence is not needed.
<code>foob[aeiou]r</code>	Matches with foob a r and foob e r , but not foob b r , foob c r , etc.
<code>foob[^aeiou]r</code>	Matches with foob b r , foob c r etc. but not foob a r , foob e r , etc.

Predefined classes

Some of the character classes are used so often that RegEx has predefined escape sequences to represent them.

Pattern	Description
---------	-------------

<code>\w</code>	Alphanumeric character, including an underscore <code>_</code> character.
<code>\W</code>	Non-alphanumeric character.
<code>\d</code>	Numeric character.
<code>\D</code>	Non-numeric character.
<code>\s</code>	White space character, same as <code>[\t\n\r\f]</code> character class.
<code>\S</code>	Any character, excluding white space characters.
<code>.</code>	Any character.

Notice that the capitalized classes act as negatives, for example, `\W` has an inverse meaning of `\w`.

Word and text boundaries

A word boundary `\b` matches a position between a word character `\w` and a non-word character `\W`. For the purpose of a word boundary position, the start and end of text are treated like a non-word character `\W`. These markers are commonly used for matching patterns as whole words, while ignoring occurrences within words.

Pattern	Description
<code>\b</code>	Word boundary.
<code>\B</code>	Not word boundary.
<code>\A</code> or <code>^</code>	Start of text
<code>\Z</code> or <code>\$</code>	End of text

For example, `\bhis\b` will match a whole word **his**, but will not match **this**, **history** or **whistle**.

Iterators

Iterators (quantifiers) are meta-characters that specify how many times the *preceding* expression has to repeat. For example, finding a numeric sequence exactly 3 to 5 digits long.

Iterators can be *greedy* or *non-greedy*. Greedy means the expression grabs as *much* matching text as possible. In contrast, the non-greedy expression tries to match as *little* as possible.

All iterators are greedy by default. Adding `?` (question mark) at the end of an iterator makes it non-greedy.

For example:

- when `b+` (a greedy expression) is applied to string **abbbbc**, it matches **bbb** (as many as possible),
- but when `b+?` (a non-greedy expression) is applied to **abbbbc**, it matches **b** (as few as possible).

Iterator	Description	Greedy?	Alternative
<code>*</code>	zero or more	Yes	<code>{0,}</code>
<code>+</code>	one or more	Yes	<code>{1,}</code>
<code>?</code>	zero or one	Yes	<code>{0,1}</code>
<code>{n}</code>	exactly n times	Yes	
<code>{n,}</code>	at least n times	Yes	
<code>{n,m}</code>	at least n but not more than m times	Yes	
<code>*?</code>	zero or more	No	<code>{0,}?</code>
<code>+?</code>	one or more	No	<code>{1,}?</code>
<code>??</code>	zero or one	No	<code>{0,1}?</code>
<code>{n}?</code>	exactly n times	No	
<code>{n,}?</code>	at least n times	No	
<code>{n,m}?</code>	at least n but not more than m times	No	

Let's see some examples:

Pattern	Remarks
<code>foob.*r</code>	matches foob ar , foob xyz123 r and foobr
<code>foob.+r</code>	matches foob ar , foob xyz123 r but not foobr
<code>foob.?r</code>	matches foob ar , foob br and foobr but not foob xyz123 r
<code>fooba{2}r</code>	matches fooba ar
<code>fooba{2,}r</code>	matches fooba ar , fooba aar , fooba aaaar but not fooba r
<code>fooba{2,3}r</code>	matches fooba ar , fooba aar but not fooba aaaar or fooba r

Alternatives

A RegEx expression can have multiple alternative characters or subexpressions. The metacharacter `|` (vertical pipe) is used to separate the alternatives. For example, `fee|fie|foe` will match **fee**, **fie**, and **foe** in the subject text.

It is a common practice to group alternatives into parentheses, to make it easier to identify and distinguish them. For example, `fee|fie|foe` can be written as `(fee|fie|foe)` or as `f(e|i|o)e`.

Alternatives are tested for a match in the order in which they appear, from left to right, iterating until the full expression match is found or all alternatives have failed to match. For example, when matching `foo|foot` against **barefoot**, the first alternative **foo** will match first.

Pattern	Remarks
<code>fee fie foe</code>	Matches fee , fie , and foe .
<code>f(e i o)e</code>	Matches fee , fie , and foe .
<code>foo(bar foo)</code>	Matches foobar or foofoo .

Also remember that alternatives cannot be used inside a character class (square brackets), because `|` is interpreted literally within `[]`. That means that `[fee|fie|foe]` is same as `[feio|]`, where repeating characters are treated as duplicates, and ignored.

Subexpressions

Parts of a pattern can be enclosed in round brackets `()`, just like using brackets in a mathematics formula `a+(b+c)`. Each part that is enclosed in brackets is called a *subexpression*. Subexpressions can provide clarity in complex expressions, and can be referenced in both the expression itself and in the replacement pattern.

Let's see some examples:

Pattern	Remarks
<code>(fee fie foe)bar</code>	Subexpression <code>fee fie foe</code> is clearly separated from the remaining expression.
<code>(foobar){2,3}</code>	Matches foobar if repeated 2 or 3 times, i.e. foobarfoobar and foobarfoobarfoobar .
<code>foobar{2,3}</code>	Matches fooba followed by the character r repeated 2 or 3 times, i.e. foobarr and foobarrrr .
<code>foob([0-9] a+)r</code>	Matches foob0r , foob1r , foobar , foobaar , foobaaaar , etc.

Backreferences

Backreferences allow you to reference individual subexpressions, enabling complex repeating patterns.

Each subexpression is identified by its index (order of appearance) in the full expression and can be referenced using a backslash `\` followed by the subexpression index, e.g. `\1`, `\2`, `\3`, and so on.

Let's see some examples:

Pattern	Remarks
<code>(.)\1+</code>	Matches any character that is repeated at least twice, e.g. aaaa and cc .
<code>(.+)\1+</code>	Matches any sequence of characters that is repeated at least twice, e.g. aaaa , cc , abababab , 123123 .
<code>(cat dog) \1</code>	Matches cat cat or dog dog .

Named groups

Named groups work like numbered subexpressions but assign a descriptive name to each group, making complex patterns easier to read and maintain. A named group is referenced by name rather than by its position in the expression.

To define a named group, use `(?P<name>pattern)`. To back-reference a named group within the same expression, use `(?P=name)`. To reference a named group in the replacement pattern, use `${name}`.

Pattern	Remarks
<code>(?P<year>\d{4}) - (?P<month>\d{2}) - (?P<day>\d{2})</code>	Defines three named groups: year , month , and day . Each group captures the corresponding part of a date string.
<code>(?P<word>\w+) (?P=word)</code>	Matches a repeated word, e.g. the the . The back-reference <code>(?P=word)</code> requires the same text that was captured by the word group.

Named groups can also be referenced by name in the replacement pattern, instead of by number:

Find	Replace	Description
<code>(?P<year>\d{4}) - (?P<month>\d{2}) - (?P<day>\d{2})</code>	<code>\${day}-\${month}-\${year}</code>	Reformat a date from <code>yyyy-mm-dd</code> to <code>dd-mm-yyyy</code> . For example, "2024-10-25" becomes "25-10-2024".

Named groups are particularly useful in long expressions with many groups, where keeping track of group numbers becomes error-prone.

Substitutions

Subexpressions can also be referenced in the replacement pattern, allowing you to assemble the output using individual subexpression matches.

Each subexpression is identified by its index (order of appearance) in the full expression and can be referenced using a dollar sign `$` followed by the subexpression index, e.g. `$1`, `$2`, `$3`, and so on. The full expression match can be referenced using `$0`.

Let's see some examples of replacement with substitutions:

Find	Replace	Description
<code>(\w+) (\w+)</code>	<code>\$2, \$1</code>	Switch two words around and put a comma between them. For example, "John Smith" becomes "Smith, John".
<code>\b(\d{2})-(\d{2})-(\d{4})\b</code>	<code>\$3-\$2-\$1</code>	Find dates in <code>dd-mm-yyyy</code> format and reverse them into <code>yyyy-mm-dd</code> format. For example, "25-10-2007" becomes "2007-10-25".

Note that the last example is not a robust approach for handling dates, because `\d` matches any digit in 0-9 range. This means that sequences like "00-00-0000" and "99-99-9999" will also match this pattern, but do not represent a valid date.

Lookaround assertions

Lookaround assertions (lookahead and lookbehind) let you enforce conditions on text immediately before or after the main match, without including that surrounding text in the matching result.

Assertions are zero-width as they consume no characters, and come in four forms: positive/negative lookahead (checks ahead) and positive/negative lookbehind (checks behind).

Type	Syntax	Description
Positive lookahead	<code>foo(?=bar)</code>	Matches foo only if immediately followed by bar ("bar" is not consumed).
Negative lookahead	<code>foo(?!bar)</code>	Matches foo only if not immediately followed by bar .
Positive lookbehind	<code>(?<=bar)foo</code>	Matches foo only if immediately preceded by bar ("bar" is not consumed).

Type	Syntax	Description
Negative lookbehind	<code>(?<!bar)foo</code>	Matches foo only if not immediately preceded by bar .

Let's see some examples:

Pattern	Subject	Remarks
<code>\d+(?= euros)</code>	100 euros	Matches 100 (digits only if followed by space and "euros").
<code>\d+(?! euros)</code>	100 dollars	Matches 100 (digits only if not followed by space and "euros").
<code>(?<=EUR)\d+</code>	EUR 50	Matches 50 (digits only if preceded by "EUR" and space).
<code>(?<!EUR)\d+</code>	USD 30	Matches 30 (digits only if not preceded by "EUR" and space).

Non-capturing groups

Non-capturing groups allow you to apply quantifiers, alternatives, or other operations to a subpattern without creating a numbered capture. This avoids unnecessary captures, keeps backreference numbering unaffected, and is especially useful when you need grouping only for structure or performance, without intending to reference the group later.

The syntax is `(?:subpattern)` to group the subpattern without creating a capture.

Let's see some examples, with and without non-capturing groups:

Pattern	Subject	Remarks
<code>(Mr Mrs) (\w+)</code>	Mr Smith	Matches the full salutation and name. Captures title (1st backreference: Mr) and name (2nd: Smith).
<code>(?:Mr Mrs) (\w+)</code>	Mr Smith	Matches the full salutation and name. Captures only name (1st backreference: Smith). Title is grouped but not captured.
<code>(https?):\/\/([^\s]+)</code>	https://example.com	Matches the full URL. Captures protocol (1st backreference: https) and domain (2nd: example.com).

Pattern	Subject	Remarks
<code>(?:https?://)([^\s]+)</code>	<code>https://example.com</code>	Matches the full URL. Captures only domain (1st backreference: example.com). Protocol is grouped but not captured.

Text case adjustments

Subexpressions can also be used to adjust the text case (upper case, lower case) of certain patterns, which otherwise cannot be easily achieved with generic text case manipulation rules.

The following flags can be combined with subexpression references in the replace pattern:

Flag	Description
<code>\L</code>	Convert all characters to lowercase.
<code>\l</code>	Convert only the first character to lowercase.
<code>\U</code>	Convert all characters to uppercase.
<code>\u</code>	Convert only the first character to uppercase.

For example, we can do the following manipulations:

Input	Find	Replace	Result
hello WORLD	<code>(.+) (.+)</code>	<code>\$1 \$2</code>	hello WORLD
hello WORLD	<code>(.+) (.+)</code>	<code>\U\$1 \$2</code>	HELLO WORLD
hello WORLD	<code>(.+) (.+)</code>	<code>\$1 \L\$2</code>	hello world
hello WORLD	<code>(.+) (.+)</code>	<code>\u\$1 \L\$2</code>	Hello world

Inline modifiers

Inline modifiers let you change matching behaviour for part of a pattern, without affecting the rest. The most practical use for file renaming is enabling case-insensitive matching selectively — for example, when only part of a pattern needs to match regardless of capitalisation.

The syntax is `(?modifier)` to enable and `(?-modifier)` to disable. A modifier can also be scoped to a specific subpattern using `(?modifier:pattern)`, so that it applies only within that group and leaves the rest of the expression unchanged.

The most commonly useful modifiers are:

Modifier	Description
<code>i</code>	Case-insensitive matching.
<code>m</code>	Multi-line mode: <code>^</code> and <code>\$</code> match the start and end of each line, not just the whole string.
<code>s</code>	Single-line mode: <code>.</code> matches any character including line-break characters.

An inline modifier affects everything that follows it in the current group or expression:

Pattern	Subject	Remarks
<code>(?i)report</code>	Report _2024.pdf	Matches report in any capitalisation.
<code>(?i:copy) of (.+)</code>	Copy of Report .pdf	The <code>(?i:...)</code> form scopes case-insensitive matching to copy only; the rest of the pattern remains case-sensitive.

See also

- [Regular-Expressions.info](https://www.regular-expressions.info) – Excellent site devoted to regular expressions. It is nicely structured, with many easy to understand examples.
- [TRegExpr](https://github.com/stevesoutherland/TRegExpr) – Regular expressions library for Delphi and Free Pascal. For syntax and API documentation see regex.sorokin.engineer.
- [FPC RegEx packages](https://github.com/stevesoutherland/FPC-RegEx) – Regular expressions libraries included in Free Pascal.

Settings and Menus

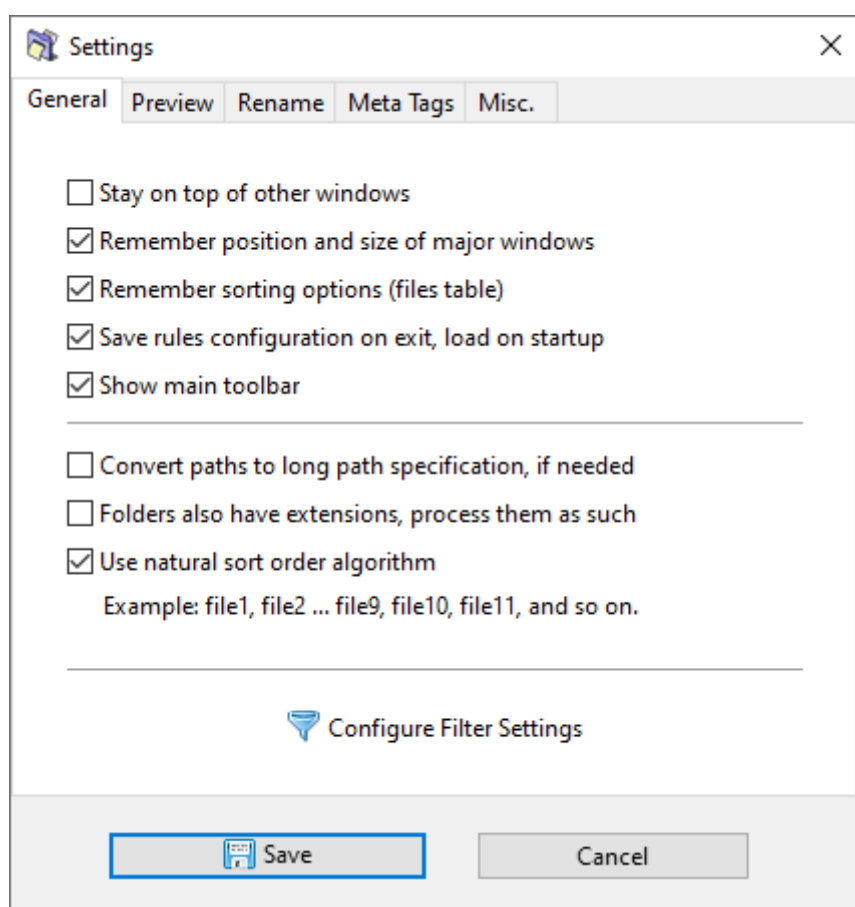
Description of program settings and menus.

Main settings

This page describes the settings that control ReNamer's behaviour in various ways.

General settings

These settings control the general behaviour of the application.



Option	Description
<i>Stay on top of other windows</i>	Keep the ReNamer window on top of other applications. This is useful when adding files via drag-and-drop from Windows Explorer, as the ReNamer window will not disappear behind the Explorer window while you work.
<i>Remember position and size of major windows</i>	Retain the size and position of major windows between sessions, so they do not need to be repositioned each time ReNamer starts.
<i>Remember sorting options (files table)</i>	Retain the file sorting order between sessions.

Option	Description
<i>Save rules configuration on exit, load on start up</i>	Remember the currently loaded rules or preset on exit and restore it automatically on the next start. As of <i>v7.2.0.6 Beta</i>, the default preset is remembered by preset name rather than as an anonymous default preset, unless the preset was in a modified state at the time of saving.
<i>Show main toolbar</i>	Show the main toolbar. When disabled, the toolbar is hidden to gain extra space for the rules and files tables. All actions normally available in the toolbar remain accessible via the main menu or keyboard shortcuts.
<i>Convert paths to long path specification, if needed</i>	Convert all added paths to long path specification when the system does not support long paths natively. This prefixes paths with <code>\\?\</code>, raising the maximum path length enforced by Windows from 260 characters to 32,767 characters. <i>Added in v7.8.0.2 Beta.</i>
<i>Folders also have extensions, process them as such</i>	A file extension is defined as whatever follows the last dot in the filename. This setting controls whether the same concept applies to folders, i.e. whether folders can have an extension. Prior to <i>v5.74.5 Beta</i>, a dot in a folder name was also treated as identifying an extension, so the <i>Skip Extension</i> option in rules applied to folders as well. In newer versions this is no longer the case, but this setting restores the old behaviour.
<i>Use natural sort order algorithm</i>	When enabled, filenames are sorted in <i>Natural</i> order; otherwise they are sorted in <i>Lexicographical</i> order. See the sorting algorithms section below for details.

Sorting algorithms

Two sorting algorithms are available for ordering files in the files table.

Natural

Numbers embedded in filenames are sorted by their numeric value rather than character by character, which produces a more intuitive ordering:

```
Name1.ext
Name2.ext
Name3.ext
Name10.ext
Name20.ext
```

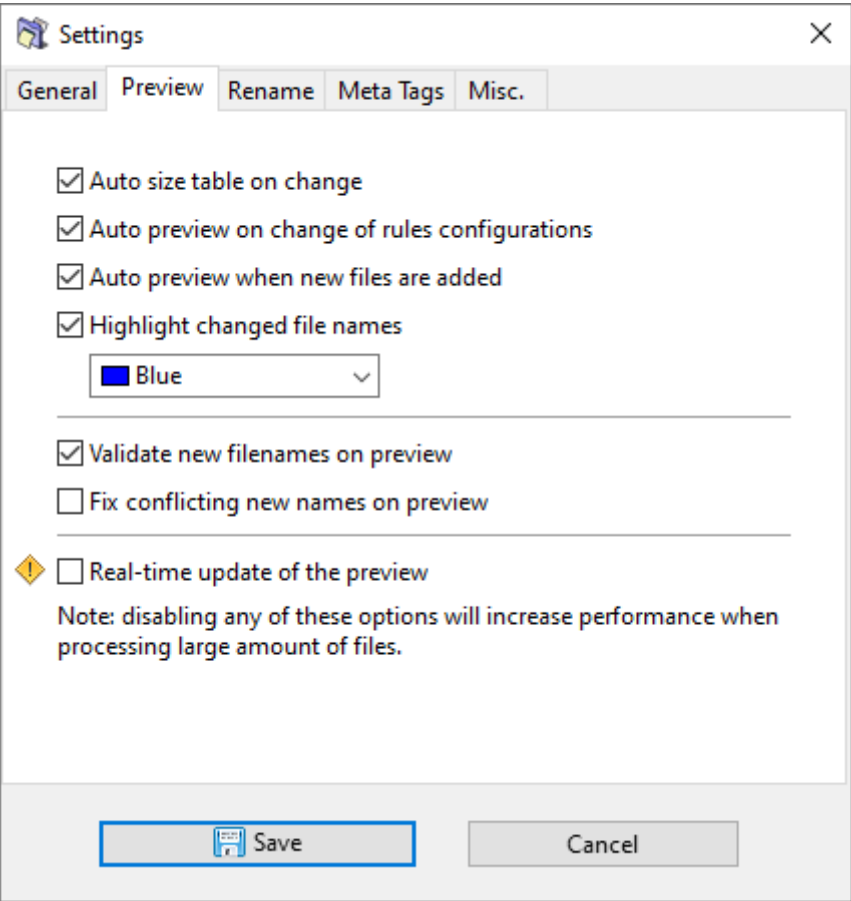
Lexicographical

Filenames are sorted as in a dictionary, strictly by character position from left to right. This means numeric portions are sorted by digit, not by value:

```
Name1.ext
Name10.ext
Name2.ext
Name20.ext
Name3.ext
```

Preview settings

These settings apply to the preview operation.

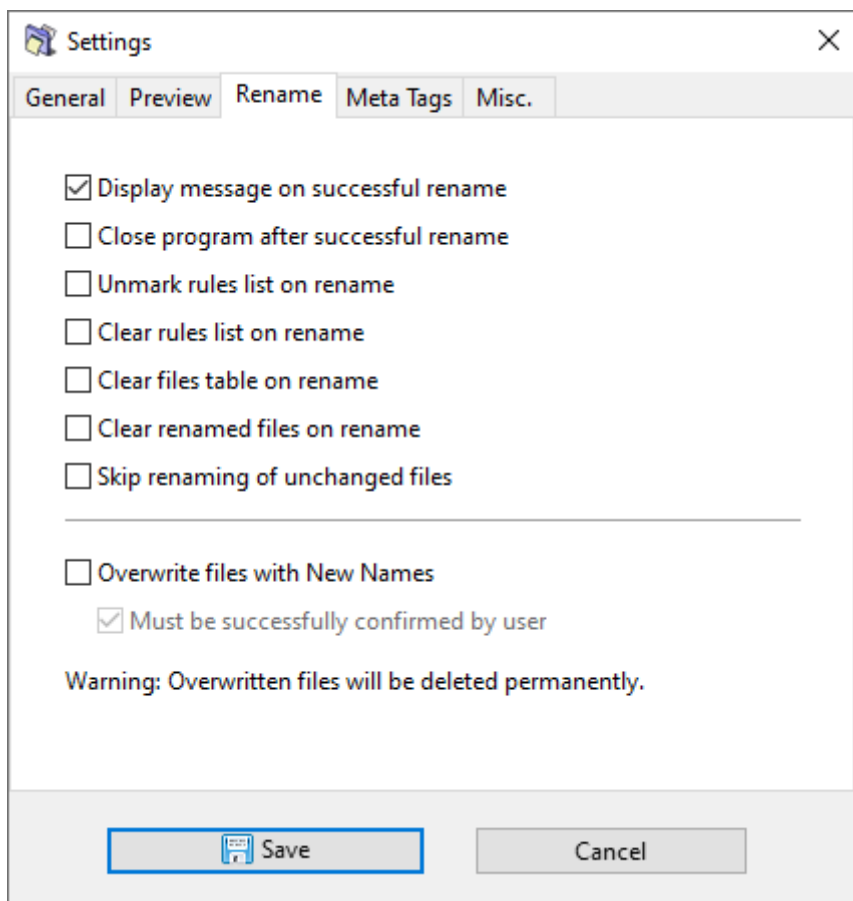


Option	Description
<i>Autosize table on change</i>	Automatically resize each column to fit the longest name it contains.
<i>Auto preview on change of rules configurations</i>	Refresh the preview automatically whenever a rule is added, removed, or edited.

Option	Description
<i>Auto preview when new files are added</i>	Refresh the preview automatically whenever a file is added to the files table.
<i>Highlight changed file names</i>	Highlight new names in a specified colour when they differ from the original names, making it easy to see which files are affected by the active rules.
<i>Validate new filenames on preview</i>	Raise a warning if any new filenames are not valid. See Validation for details on what is checked.
<i>Fix conflicting new names on preview</i>	Resolve new name conflicts automatically by appending a numeric suffix, producing names such as <code>Name (2)</code> , <code>Name (3)</code> , and so on. This can also be triggered on demand via the Options menu .
<i>Real-time update of the preview</i>	When disabled (the default), clicking <i>Preview</i> shows no progress until all files have been processed. When enabled, files are updated one by one to provide live visual feedback. Warning: ReNamer continues to accept mouse and keyboard input while processing in order to keep the interface responsive. Any interaction with the UI during processing may lead to unpredictable behaviour. Use this option with extreme caution, and only when heavy per-file processing is involved, such as hash or checksum calculations or other content analysis.

Rename settings

These settings apply to the renaming process.



These options let you automate certain actions that follow a rename operation, or keep them manual if you prefer full control each time.

Option	Description
<i>Display message on successful rename</i>	When disabled, no confirmation message is shown after a successful rename. This setting does not affect error messages.
<i>Close program after successful rename</i>	Close ReNamer automatically after a successful rename. Useful if you rename and then immediately move on to other tasks. If any errors occur, ReNamer stays open.
<i>Unmark rules list on rename</i>	Unmark all rules after a rename operation. This acts as a safety check, requiring you to explicitly re-enable rules before the next rename.
<i>Clear rules list on rename</i>	Clear the <i>Rules</i> pane automatically after renaming. Useful when you use a different set of rules for each renaming session.
<i>Clear files table on rename</i>	Clear all files from the table after renaming, including any files that were not renamed in the last operation. Disable this option if you plan to apply additional rules to the same set of files across multiple separate rename operations. Note: Enabling this option makes <i>Undo</i> unavailable.

Option	Description
<i>Clear renamed files on rename</i>	Remove only the files that were successfully renamed, leaving any unaffected files in the table so you can apply a new set of rules to them. Note that files are considered renamed even if the active rules did not produce a name change (for example, when inserting text at a position beyond the length of a short filename). Note: Enabling this option makes <i>Undo</i> unavailable.
<i>Skip renaming of unchanged files</i>	By default, all marked files are renamed even when the new name matches the original, to maintain consistent behaviour with other options that act on renamed files. Enable this option to skip files whose name would not change.
<i>Overwrite files with new names</i>	When a naming conflict arises, allows the renamed file to silently overwrite the existing file with the same name. The conflict may involve a file that exists in the same folder but is not loaded in ReNamer. When multiple files rename to the same target name, they overwrite each other in sequence, and only the last one survives. Warning: Overwritten files are permanently deleted and cannot be recovered.
<i>Must be successfully confirmed by user</i>	Applies only when <i>Overwrite files with new names</i> is enabled. When enabled, a confirmation dialog appears for each file that would be overwritten, and the overwrite proceeds only if confirmed. When disabled, conflicting files are overwritten silently without prompting. Warning: Overwritten files are permanently deleted and cannot be recovered.

Meta tags settings

These settings apply to meta tag extraction.

Settings

General

Preview

Rename

Meta Tags

Misc.

☒ MetaTag support

Note: Disabling MetaTag support will increase performance when processing large amount of files.

Date Format:

yyyy-mm-dd hh-nn-ss

Preview:

2026-01-24 22-55-34

Note: formats will be saved only if validated successfully.

Help

Save

Cancel

Option	Description
<i>Meta tag support</i>	Enable the processing of meta tags when they are encountered. Meta tag extraction can require significant system resources. Disabling this option causes meta tags to be left in place unprocessed, which avoids that overhead entirely.
<i>Date format</i>	Specify the format used when displaying date and time meta tags. See Date and Time format for available formatting options.
<i>Preview</i>	Test the specified date and time format using the current time.

Miscellaneous settings

This tab contains settings that do not belong to any other category.


Settings
✕

General
Preview
Rename
Meta Tags
Misc.

☐ Register preset extension (*.rnp)
☐ Add to folders context menu
☒ Add to "Send To" context menu

Change text of "Drag your files here":

Change text of "Click here to add a rule":


Save

Cancel

Option	Description
<i>Register preset extension (*.rnp)</i>	Associate the <code>.rnp</code> file extension with ReNamer.
<i>Add to folders context menu</i>	Add a ReNamer entry to the context menu for folders in Windows Explorer.
<i>Add to "Send To" context menu</i>	Add a ReNamer entry to the <i>Send To</i> context menu in Windows Explorer.
<i>Change text of "Drag your files here"</i>	Customise the placeholder label displayed in the files table when it is empty.
<i>Change text of "Click here to add a rule"</i>	Customise the placeholder label displayed in the rules table when it is empty.

Main menu

This article describes the options available in the main menu.

File menu

Option	Shortcut	Description
<i>New Project</i>	Ctrl+N	Create a new project, clearing all rules and files.
<i>Undo Renaming</i>	Shift+Ctrl+Z	Reverse the effect of the last renaming operation, if possible.
<i>Paste Files</i>	Shift+Ctrl+V	Paste a selection of files or folders from the clipboard into the <i>Files</i> pane.
<i>Add Files</i>	F3	Open a window for selecting files from any folder and adding them to the <i>Files</i> pane.
<i>Add Folders</i>	F4	Open a window for adding whole folders, according to the configured filter settings .
<i>Preview</i>	F5	Generate new names by applying the current renaming rules. Not required if the auto preview option is enabled.
<i>Rename</i>	F6	Rename all files to the names shown in the <i>New Name (New Path)</i> column in the <i>Files</i> pane.
<i>Exit</i>	Alt+F4	Close the application.

Settings menu

Option	Shortcut	Description
<i>All Settings</i>	F8	Open the Settings dialog.
<i>General</i>		Open the General tab of the <i>Settings</i> dialog.

Option	Shortcut	Description
<i>Preview</i>		Open the Preview tab of the <i>Settings</i> dialog.
<i>Rename</i>		Open the Rename tab of the <i>Settings</i> dialog.
<i>Meta Tags</i>		Open the Meta Tags tab of the <i>Settings</i> dialog.
<i>Miscellaneous</i>		Open the Miscellaneous tab of the <i>Settings</i> dialog.
<i>Filters</i>	Ctrl+F	Open the Filter settings dialog, which controls the default behavior when adding folders and allows specifying filtering based on file mask.

Presets menu

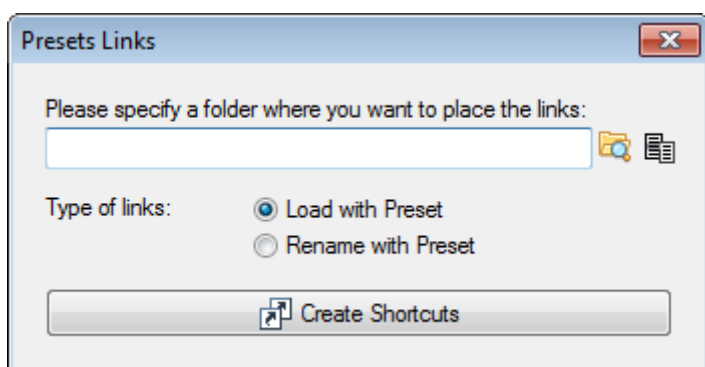
Presets store a named set of rules that can be quickly loaded and reused. For a full overview of working with presets, see [Using Presets](#).

Option	Shortcut	Description
<i>Load</i>		Open a submenu listing all available presets. Click a preset to load its rules into the <i>Rules</i> pane, replacing the current set (any unsaved rules will be lost). If the preset was saved with filter settings , those will also be applied. To append rules rather than replace, hold Shift while loading a preset, or use the <i>Append Preset</i> option in the Preset Manager .
<i>Save</i>	Shift+Ctrl+S	Save the current rules to the loaded preset. If no preset is currently loaded, prompts for a name.
<i>Save As</i>	Ctrl+S	Save the current rules as a new preset, always prompting for a name.
<i>Manage</i>	Ctrl+M	Open the Preset Manager dialog.
<i>Browse...</i>		Open the folder where presets are stored in Windows Explorer.
<i>Import...</i>		Select preset files from any location to import (copy) them into ReNamer's presets folder.

Option	Shortcut	Description
<i>Create Links</i>		Create shortcut files (droplets) for each available preset. See the Create Links section below.
<i>Rescan</i>		Rescan the presets folder and repopulate the <i>Load</i> submenu. Useful after manually adding or removing preset files without restarting ReNamer.

Create Links

The *Create Links* option opens a dialog for generating shortcut files, known as **droplets**, for each available preset.



Shortcuts will be created for every available preset and placed in the specified folder. Two modes are available:

- *Load with Preset* — open a new ReNamer window with the preset loaded into the *Rules* pane and the files sent to the shortcut loaded into the *Files* pane. You can then review or adjust the rules before renaming. Any filter settings saved with the preset will be applied to the incoming files.
- *Rename with Preset* — load the preset and automatically rename all files sent to the shortcut, without opening the main window.

For more information, see the [Command line](#) article.

Warning: The *Rename with Preset* mode renames files immediately, without any confirmation prompt.

Help menu

Option	Shortcut	Description
<i>Help (Online)</i>	F1	Open the online documentation.

Option	Shortcut	Description
<i>User Manual</i>		Open the User Manual file distributed with the application.
<i>History</i>		Open the <code>History.txt</code> file distributed with the application, which lists changes across all versions.
<i>About</i>	<code>Alt+F1</code>	Show general information about ReNamer and contact details. Right-clicking the header of the dialog copies the version information to the clipboard.

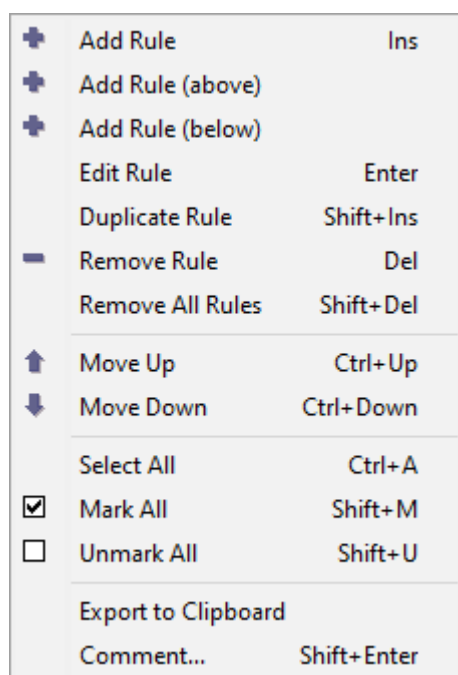
Rules menu

The toolbar above the *Rules* table offers some quick actions:



- Add a new rule.
- Remove selected rules.
- Move up selected rules.
- Move down selected rules.
- Open the [Presets Manager](#).

Right-click on the *Rules* table to see the context menu that offers extra options for manage rules.

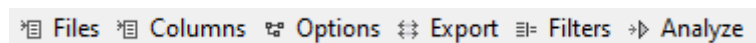


Action	Shortcut	Description
Add Rule	Ins	Add a new rule. The rules dialog will appear.
Add Rule (above)		Add a new rule above the selected rule. The rules dialog will appear.
Add Rule (below)		Add a new rule below the selected rule. The rules dialog will appear.
Edit Rule	Enter	Edit the selected rule. The rules dialog will appear.

Action	Shortcut	Description
Duplicate Rule	Shift+Ins	Duplicate the selected rule. Duplicated rule will appear just below the original rule.
Remove Rule	Del	Remove selected rules.
Remove All Rules	Shift+Del	Remove all rules.
Move Up	Ctrl+Up	Move selected rules upward in the stack.
Move Down	Ctrl+Down	Move selected rules downward in the stack.
Select All	Ctrl+A	Select all rules.
Mark All	Shift+M	Mark all rules. All rules will become active.
Unmark All	Shift+U	Unmark all rules. All rules will become inactive. This is useful if you have added too many rules and getting strange results. First, unmark all rules, so that none of them are active. Then, mark one rule at a time and see its incremental effect on the files.
Export to Clipboard		Export the description of all rules to Clipboard. This can be useful for demonstrating and sharing your rules setup.
Comment...	Shift+Enter	Annotate the selected rule with a custom description, which will be visible in the <i>Comment</i> column. This can be useful for describing complex rules, such as <i>Pascal Script</i> and <i>Regular Expressions</i>.

Files and Tools

The toolbar above the *Files* table provides quick access to file management options and various tools.

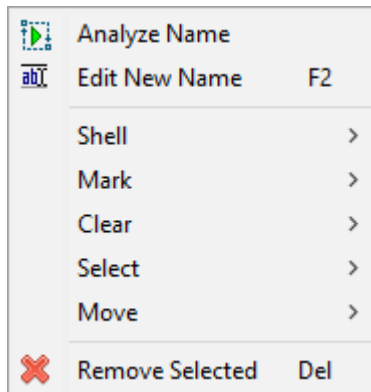


Button	Opens	Description
<i>Files</i>	Files context menu	Add, remove, sort, mark, and otherwise manage items in the <i>Files</i> table.
<i>Columns</i>	Files header menu	Control which columns are visible in the <i>Files</i> table.
<i>Options</i>	Options menu	Various ad-hoc actions: validate new names, fix name conflicts, apply rules to clipboard content, and more.
<i>Export</i>	Export menu	Export and import the contents of the <i>Files</i> table in various formats.
<i>Filters</i>	Filter settings	Control which files and folders are included when using the <i>Add folders</i> function.
<i>Analyze</i>	Analyze tool	Apply the current renaming rules to arbitrary text, without loading any files.

Files context menu

Right-click on the *Files* table content area to see the context menu that offers numerous options for manage files.

Menu



Option	Shortcut	Description
Analyze Name		Open the Analyze tool , and load the selected filenames for analysis.
Edit New Name	F2	Start manual editing of the selected filename.
Shell		Options for shell operations, including opening files, copying into clipboard, and deleting to the recycle bin.
Mark		Options for marking specific items, including marking only changed files, and marking by mask.
Clear		Options for removing specific items from the table selectively, including by status.
Select		Options for selecting specific items, including selecting by name length, extension, and mask.
Move		Options for moving specific items up or down.
Remove Selected	Del	Remove selected items from the table. This command does not delete the items from the disk.

Shell menu

Open File	Enter
Open with Notepad	Shift+Enter
Open containing folder	Ctrl+Enter
File Properties	Alt+Enter
Cut Files to Clipboard	Shift+Ctrl+X
Copy Files to Clipboard	Shift+Ctrl+C
Delete Files to Recycle Bin	

Option	Shortcut	Description
Open File	Enter	Open the selected file using its default associated application.
Open with Notepad	Shift+Enter	Open the selected file with Notepad to see the raw file content. Avoid using this option with binary files as the content will not make any sense, and you may risk accidentally corrupting the file.
Open containing folder	Ctrl+Enter	Launch File Explorer and open the folder where the selected file is located.
File Properties	Alt+Enter	Display the File Properties dialog offered by the operating system.
Cut Files to Clipboard	Ctrl+Shift+X	Place the source file paths to clipboard as a "cut" operation. If you paste these files in File Explorer, all files will be <i>moved</i> to one folder, no matter where they were initially located.
Copy Files to Clipboard	Ctrl+Shift+C	Place the source file paths to clipboard as a "copy" operation. If you paste these files in File Explorer, all files will be <i>copied</i> to one folder, no matter where they were initially located.
Delete Files to Recycle Bin		Delete the source files to the Recycle Bin. The files will also be removed from the table.

Mark menu

☒	Mark	Shift+M
☐	Unmark	Shift+U
☒	Invert Marking	Ins
Mark Only Changed (Inc. Case)		
Mark Only Changed (Exc. Case)		
Mark Only Selected		
Mark by Mask		

Option	Shortcut	Description
Mark	Shift+M	Mark all selected files. If some files are already marked, they remain marked.
Unmark	Shift+U	Unmark all selected files. If some files are already unmarked, they remain unmarked.
Invert Marking	Ins	Invert the marked state of selected files. Marked files become unmarked, and vice versa.
Mark Only Changed (Inc. Case)		Mark only those files whose new name differs from the original name, <i>including</i> differences in lower/upper case.
Mark Only Changed (Exc. Case)		Mark only those files whose new name differs from the original name, <i>excluding</i> differences in lower/upper case.
Mark Only Selected		Mark only the currently selected files, unmark all others.
Mark by Mask		Mark only those files that match a specific file mask. Brings up a dialog for specifying the file mask. You can specify multiple masks by separating them with semicolons. For example: <code>*.jpg;*.png</code> and <code>*Birthday*;*Holiday*</code>.

Clear menu

Clear All	Ctrl+Del
Clear Renamed	
Clear Failed	
Clear Valid	
Clear Invalid	
Clear Marked	
Clear Not Marked	
Clear Not Changed	Ctrl+D

These options offer you several criteria for clearing (removing) items from the table:

- Clearing all items (shortcut `Ctrl+Del`).
- Clear items with a specific status: Renamed, Failed, Valid, and Invalid.
- Clear marked or not marked items.
- Clear unchanged items (shortcut `Ctrl+D`), those whose new name is exactly the same as the original name and those whose new name is empty, as happens after successful renaming.

Select menu

Select All	Ctrl+A
Invert Selection	Ctrl+I
Select by Name Length	Ctrl+L
Select by Extension	Ctrl+E
Select by Mask	Ctrl+M

Option	Shortcut	Description
Select All	<code>Ctrl+A</code>	Select all items in the table.
Invert Selection	<code>Ctrl+I</code>	Invert the selection. Already selected items become unselected, and vice versa.

Option	Shortcut	Description
Select by Name Length	Ctrl+L	Select only those files whose filename exceeds a certain length (number of characters). Brings up a dialog for specifying the length. The length refers to the whole filename including the file extension and the dot. This is particularly useful for identifying filenames that may not satisfy the maximum length requirements for certain applications and file systems, like ISO 9660 used for CD/DVD.
Select by Extension	Ctrl+E	Select only those files which match a specific file extension. Brings up a dialog for specifying the file extension, without the leading dot. You can specify multiple file extensions by separating them with semicolons. For example: <code>jpg;png</code>.
Select by Mask	Ctrl+M	Select only those files that match a specific file mask. Brings up a dialog for specifying the file mask. You can specify multiple masks by separating them with semicolons. For example: <code>*.jpg;*.png</code> and <code>*Birthday*;*Holiday*</code>.

Move menu

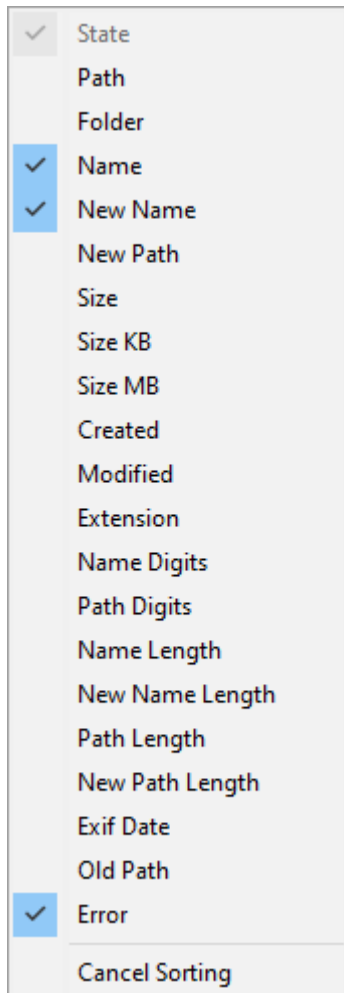
Up	Ctrl+Up
Down	Ctrl+Down

These options simply move the currently selected items up (`Ctrl+Up`) or down (`Ctrl+Down`).

Alternatively, you can move the selected items by dragging them with a mouse click.

Files header menu

Right-click on the *Files* table header (columns) to see the context menu that allows you to enable/disable available columns.



Click on any of the listed items to toggle them on or off. Once the desired columns become visible, you can rearrange them by dragging them around with the mouse.

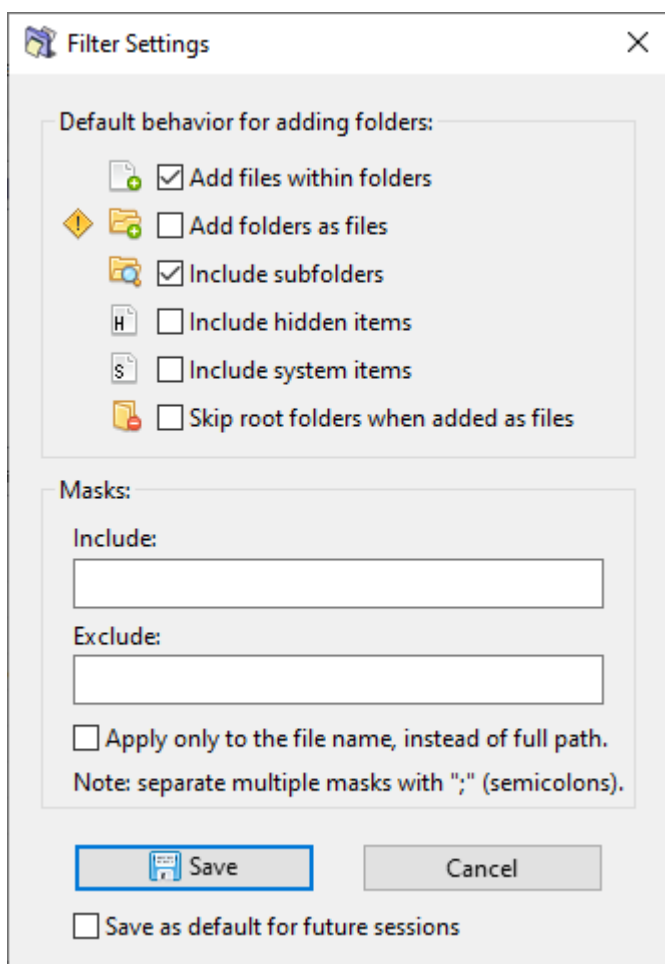
The last option in the menu allows you to cancel the sorting, if you have previously clicked on of the columns to enable sorting.

Filter settings

Filter settings give you a finer control over which files and folders get added to ReNamer.

Access these settings via the [Main settings](#), the [Tools menu](#) above the Files table, or by clicking on the filter icons in the status bar. You can use a range of methods for [adding new files and folders](#) and the selected filter settings will be applied consistently in all cases.

Note that the changes to filter settings are applied only to the newly added files/folders and do not affect items that were already added to the Files table.



Default behaviour for adding folders:

- **Add files within folders** – Scan the content of folders for contained files and add them to the table for renaming.
- **Add folders as files** – Add folders to the table for renaming, as if they are ordinary files. This essentially allows you to rename folders. Care must be taken when renaming folders and their contents at the same time.

- **Include subfolders** – Scan all folders recursively. If not selected, subfolders will be ignored.
- **Include hidden items** – Include or exclude items marked as "hidden" in the file system. This may include system or otherwise protected files.
- **Include system items** – Include or exclude items marked as "system" in the file system. Care must be taken when renaming system files as it may cause issues with stability of the operating system and applications.
- **Skip root folders when added as files** – When **Add folders as files** and **Include subfolders** options are enabled, this option skips adding top level folders for renaming and only adds the subfolders for renaming.

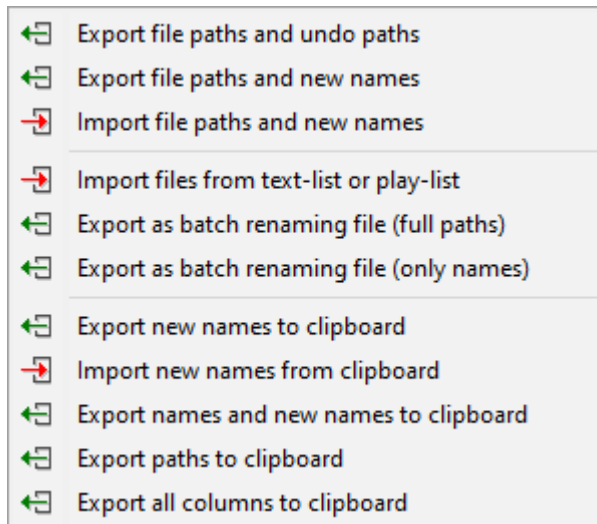
Masks:

- **Include** and **Exclude** – All added files (or folders as files) will be filtered according to the specified [masks](#). For example, if you enter `*.jpg;*.gif;*.png` as an *Include* mask, only files matching these extensions will be added. Everything else will be filtered out. The *Exclude* mask works in the opposite way, it will filter out everything that matches the mask.
- **Apply only to the file name, instead of full path** – The filter masks can be applied just to the filename or the whole path, whichever suits your needs.

The filter settings can be saved as default for future sessions using the option located below the **Save** button.

Export menu

The Export menu offers several options for exporting and importing the contents of the Files table.



Option	Description
Export file paths and undo paths	Exports <i>file paths</i> and <i>undo paths</i> to a CSV (comma separated) or TXT (tab separated) file. The <i>undo paths</i> are a result of a successful renaming operation, they represent original file paths before the renaming. This may be useful as a way for storing a backup of the renaming operation, which can later be imported back in order to restore the original names.
Export file paths and new names	Exports <i>file paths</i> and <i>new names</i> to a CSV (comma separated) or TXT (tab separated) file. This may be useful as a template for reuse in the future or for making changes to new names in an external application and then importing them back for renaming.
Import file paths and new names	Imports <i>file paths</i> and <i>new names</i> from a CSV (comma separated) or a TXT (tab separated) file. The importing feature supports both absolute and relative paths, which will get resolved relative to the path of the file being imported.

Option	Description
Import files from text-list or play-list	Imports <i>file paths</i> from a list file (one file per line) or a playlist file (M3U/PLS format). The importing feature supports both absolute and relative paths, which will get resolved relative to the path of the file being imported. Note that only those files which actually exist in the file system will be loaded into the Files table.
Export as batch renaming file	Generate a BAT/CMD script containing commands for executing the renaming of files. This script can be executed directly or via the Command Prompt, independently of ReNamer.
Export new names to clipboard	Export <i>new names</i> to clipboard. One name per line. This may be useful for performing adjustments to new names via an external application (e.g. Notepad, Excel, etc) and then importing back into ReNamer.
Import new names from clipboard	Imports <i>new names</i> from clipboard. Every name should be placed in a new line. If there are fewer lines in the clipboard text than the number of files in table, the excess files will have an empty new name. If there are more lines in the clipboard text than the number of files in table, the excess lines will be discarded.
Export names and new names to clipboard	Export <i>original names</i> and <i>new names</i> to clipboard. One pair of names per line, with names separated by a tab character.
Export paths to clipboard	Export <i>original file paths</i> of all files to clipboard. One path per line.
Export all columns to clipboard	Exports the content of the Files table (visible columns only) to clipboard, with values separated by a tab character.

Options menu

The Options menu offers various ad-hoc actions.

	Autosize columns	Shift+S
	Validate new names	Shift+V
	Fix conflicting new names	Shift+F
	Analyze sample text	Shift+A
	Apply rules to the clipboard	Shift+C
	Count marked and selected files	Shift+I
	Sort by path for renaming folders	Shift+R

Autosize columns

Auto-size all columns to accommodate the longest entry in each column.

Validate new names

Run the [validation](#) process to analyze the list of files and their target destinations for common problems. A warning is displayed for each potential problem discovered. Eliminate all warnings before renaming to avoid [failed renames](#).

Fix conflicting new names

Add incremental numbers as suffixes to avoid conflicts in names. For example, if a new name for a file is `Name1`, and such a file already exists in the target folder, then a suffix `(2)` is added to the new name. If more than one file has such name conflicts, further incremental numbers (`(3)`, `(4)` and so on) will be added as suffixes to ensure that each file gets a unique name.

To apply this automatically on every preview, enable *Fix conflicting new names on preview* in [Preview settings](#).

Analyze sample text

Open the [Analyze tool](#) to apply renaming rules to any arbitrary text.

It allows you to play with the renaming rules, tweak and experiment with them, and observe the effects without having to load any files.

Apply rules to the clipboard

Apply the current renaming rules to the clipboard content.

This is a very quick and useful mechanism for leveraging the power of renaming rules to modify externally provided text. For example, you could perform text replacements, adjust capitalization, and apply Regular Expressions.

Count marked and selected files

Display the number of marked items, selected items, and the total number of items in the Files table.

Sort by path for renaming folders

Sort all items in the Files table by the full path in reverse order.

This step is commonly used when [renaming folders](#) and their content in one operation, to ensure that the content of any folder is renamed before its parent folder.

Pascal Script

Reference for the Pascal Script syntax, types, functions, and example scripts.

Pascal Script: Overview

The [Pascal Script rule](#) in ReNamer allows advanced users to create custom renaming logic using a scripting language based on Pascal/Delphi syntax. This provides full programmatic control over filenames, paths, metadata, and file system operations, making it ideal for complex tasks that standard rules cannot handle easily.

A collection of specialized functions and extensions makes scripting particularly powerful for Unicode filename handling, dynamic meta tag extraction, file system interactions, and integrating external data or tools.

If you're new to scripting or programming, start with the [Basics](#) to learn the fundamentals step by step, and then explore the [Examples](#). The following sections provide reference details once you're comfortable with the fundamentals.

Key concepts

- **Script Execution:** Scripts are executed during the Preview stage just like other renaming rules, allowing you to manipulate the new name for every processed file.
- **Main Variable:** The core of most scripts is the `FileName` variable of type `WideString`, which represents the new basename with extension. Modify this variable to change the new name for a file.
- **Path Access:** The `FilePath` constant provides the full original path to the file, allowing direct file system operations, like reading content, applying external tools, etc.
- **Unicode Support:** ReNamer's implementation uses `WideString` as the primary string type for full Unicode compatibility, supporting non-English characters and international scripts.

Extensions and functions

ReNamer extends the standard Pascal Script engine with custom [types](#) and [functions](#) tailored for renaming tasks, including:

- Wide-string variants of common functions, e.g. `WideUpperCase`, `WideReplaceStr`.
- File utilities, e.g. `WideFileExists`, `FileReadTextLines`.
- [Meta tag](#) extraction via `CalculateMetaTag`.
- Support for [user-defined functions](#) (UDFs) and [importing external functions](#) from DLLs.
- Include directive `{ $INCLUDE 'filename.inc' }` for reusing code from external files.

Best practices

Follow these recommendations for reliable, efficient scripts:

- Always prefer `WideString` over `String` for paths and names to ensure Unicode compatibility.
- Avoid overriding built-in variables, types, or functions.
- Some functions can modify the file system (e.g. create folders, delete files), use these with caution.
- Test scripts thoroughly in Preview mode.

Warnings

- Scripts execute on every preview, which can impact performance with large file lists or heavy operations.
- If [auto preview options](#) are enabled, scripts can be triggered automatically on any change to rules or files, which may cause unintended repeated execution — particularly relevant for scripts that perform file system operations.
- Infinite loops or errors can freeze ReNamer, include safeguards like counters where needed.
- File system changes in scripts are immediate and irreversible.

Further reading

These articles will help you learn and master the power of Pascal Script:

- [Basics](#) – Syntax fundamentals, control structures, and quick start guide.
- [Types](#) – Data types and structures.
- [Functions](#) – Full reference of built-in functions.
- [Examples](#) – Educational snippets and how-tos.
- [Library](#) – Curated, ready-to-use scripts for common tasks and integrations with 3rd party tools.

You can also find examples, discussions and user contributed scripts on the [User Forum](#).

Pascal Script: Basics

This page provides a quick introduction to Pascal syntax and the basic structure of scripts.

Once you are comfortable with these concepts, you can jump into [Examples](#) to explore practical applications and educational snippets. For a broader introduction to the Pascal language, see this [tutorial by Tao Yue](#).

Basic structure

A script is divided into an optional declarations section and a mandatory executable section:

```
const
    // Constant declarations
type
    // Type declarations
var
    // Variable declarations
begin
    // Executable statements
end.
```

The `const`, `type`, and `var` sections are optional and can be omitted if not needed. The `begin`...`end.` block is required and is where all executable statements go, separated by semicolons `;`.

In the examples below, the following placeholders are used:

- `<Condition>` — an expression that evaluates to *true* or *false*, for example `X > 5`.
- `<Action>` — a code block that performs an operation, for example `X := Y + 5`.

Comments

Comments can be written in two forms, single-line and multi-line:

```
// This is a single-line comment
```



```
{ This comment is a  
  multi-line comment }
```

Branching

Branching allows different parts of the code to execute depending on a condition.

If-then

```
if <Condition> then  
  begin  
    <Action>  
  end;
```

<Action> is executed only if <Condition> is *true*, otherwise it is skipped.

If-then-else

```
if <Condition> then  
  begin  
    <Action-1>  
  end  
else  
  begin  
    <Action-2>  
  end;
```

If <Condition> is *true*, <Action-1> is executed, otherwise <Action-2> is executed.

Case/switch

```
case X of  
  1:  
    begin  
      <Action-1>  
    end;  
  2:  
    begin  
      <Action-2>  
    end;
```

```
else
  begin
    <Action-3>
  end;
end;
```

This structure tests a single expression (e.g. variable `X`) against multiple possible values (e.g. `1` and `2`). Each value has its own `<Action>` block. The values are checked in order of appearance, and the first matching block is executed, no further values are checked afterward. The `else` section executes if no match is found and is optional.

Loops

Loops allow a block of code to execute repeatedly until a condition is met.

For-to-do

```
for I := X to Y do
  begin
    <Action>
  end;
```

`<Action>` executes a fixed number of times. The iterator variable `I` starts at `X` and increments by 1 on each iteration until it reaches `Y`. To count in reverse, use `downto` instead of `to`.

While-do

```
while <Condition> do
  begin
    <Action>
  end;
```

`<Action>` executes repeatedly as long as `<Condition>` is *true*. The condition is checked *before* each iteration, so if it is *false* from the start, the loop body never executes.

Repeat-until

```
repeat
  <Action>
until <Condition>;
```

`<Action>` executes repeatedly until `<Condition>` becomes *true*. The condition is checked *after* each iteration, so the loop always executes at least once.

Loop control

Within any loop, two statements provide additional flow control:

- `Break;` — immediately exits the loop and continues execution after it.
- `Continue;` — skips the rest of the current iteration and moves to the next one.

Both are commonly used inside an `if...then` block:

```
if <Condition> then Break;  
if <Condition> then Continue;
```

Early exit

The `Exit;` statement terminates the current script immediately, regardless of where it appears in the code:

```
if <Condition> then Exit;
```

Use it sparingly, because it causes abrupt termination, which can make scripts harder to read and debug.

Pascal Script: Types

This page lists and explains all supported types in Pascal Script used within ReNamer.

Integer types

Integer types store whole numbers and are used for counting, indexing, and arithmetic operations. Choose the appropriate size based on the range of values you need to represent.

Type	Size	Minimum Value	Maximum Value
Byte	1 byte	0	255
ShortInt	1 byte	-128	127
Word	2 bytes	0	65,535
SmallInt	2 bytes	-32,768	32,767
Cardinal	4 bytes	0	4,294,967,295
Integer	4 bytes	-2,147,483,648	2,147,483,647
Int64	8 bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Floating point types

Floating point types store decimal numbers and are used for calculations requiring fractional values or very large/small magnitudes.

Type	Size	Smallest Number	Largest Number
Single	4 bytes	1.5×10^{-45}	3.4×10^{38}
Double	8 bytes	5.0×10^{-324}	1.7×10^{308}
Extended	10 bytes	3.6×10^{-4951}	1.1×10^{4932}

String types

String types store text and sequences of characters.

Type	Description
<code>Char</code>	Stores a single 8-bit character.
<code>String</code>	Holds a sequence of 8-bit characters. Commonly used for ANSI or UTF-8 encoded text.
<code>AnsiChar</code>	Alias for <i>Char</i> type.
<code>AnsiString</code>	Alias for <i>String</i> type.
<code>WideChar</code>	Stores a single 16-bit character.
<code>WideString</code>	Holds a sequence of 16-bit characters. Commonly used for UCS-2 or UTF-16 encoded text.
<code>UnicodeChar</code>	Alias for <i>WideChar</i> type.
<code>UnicodeString</code>	Alias for <i>WideString</i> type.

The default encoding for `String` type and the conversion process to/from `WideString` type differ between versions of ReNamer.

ReNamer version	Default encoding for <i>String</i> type	Conversion between <i>WideString</i> and <i>String</i> types
7.0 and later	UTF-8	Automatic, on assignment
Prior to 7.0	Active system code page	Manual, using conversion functions

The [Unicode](#) article highlights the differences between various encodings.

Mixed types

Type	Description
<code>Boolean</code>	Logical value which can be either <code>True</code> or <code>False</code> .
<code>Array</code>	Single and multi dimensional indexable sequence of data.
<code>Record</code>	A structure that holds a set of different data types.
<code>Variant</code>	Type that can store values of different types and convert between them at runtime.
<code>PChar</code>	Pointer to a <i>Char</i> value, and can also be used to point to characters within a string.

A few examples:

```
type
  TPerson = record
```

```

    Name: WideString;
    Age: Integer;
end;
var
    Person: TPerson;
    Numbers: Array of Integer;

```

Extra types

Several extra types have been defined to simplify the use of some functions.

Type	Declared as	Description
<code>TDateTime</code>	<code>Double</code>	Represents a date and time.
<code>TStringsArray</code>	<code>Array of WideString</code>	Indexed list of <i>WideString</i> values. <i>Deprecated in v5.74.4 Beta. Please use <code>TWideStringArray</code> instead.</i>
<code>TWideStringArray</code>	<code>Array of WideString</code>	Indexed list of <i>WideString</i> values. <i>Added in v5.74.4 Beta. Replaces ambiguous <code>TStringsArray</code> type.</i>
<code>TAnsiStringArray</code>	<code>Array of AnsiString</code>	Indexed list of <i>AnsiString</i> values. <i>Added in v5.74.4 Beta.</i>
<code>TIntegerArray</code>	<code>Array of Integer</code>	Indexed list of <i>Integer</i> values. <i>Added in v7.3.0.4 Beta.</i>

Enumerations and Sets

An enumeration is simply a fixed range of named values.

For example, the fundamental *Boolean* data type can be considered as an enumeration consisting of two values: *True* and *False*.

A variable of an enumeration type can be assigned a single value from the enumeration.

```

type
    TDay = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
var

```

```
    Day: TDay;
begin
    Day := Mon;
    if Day <> Tue then
        Day := Wed;
    end.
```

Sets allow you to define variables which can hold multiple values out of the enumeration.

```
type
    TDay = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
    TDays = set of TDay;
var
    Days: TDays;
begin
    Days := [Mon, Tue, Wed];
    if Sun in Days then
        Days := Days - [Sun];
    end.
```

Pascal Script: Functions

This page provides a complete reference of all functions available in the [Pascal Script rule](#) within ReNamer, organized by category.

A function generally performs some work, it may take some input parameters, and may produce or return some output. Functions that do not return anything are also known as procedures.

In this document, functions have the following general signature: `FunctionName (Parameters): ReturnType`, where the parameters and the return type vary by function.

Parameters to function may have prefixes indicating how they're used:

- `const` — Value is read-only and cannot be modified.
- `var` — Value can be modified by the function and changes are returned.
- `out` — Variable receives output from the function.

A common "Wide" prefix in function names indicates that the function operates on the `WideString` type. In versions prior to 7.0, this was the only reliable way to handle Unicode text, as `String` used the system code page. From version 7.0 onwards, `String` is UTF-8 encoded with automatic conversion to and from `WideString`, so both variants are generally interchangeable (e.g. `Trim` vs `WideTrim`) — however, for optimal performance, use the variant that matches your input and output types. See [String Types](#) for details.

For a demonstration of these functions in practice, see the [Examples](#) article.

Length Management

Function	Description
<code>SetLength (var S: Array; NewLength: Integer)</code>	Sets the length of array variable <code>S</code> to <code>NewLength</code> .
<code>SetLength (var S: String; NewLength: Integer)</code>	Sets the length of string variable <code>S</code> to <code>NewLength</code> .
<code>SetLength (var S: WideString; NewLength: Integer)</code>	Sets the length of <code>WideString</code> <code>S</code> to <code>NewLength</code> .
<code>Length (const S: Array): Integer</code>	Returns the length of array <code>S</code> (number of elements).
<code>Length (const S: String): Integer</code>	Returns the length of string <code>S</code> (number of characters).
<code>Length (const S: WideString): Integer</code>	Returns the length of <code>WideString</code> <code>S</code> (number of characters).

String Handling

Function	Description
<code>Insert (Source: String; var S: String; Index: Integer)</code>	Inserts the string <code>S</code> into string <code>Source</code> at position <code>Index</code> .
<code>Delete (var S: String; Index, Count: Integer)</code>	Deletes <code>Count</code> characters from the string <code>S</code> , starting from position <code>Index</code> .
<code>Copy (S: String; Index, Count: Integer): String</code>	Copies <code>Count</code> characters from string <code>S</code> , starting at position <code>Index</code> , and returns them as a new string.
<code>Pos (Substr: String; S: String): Integer</code>	Returns the position of string <code>Substr</code> in string <code>S</code> . Returns 0 if not found.
<code>Trim (const S: String): String</code>	Strip blank characters (including spaces and control characters with ordinal values under 32) at the beginning and end of string <code>S</code> .
<code>StringOfChar (C: Char; Count: Integer): String</code>	Create a string of length <code>Count</code> and fill it with the character <code>C</code> .

Note: String indexes are 1-based, so the first character in string `S` is `S[1]`.

String Array Handling

Function	Description
<code>SplitString (const Input, Delimiter: String): TArray<String></code>	Splits the string <code>Input</code> into an array of substrings, using <code>Delimiter</code> as the separator. <i>Added in v7.8.0.4 Beta.</i>
<code>JoinStrings (const Strings: TArray<String>; const Delimiter: String): String</code>	Joins all elements of the array <code>Strings</code> into a single string, inserting <code>Delimiter</code> between each element. <i>Added in v7.8.0.4 Beta.</i>
<code>AppendStringArray (var Strings: TArray<String>; const NewString: String)</code>	Appends <code>NewString</code> to the end of the array <code>Strings</code> . <i>Added in v7.8.0.4 Beta.</i>
<code>CopyStringArray (const Strings: TArray<String>; Index, Count: Integer): TArray<String></code>	Returns a new array containing <code>Count</code> elements from <code>Strings</code> , starting at position <code>Index</code> . <i>Added in v7.8.0.4 Beta.</i>
<code>ConcatStringArrays (const A, B: TArray<String>): TArray<String></code>	Returns a new array containing all elements of <code>A</code> followed by all elements of <code>B</code> . <i>Added in v7.8.0.4 Beta.</i>
<code>InsertStringArray (const S: String; var Strings: TArray<String>; Index: Integer)</code>	Inserts the string <code>S</code> into the array <code>Strings</code> at position <code>Index</code> , shifting subsequent elements forward. <i>Added in v7.8.0.4 Beta.</i>
<code>DeleteStringArray (var Strings: TArray<String>; Index, Count: Integer)</code>	Deletes <code>Count</code> elements from the array <code>Strings</code> , starting at position <code>Index</code> , shifting subsequent elements back. <i>Added in v7.8.0.4 Beta.</i>

Function	Description
<code>FindStringArray (const Find: String; const Strings: TAnsiStringArray; CaseSensitive: Boolean): Integer</code>	Searches the array <code>Strings</code> for the first occurrence of <code>Find</code> string. Returns the index of the matching element, or -1 if not found. The <code>CaseSensitive</code> parameter controls whether the comparison is case-sensitive. <i>Added in v7.8.0.4 Beta.</i>

Note: Dynamic array indexes are 0-based, so the first element in array `A` is `A[0]`.

Wide String Handling

Function	Description
<code>WideInsert (const Substr: WideString; var Dest: WideString; Index: Integer)</code>	Inserts <code>Substr</code> in <code>Dest</code> at position <code>Index</code> .
<code>WideDelete (var S: WideString; Index, Count: Integer)</code>	Deletes <code>Count</code> characters from <code>S</code> , starting from position <code>Index</code> .
<code>WideDeleteRight (var S: WideString; Index, Count: Integer)</code>	Deletes <code>Count</code> characters from <code>S</code> , starting from position <code>Index</code> from the end and counting towards the start. <i>Added in v6.0.0.9 Alpha.</i>
<code>WideSetLength (var S: WideString; NewLength: Integer)</code>	Changes the length of string <code>S</code> to <code>NewLength</code> . If the new length is smaller than the original, the string is truncated. If the new length is larger, the string is expanded but the additional characters will be uninitialized and may contain random data.
<code>WideLength (const S: WideString): Integer</code>	Returns the length of WideString <code>S</code> .
<code>WideCopy (const S: WideString; Index, Count: Integer): WideString</code>	Returns <code>Count</code> characters from <code>S</code> , starting at position <code>Index</code> .
<code>WideCopyRight (const S: WideString; Index, Count: Integer): WideString</code>	Returns <code>Count</code> characters from <code>S</code> , starting at position <code>Index</code> from the end and counting towards the start. <i>Added in v6.0.0.9 Alpha.</i>
<code>WidePos (const SubStr, S: WideString): Integer</code>	Finds the first occurrence of <code>SubStr</code> in <code>S</code> . Returns the position of first occurrence, or 0 if not found.
<code>WidePosEx (const SubStr, S: WideString; Offset: Cardinal): Integer</code>	Finds the first occurrence of <code>SubStr</code> in <code>S</code> starting from position <code>Offset</code> . Returns the position of first occurrence, or 0 if not found.
<code>WideUpperCase (const S: WideString): WideString</code>	Returns the uppercase version of WideString <code>S</code> .
<code>WideLowerCase (const S: WideString): WideString</code>	Returns the lowercase version of WideString <code>S</code> .
<code>WideCompareStr (const S1, S2: WideString): Integer</code>	Compares two WideStrings case-sensitively. Returns a value less than 0 if <code>S1 < S2</code> , 0 if <code>S1 = S2</code> , or greater than 0 if <code>S1 > S2</code> .

Function	Description
<code>WideCompareText (const S1, S2: WideString): Integer</code>	Compares two WideStrings case-insensitively. Returns a value less than 0 if S1 < S2, 0 if S1 = S2, or greater than 0 if S1 > S2.
<code>WideSameText (const S1, S2: WideString): Boolean</code>	Compares two WideStrings case-insensitively. Returns <code>True</code> if identical, otherwise <code>False</code> .
<code>WideTextPos (const SubStr, S: WideString): Integer</code>	Like <code>WidePos</code> but case-insensitive.
<code>WideTextPosEx (const SubStr, S: WideString; Offset: Cardinal): Integer</code>	Like <code>WidePosEx</code> but case-insensitive. <i>Added in v6.6.0.1 Beta.</i>
<code>WideTrim (const S: WideString): WideString</code>	Removes leading and trailing spaces and control characters from string <code>S</code> .
<code>WideTrimChars (const S, CharsToTrim: WideString): WideString</code>	Removes characters that appear in <code>CharsToTrim</code> from the beginning and end of <code>S</code> . <i>Added in v6.0.0.9 Alpha.</i>
<code>WideTrimCharsLeft (const S, CharsToTrim: WideString): WideString</code>	Removes characters that appear in <code>CharsToTrim</code> from the beginning of <code>S</code> . <i>Added in v6.0.0.9 Alpha.</i>
<code>WideTrimCharsRight (const S, CharsToTrim: WideString): WideString</code>	Removes characters that appear in <code>CharsToTrim</code> from the end of <code>S</code> . <i>Added in v6.0.0.9 Alpha.</i>
<code>WideReverseString (const S: WideString): WideString</code>	Returns a reversed version of string <code>S</code> . <i>Added in v6.7.0.4 Beta.</i>
<code>WideRepeatString (const S: WideString; Count: Integer): WideString</code>	Repeats string <code>S</code> the specified number of times. <i>Added in v6.9.0.3 Beta.</i>
<code>WideReplaceStr (const S, OldPattern, NewPattern: WideString): WideString</code>	Replaces all case-sensitive occurrences of <code>OldPattern</code> with <code>NewPattern</code> in string <code>S</code> .
<code>WideReplaceText (const S, OldPattern, NewPattern: WideString): WideString</code>	Replaces all case-insensitive occurrences of <code>OldPattern</code> with <code>NewPattern</code> in string <code>S</code> .
<code>WideCaseSentence (const S: WideString): WideString</code>	Returns sentence case version of <code>S</code> . Only the first alphabetic character is capitalized, all others are lowercase. <i>Added in v6.0.0.3 Alpha.</i>
<code>WideCaseCapitalize (const S: WideString): WideString</code>	Returns title case version of <code>S</code> . First letter of every word is capitalized, all others are lowercase.
<code>WideCaseInvert (const S: WideString): WideString</code>	Inverts the case of all characters in <code>S</code> .

Wide String Array Handling

Function	Description
----------	-------------

<code>WideSplitString (const Input, Delimiter: WideString): TWideStringArray</code>	Splits <code>Input</code> wherever <code>Delimiter</code> occurs and returns an array of the split parts. The <code>Delimiter</code> can be a multi-character string (not limited to single characters like comma or space). The returned parts do not include the delimiter.
<code>WideJoinStrings (const Strings: TWideStringArray; const Delimiter: WideString): WideString</code>	Joins all items from <code>Strings</code> into a single <code>WideString</code> , with <code>Delimiter</code> inserted between items.
<code>WideAppendStringArray (var Strings: TWideStringArray; const NewString: WideString)</code>	Adds a <code>WideString</code> to the end of a <code>WideString</code> array. <i>Added in v7.5.0.3 Beta.</i>
<code>WideCopyStringArray (const Strings: TWideStringArray; Index, Count: Integer): TWideStringArray</code>	Copies a portion of a <code>WideString</code> array by taking <code>Count</code> elements starting from <code>Index</code> position. <i>Added in v7.5.0.3 Beta.</i>
<code>WideConcatStringArrays (const A, B: TWideStringArray): TWideStringArray</code>	Concatenates two <code>WideString</code> arrays. <i>Added in v7.5.0.3 Beta.</i>
<code>WideInsertStringArray (const S: WideString; var Strings: TWideStringArray; Index: Integer)</code>	Inserts the string <code>S</code> into the array <code>Strings</code> at position <code>Index</code> , shifting subsequent elements forward. <i>Added in v7.8.0.4 Beta.</i>
<code>WideDeleteStringArray (var Strings: TWideStringArray; Index, Count: Integer)</code>	Deletes <code>Count</code> elements from the array <code>Strings</code> , starting at position <code>Index</code> , shifting subsequent elements back. <i>Added in v7.8.0.4 Beta.</i>
<code>WideFindStringArray (const Find: WideString; const Strings: TWideStringArray; CaseSensitive: Boolean): Integer</code>	Searches the array <code>Strings</code> for the first occurrence of <code>Find</code> . Returns the index of the matching element, or -1 if not found. The <code>CaseSensitive</code> parameter controls whether the comparison is case-sensitive. <i>Added in v7.8.0.4 Beta.</i>

Wide Character Handling

Function	Description
<code>IsWideCharUpper (WC: WideChar): Boolean</code>	Checks if character is uppercase according to Unicode character properties.
<code>IsWideCharLower (WC: WideChar): Boolean</code>	Checks if character is lowercase according to Unicode character properties.
<code>IsWideCharDigit (WC: WideChar): Boolean</code>	Checks if character is a numeric digit (0-9 in any script).
<code>IsWideCharSpace (WC: WideChar): Boolean</code>	Checks if character is whitespace (spaces, tabs, line breaks, and other Unicode whitespace characters).
<code>IsWideCharPunct (WC: WideChar): Boolean</code>	Checks if character is a punctuation mark according to Unicode character properties.
<code>IsWideCharCntrl (WC: WideChar): Boolean</code>	Checks if character is a control character (non-printing characters like null, backspace, escape, etc.).
<code>IsWideCharBlank (WC: WideChar): Boolean</code>	Checks if character is a blank (space or tab).
<code>IsWideCharXDigit (WC: WideChar): Boolean</code>	Checks if character is a hexadecimal digit (0-9, A-F, a-f).

Function	Description
<code>IsWideCharAlpha (WC: WideChar): Boolean</code>	Checks if character is alphabetic according to Unicode character properties. This includes letters from all scripts (Latin, Arabic, Chinese, Cyrillic, etc.).
<code>IsWideCharAlphaNumeric (WC: WideChar): Boolean</code>	Checks if character is alphabetic or numeric according to Unicode character properties.
<code>IsWideWordBoundaryLeft (const Subject: WideString; CharPosition: Integer): Boolean</code>	Checks if a character at the specified position is on a word boundary to the left. A word boundary to the left exists if: (1) it's the first character and is a word character, or (2) it's between a word character and a non-word character. <i>Added in v6.6.0.1 Beta.</i>
<code>IsWideWordBoundaryRight (const Subject: WideString; CharPosition: Integer): Boolean</code>	Checks if a character at the specified position is on a word boundary to the right. A word boundary to the right exists if: (1) it's the last character and is a word character, or (2) it's between a word character and a non-word character. <i>Added in v6.6.0.1 Beta.</i>
<code>WideCharUpper (const WC: WideChar): WideChar</code>	Converts character to uppercase according to Unicode case mapping rules. Non-alphabetic characters are returned unchanged.
<code>WideCharLower (const WC: WideChar): WideChar</code>	Converts character to lowercase according to Unicode case mapping rules. Non-alphabetic characters are returned unchanged.
<code>WideChr (Code: Word): WideChar</code>	Creates a character from a Unicode code point. <i>Added in v6.9.0.3 Beta.</i>

Note: These functions use Windows character classification based on Unicode character properties. For technical details, see the [GetStringTypeW function](#) in the Windows API documentation.

Unicode Conversion

Function	Description
<code>WideToAnsi (const WS: WideString): String</code>	Converts WideString to AnsiString. <i>In v7.0 and later, this conversion happens automatically on assignment.</i>
<code>AnsiToWide (const S: String): WideString</code>	Converts AnsiString to WideString. <i>In v7.0 and later, this conversion happens automatically on assignment.</i>
<code>UTF8Encode (const WS: WideString): String</code>	Converts WideString to UTF-8 encoded string.
<code>UTF8Decode (const S: String): WideString</code>	Converts UTF-8 encoded string to WideString.

Function	Description
<code>WinCPToUTF8 (const S: String): String</code>	Converts a string encoded with the active system code page to UTF-8. <i>Added in v7.4.0.3 Beta.</i>
<code>UTF8ToWinCP (const S: String): String</code>	Converts UTF-8 encoded string to a string encoded with the active system code page . <i>Added in v7.4.0.3 Beta.</i>
<code>ConvertEncoding (const S, FromEncoding, ToEncoding: String): String</code>	Converts the encoding of string <code>S</code> from <code>FromEncoding</code> to <code>ToEncoding</code> . Supported encoding identifiers are listed in the table below. <i>Added in v7.8.0.4 Beta.</i>

See [String Types](#) for additional information on default encoding and conversion procedures.

Supported encoding identifiers

Encoding identifiers are case-insensitive and the separating dashes are optional.

- **Unicode**

- `UTF-8` — Unicode encoded as UTF-8.
- `UTF-8BOM` — Same as UTF-8, but with a byte order mark (BOM) prepended.

- **Windows code pages** — Legacy ANSI encodings used by Windows applications for various scripts and regions.

- `cp1250` `cp1251` `cp1252` `cp1253` `cp1254` `cp1255` `cp1256` `cp1257` `cp1258`

- **OEM/DOS code pages** — Legacy encodings used by DOS applications and the Windows command line.

- `cp437` `cp850` `cp852` `cp865` `cp866` `cp874` `cp932` `cp936` `cp949` `cp950`

- **ISO 8859** — Single-byte encodings standardised by ISO, covering a wide range of Latin and non-Latin scripts.

- `ISO-8859-1` `ISO-8859-2` `ISO-8859-3` `ISO-8859-4` `ISO-8859-5` `ISO-8859-7` `ISO-8859-9`
`ISO-8859-10` `ISO-8859-13` `ISO-8859-14` `ISO-8859-15` `ISO-8859-16`

- **KOI8** — Cyrillic encodings widely used on Unix systems.

- `KOI8-R` `KOI8-U` `KOI8-RU`

- **Other**

- `Macintosh` — Legacy Apple Macintosh encoding, found in older files and some metadata formats.

Console Output Conversion

OEM-defined character set is commonly used in console application output.

Function	Description
----------	-------------

<code>OemToAnsi (const S: String): String</code>	Converts OEM string to ANSI string. <i>Added in v6.6.0.2 Beta.</i>
<code>OemToWide (const S: String): WideString</code>	Converts OEM string to WideString. <i>Added in v6.6.0.2 Beta.</i>
<code>AnsiToOem (const S: String): String</code>	Converts ANSI string to OEM string. <i>Added in v6.6.0.2 Beta.</i>
<code>WideToOem (const S: WideString): String</code>	Converts WideString to OEM string. <i>Added in v6.6.0.2 Beta.</i>

Basic Conversion

Function	Description
<code>BoolToStr (B: Boolean): String</code>	Converts boolean value to string. Returns "True" or "False".
<code>IntToStr (Value: Integer): String</code>	Converts an integer to string. For example, <code>IntToStr(123)</code> returns <code>'123'</code> string. Warning: Be cautious when passing Int64 values. They will be automatically type cast to Integer, which significantly reduces the range of possible values (see Types for details). For Int64 values, use <code>Int64ToStr</code> or <code>FormatFloat</code> to avoid range loss.
<code>Int64ToStr (Value: Int64): String</code>	Converts an Int64 value to string.
<code>StrToInt (const S: String): Integer</code>	Converts a string to integer. For example, <code>StrToInt('123')</code> returns <code>123</code> integer. Warning: Raises an error if the string cannot be converted to an integer.
<code>StrToInt64 (const S: String): Int64</code>	Converts a string to Int64. Raises an error if conversion fails.
<code>StrToIntDef (const S: String; const Default: Integer): Integer</code>	Converts a string to integer. Returns <code>Default</code> value if conversion fails instead of raising an error.
<code>StrToInt64Def (const S: String; Default: Int64): Int64</code>	Converts a string to Int64. Returns <code>Default</code> value if conversion fails.
<code>TryStrToInt (const S: String; out Value: Integer): Boolean</code>	Converts a string to integer. Returns <code>False</code> if conversion fails instead of raising an error. The converted value is stored in <code>Value</code> parameter.
<code>FloatToStr (Value: Extended): String</code>	Converts floating point value to string using default system format.
<code>StrToFloat (const S: String): Extended</code>	Converts string to floating point value. Raises an error if conversion fails.

Function	Description
<code>StrToFloatDef (const S: String; const Default: Extended): Extended</code>	Converts string to floating point value. Returns <code>Default</code> value if conversion fails.
<code>FormatFloat (const Format: String; Value: Extended): String</code>	Converts floating point value to string using custom format. See floating point format specifiers in the table below.
<code>DateToStr (D: TDateTime): String</code>	Converts date to string using system short date format (e.g., dd/mm/yyyy).
<code>StrToDate (const S: String): TDateTime</code>	Converts date string to TDateTime using system short date format (e.g., dd/mm/yyyy).
<code>IntToHex (Value: Integer; Digits: Integer): String</code>	Converts integer to hexadecimal representation. The <code>Digits</code> parameter specifies minimum number of digits (padded with zeros if needed). Examples: <code>IntToHex(1234, 1)</code> returns <code>'4D2'</code> <code>IntToHex(1234, 8)</code> returns <code>'000004D2'</code>
<code>HexToInt (const HexNum: String): Integer</code>	Converts hexadecimal string to integer. Raises an error if conversion fails.
<code>HexToIntDef (const HexNum: String; Default: Integer): Integer</code>	Converts hexadecimal string to integer. Returns <code>Default</code> value if conversion fails.
<code>IntToRoman (Value: Integer): String</code>	Converts decimal number to Roman numerals. <i>Added in v7.3.0.4 Beta.</i>
<code>RomanToInt (const S: String): Integer</code>	Converts Roman numerals to decimal number. <i>Added in v7.3.0.4 Beta.</i>
<code>RomanToIntDef (const S: String; Default: Integer): Integer</code>	Converts Roman numerals to decimal number. Returns <code>Default</code> if conversion fails. <i>Added in v7.3.0.4 Beta.</i>
<code>TryRomanToInt (const S: String; out Value: Integer): Boolean</code>	Converts Roman numerals to decimal number. Returns <code>False</code> if conversion fails instead of raising an error. <i>Added in v7.3.0.4 Beta.</i>
<code>Ord (X: Char): Byte</code>	Returns ordinal value (byte representation) of a character.
<code>Chr (X: Byte): Char</code>	Returns a character from its ordinal value (byte representation).

Floating point format specifiers

Specifier	Description
<code>0</code> (zero)	Digit placeholder. If a digit exists in this position, it's copied to output, otherwise a "0" is inserted.
<code>#</code> (hash)	Digit placeholder. If a digit exists in this position, it's copied to output, otherwise nothing is inserted.
<code>.</code> (dot)	Decimal point. The first "." determines the decimal separator location. Additional "." characters are ignored.

Specifier	Description
<code>,</code> (comma)	Thousand separator. Inserts separators between each group of three digits to the left of the decimal point.

Date and Time

Function	Description
<code>Date: TDateTime</code>	Returns the current system date.
<code>Time: TDateTime</code>	Returns the current system time.
<code>Now: TDateTime</code>	Returns the current system date and time.
<code>EncodeDate (Year, Month, Day: Word): TDateTime</code>	Creates a date value from year, month, and day components. Valid ranges: Year = 0..9999, Month = 1..12, Day = 1..31 (depending on month/year). Raises an error if parameters are invalid.
<code>EncodeTime (Hour, Min, Sec, MSec: Word): TDateTime</code>	Creates a time value from hour, minute, second, and millisecond components. Valid ranges: Hour = 0..23, Min = 0..59, Sec = 0..59, MSec = 0..999. Raises an error if parameters are invalid.
<code>EncodeDateTime (Year, Month, Day, Hour, Minute, Second, MilliSecond: Word): TDateTime</code>	Creates a date-time value from date and time components. Similar to combining <code>EncodeDate</code> and <code>EncodeTime</code> . <i>Added in v6.2.0.5 Beta.</i>
<code>TryEncodeDate (Year, Month, Day: Word; var Date: TDateTime): Boolean</code>	Like <code>EncodeDate</code> but returns <code>True</code> / <code>False</code> instead of raising an error. If successful, the generated date value is written to the <code>Date</code> parameter.
<code>TryEncodeTime (Hour, Min, Sec, MSec: Word; var Time: TDateTime): Boolean</code>	Like <code>EncodeTime</code> but returns <code>True</code> / <code>False</code> instead of raising an error. If successful, the generated time value is written to the <code>Time</code> parameter.
<code>TryEncodeDateTime (Year, Month, Day, Hour, Minute, Second, MilliSecond: Word; out ADateTime: TDateTime): Boolean</code>	Like <code>EncodeDateTime</code> but returns <code>True</code> / <code>False</code> instead of raising an error. If successful, the generated date-time value is written to the <code>ADateTime</code> parameter. <i>Added in v6.2.0.5 Beta.</i>
<code>DecodeDate (const DateTime: TDateTime; var Year, Month, Day: Word)</code>	Extracts year, month, and day components from a <code>TDateTime</code> value.
<code>DecodeTime (const DateTime: TDateTime; var Hour, Min, Sec, MSec: Word)</code>	Extracts hour, minute, second, and millisecond components from a <code>TDateTime</code> value.
<code>DecodeDateTime (const DateTime: TDateTime; out Year, Month, Day, Hour, Minute, Second, MilliSecond: Word)</code>	Extracts both date and time components from a <code>TDateTime</code> value. <i>Added in v6.2.0.5 Beta.</i>

Function	Description
<code>ComposeDateTime (const Date, Time: TDateTime): TDateTime</code>	Combines separate date and time values into a single TDateTime value. <i>Added in v6.2.0.5 Beta.</i>
<code>DateTimeToUnix (D: TDateTime): Int64</code>	Converts TDateTime value to Unix timestamp.
<code>UnixToDateTime (U: Int64): TDateTime</code>	Converts Unix timestamp to TDateTime value.
<code>FormatDateTime (const Format: String; DateTime: TDateTime): String</code>	Converts date and time to string using a custom date and time format .
<code>ScanDateTime (const Pattern, Subject: String): TDateTime</code>	Parses a date/time string according to a date and time format pattern. <i>Added in v7.1.0.3 Beta.</i>
<code>TryScanDateTime (const Pattern, Subject: String; out DateTime: TDateTime): Boolean</code>	Like <code>ScanDateTime</code> but returns <code>True</code> / <code>False</code> instead of raising an error. <i>Added in v7.1.0.3 Beta.</i>
<code>IncYear (const AValue: TDateTime; const ANumberOfYears: Integer): TDateTime</code>	Adds or subtracts years from a TDateTime value.
<code>IncMonth (const AValue: TDateTime; ANumberOfMonths: Integer): TDateTime</code>	Adds or subtracts months from a TDateTime value.
<code>IncWeek (const AValue: TDateTime; const ANumberOfWeeks: Integer): TDateTime</code>	Adds or subtracts weeks from a TDateTime value.
<code>IncDay (const AValue: TDateTime; const ANumberOfDays: Integer): TDateTime</code>	Adds or subtracts days from a TDateTime value.
<code>IncHour (const AValue: TDateTime; const ANumberOfHours: Int64): TDateTime</code>	Adds or subtracts hours from a TDateTime value.
<code>IncMinute (const AValue: TDateTime; const ANumberOfMinutes: Int64): TDateTime</code>	Adds or subtracts minutes from a TDateTime value.
<code>IncSecond (const AValue: TDateTime; const ANumberOfSeconds: Int64): TDateTime</code>	Adds or subtracts seconds from a TDateTime value.
<code>IncMilliSecond (const AValue: TDateTime; const ANumberOfMilliSeconds: Int64): TDateTime</code>	Adds or subtracts milliseconds from a TDateTime value.
<code>SecondSpan (const ANow, AThen: TDateTime): Double</code>	Calculates approximate number of seconds between two date-time values. <i>Added in v6.2.0.5 Beta.</i>
<code>DayOfWeek (const DateTime: TDateTime): Word</code>	Returns day number of the week (1=Monday, 2=Tuesday, 3=Wednesday, 4=Thursday, 5=Friday, 6=Saturday, 7=Sunday). <i>Before v6.1 this function returned 1=Sunday to 7=Saturday.</i>
<code>DayOfMonth (const DateTime: TDateTime): Word</code>	Returns the day number of the month. <i>Added in v6.1.</i>
<code>DayOfYear (const DateTime: TDateTime): Word</code>	Returns the day number of the year. <i>Added in v6.1.</i>

Function	Description
<code>WeekOfMonth (const DateTime: TDateTime): Word</code>	Returns the week number of the month. <i>Added in v6.1.</i>
<code>WeekOfYear (const DateTime: TDateTime): Word</code>	Returns the week number of the year. <i>Added in v6.1.</i>

File Management

Function	Description
<code>WideFileSize (const FileName: WideString): Int64</code>	Returns file size in bytes. Returns -1 if file does not exist.
<code>WideFileExists (const FileName: WideString): Boolean</code>	Checks if specified file exists.
<code>WideDirectoryExists (const Directory: WideString): Boolean</code>	Checks if specified directory exists.
<code>WideForceDirectories (const Dir: WideString): Boolean</code>	Ensures all directories in the path exist, creating them recursively if needed. Returns <code>True</code> if all folders exist or were successfully created.
<code>WideCreateDir (const Dir: WideString): Boolean</code>	Creates specified directory (non-recursive). Returns <code>True</code> on success.
<code>WideRemoveDir (const Dir: WideString): Boolean</code>	Removes a directory if it is empty. Returns <code>True</code> on success. <i>Added in v6.4.0.1 Beta.</i>
<code>WideDeleteFile (const FileName: WideString): Boolean</code>	Deletes file from disk. Returns <code>True</code> on success.
<code>WideDeleteToRecycleBin (const FileName: WideString): Boolean</code>	Moves file or folder to Recycle Bin. Returns <code>True</code> on success. Full path must be provided (a quirk of Windows API). <i>Added in v6.4.0.1 Beta.</i>
<code>WideRenameFile (const OldName, NewName: WideString): Boolean</code>	Renames file from <code>OldName</code> to <code>NewName</code> . Returns <code>True</code> on success.
<code>WideCopyFile (const FromFile, ToFile: WideString; FailIfExists: Boolean): Boolean</code>	Copies file from <code>FromFile</code> to <code>ToFile</code> . If <code>FailIfExists</code> is <code>True</code> , operation fails when destination exists; otherwise destination is overwritten. Returns <code>True</code> on success.
<code>WideFileSearch (const Name, DirList: WideString): WideString</code>	Searches for a file named <code>Name</code> in directories listed in <code>DirList</code> (semicolon-delimited list of path names). Returns full path if found, empty string otherwise.

Function	Description
<code>WideScanDirForFiles (const Dir: WideString; var Files: TWideStringArray; const Recursive, IncludeHidden, IncludeSystem: Boolean; const Mask: WideString)</code>	Scans a directory for files and returns the list in the <code>Files</code> array. <code>Dir</code> — Directory to scan <code>Files</code> — Array to receive the file list <code>Recursive</code> — Scan subdirectories <code>IncludeHidden</code> — Include hidden files <code>IncludeSystem</code> — Include system files <code>Mask</code> — Wildcard mask to filter files, or empty string for all files
<code>WideScanDirForFolders (const Dir: WideString; var Folders: TWideStringArray; const Recursive, IncludeHidden, IncludeSystem: Boolean)</code>	Scans a directory for folders and returns the list in the <code>Folders</code> array. <code>Dir</code> — Directory to scan <code>Folders</code> — Array to receive the folder list <code>Recursive</code> — Scan subdirectories <code>IncludeHidden</code> — Include hidden folders <code>IncludeSystem</code> — Include system folders

File Name Utilities

Function	Description
<code>WideExtractFilePath (const FileName: WideString): WideString</code>	Returns drive and directory portion with trailing path delimiter (e.g., "C:\Folder").
<code>WideExtractFileDir (const FileName: WideString): WideString</code>	Returns drive and directory portion without trailing path delimiter (e.g., "C:\Folder").
<code>WideExtractFileDrive (const FileName: WideString): WideString</code>	Returns the drive letter (e.g., "C:").
<code>WideExtractFileName (const FileName: WideString): WideString</code>	Returns filename with extension (e.g., "Document.txt").
<code>WideExtractBaseName (const FileName: WideString): WideString</code>	Returns filename without extension or path (e.g., "Document.txt" → "Document", "C:\Folder\Document.txt" → "Document").
<code>WideExtractFileExt (const FileName: WideString): WideString</code>	Returns file extension with dot (e.g., ".txt").
<code>WideChangeFileExt (const FileName, Extension: WideString): WideString</code>	Replaces file extension (e.g., "File.txt" → "File.pdf").
<code>WideStripExtension (const FileName: WideString): WideString</code>	Removes extension while preserving path (e.g., "Document.txt" → "Document", "C:\Folder\Document.txt" → "C:\Folder\Document").
<code>WideExpandFileName (const FileName: WideString): WideString</code>	Converts relative filename to fully qualified path. Does not verify file existence.

Function	Description
<code>WideExtractRelativePath (const BaseName, DestName: WideString): WideString</code>	Creates relative path from <code>BaseName</code> to <code>DestName</code> . For example, going from "C:\Folder\File.txt" to "C:\Documents\Article.pdf" returns ".\Documents\Article.pdf".
<code>WideExtractShortPathName (const FileName: WideString): WideString</code>	Converts path to DOS 8.3 format.
<code>WideIncludeTrailingPathDelimiter (const S: WideString): WideString</code>	Ensures path ends with path delimiter ("\").
<code>WideExcludeTrailingPathDelimiter (const S: WideString): WideString</code>	Ensures path does not end with path delimiter ("\").
<code>WideSameFileName (const FileName1, FileName2: WideString): Boolean</code>	Compares filenames for equality. On Windows, filenames are case-insensitive for comparison purposes but case-preserving when creating new files. <i>Added in v6.7.0.4 Beta.</i>
<code>WideMatchesMask (const FileName, Mask: WideString): Boolean</code>	Checks if filename matches wildcard mask (e.g., "*.txt"). <i>Added in v6.7.0.4 Beta.</i>
<code>WideMatchesMaskList (const FileName, MaskList: WideString): Boolean</code>	Checks if filename matches any wildcard mask in semicolon-separated list (e.g., "*.txt;*.doc"). <i>Added in v6.7.0.4 Beta.</i>

File Read/Write

Function	Description
<code>FileReadFragment (const FileName: WideString; Start, Length: Integer): String</code>	Reads <code>Length</code> number of characters from file starting at position <code>Start</code> (0-based, so 0 is the beginning of the file).
<code>FileReadLines (const FileName: WideString): TAnsiStringArray</code>	Reads all lines from a file and returns as array. <i>Added in v5.74.4 Beta.</i>
<code>FileReadLine (const FileName: WideString; LineNum: Integer): String</code>	Reads a specific line from file by line number (1-based, so 1 is the first line). Note: This function is extremely inefficient and provided only for convenience. Avoid using it repeatedly.
<code>FileCountLines (const FileName: WideString): Integer</code>	Counts the number of lines in a file. Note: This function is extremely inefficient and provided only for convenience. Avoid using it repeatedly.
<code>FileReadContent (const FileName: WideString): String</code>	Returns entire file content as a string.
<code>FileWriteContent (const FileName: WideString; const Content: String)</code>	Writes content to file, overwriting if file exists.
<code>FileAppendContent (const FileName: WideString; const Content: String)</code>	Appends content to end of file, creating file if it doesn't exist.

Function	Description
<code>FileReadText (const FileName: WideString): WideString</code>	Reads text from file with automatic encoding detection. <i>Added in v6.6.0.6 Beta.</i>
<code>FileReadTextLines (const FileName: WideString): TWideStringArray</code>	Reads text lines from file with automatic encoding detection. <i>Added in v6.7.0.1 Beta.</i>

Note: Automatic encoding detection is based on [byte order mark](#) (BOM) and supports UTF-8, UTF-16BE, and UTF-16LE. Files without BOM are interpreted as ANSI encoding.

File Time

Function	Description
<code>FileTimeModified (const FileName: WideString): TDateTime</code>	Returns last modified time of file.
<code>FileTimeCreated (const FileName: WideString): TDateTime</code>	Returns creation time of file.
<code>SetFileTimeCreated (const FileName: WideString; const DateTime: TDateTime): Boolean</code>	Sets creation time of file.
<code>SetFileTimeModified (const FileName: WideString; const DateTime: TDateTime): Boolean</code>	Sets last modified time of file.

Meta Tags Extraction

Function	Description
<code>CalculateMetaTag (const FilePath: WideString; const MetaTagName: String): WideString</code>	Extracts meta tag value from file. <i>Return type changed from String to WideString in v5.74.4 Beta.</i>
<code>CalculateMetaTagFormat (const FilePath: WideString; const MetaTagName, DateTimeFormat: String): WideString</code>	Same as <code>CalculateMetaTag</code> but with custom date and time format instead of application default. <i>Added in v5.74.4 Beta.</i>

Regular Expressions

Function	Description
<code>IsMatchingRegEx (const Input, Pattern: WideString; const CaseSensitive: Boolean): Boolean</code>	Checks if input matches a regular expression pattern. <i>Added in v5.74.4 Beta.</i>

Function	Description
<code>ReplaceRegex (const Input, Find, Replace: WideString; const CaseSensitive, UseSubstitution: Boolean): WideString</code>	Finds and replaces all regex pattern matches. Works like the Regular Expressions rule . <code>Input</code> — Original subject text <code>Find</code> — Search pattern <code>Replace</code> — Replacement pattern <code>CaseSensitive</code> — Search in case-sensitive mode <code>UseSubstitution</code> — Substitute backreferences in replacement pattern
<code>FindRegex (const Input, Find: WideString; const CaseSensitive: Boolean; out Positions: TIntegerArray; out Matches: TWideStringArray): Integer</code>	Finds all matches and positions of a regex pattern. Returns the number of matches found. <code>Input</code> — Original subject text <code>Find</code> — Search pattern <code>CaseSensitive</code> — Search in case-sensitive mode <code>Positions</code> — Receives positions where matches were found <code>Matches</code> — Receives the matched strings <i>Added in v7.3.0.4 Beta.</i>
<code>MatchesRegex (const Input, Find: WideString; const CaseSensitive: Boolean): TWideStringArray</code>	Returns array of full regex matches (not sub-patterns). The function matches the entire expression.
<code>SubMatchesRegex (const Input, Find: WideString; const CaseSensitive: Boolean): TWideStringArray</code>	Returns array of sub-expression matches for the first full match only. This can be combined with <code>MatchesRegex</code> to find all global matches, then parse each match through <code>SubMatchesRegex</code> to extract individual sub-expression matches.

Example matches for `MatchesRegex` and `SubMatchesRegex` :

Input	Pattern	MatchesRegex	SubMatchesRegex
A1_B2_C3	<code>[A-Z]\d</code>	A1, B2, C3	(empty)
A1_B2_C3	<code>(([A-Z])(\d))</code>	A1, B2, C3	A, 1

Process Execution

Function	Description
----------	-------------

<code>ShellOpenFile (const FileName: WideString): Boolean</code>	Opens a file with its associated application. Works like "Start > Run". The parameter can be a file path, URL, or any registered protocol. For example, you can open a Word document or web page and the associated application will launch. Warning: This function evaluates the command and may produce unexpected results. For example, <code>ShellOpenFile('notepad')</code> could run Windows Notepad from <code>C:\Windows\notepad.exe</code>, a different <code>notepad.exe</code> found in the PATH environment variable, or even open a folder named "notepad" in the current directory.
<code>ExecuteProgram (const Command: String; WaitForProgram: Boolean): Cardinal</code>	Executes a command. The <code>WaitForProgram</code> parameter specifies whether to wait for the program to finish before continuing. Returns exit code.
<code>ExecuteProgramShow (const Command: String; WaitForProgram: Boolean; ShowWindowFlag: Word): Cardinal</code>	Executes a command with control over window visibility. Returns exit code. <code>Command</code> — Command to execute <code>WaitForProgram</code> — Wait for program to finish before continuing <code>ShowWindowFlag</code> — Window display mode: 0 = Hide window 1 = Normal (activate and display) 2 = Minimized (activate) 3 = Maximized (activate) 4 = Show without activating 7 = Minimized without activating - More information: ShowWindow API function <i>Added in v5.75.3 Beta.</i>
<code>ExecConsoleApp (const CommandLine: String; out Output: String): Cardinal</code>	Executes a console application and captures its standard output in the <code>Output</code> variable. Returns the exit code. This function should be used only for console applications. Console applications use OEM encoding by default unless they change their own output code page. Use <code>OemToAnsi</code> or <code>OemToWide</code> functions to convert the output if needed. <i>Prior to v6.6.0.2 Beta, OEM to ANSI conversion was applied automatically. Since v6.6.0.2 Beta, output is returned unmodified to prevent corruption of binary or non-OEM encoded data.</i>

Dialogs

Function	Description
<code>ShowMessage (const Msg: String)</code>	Shows message dialog with OK button.
<code>WideShowMessage (const Msg: WideString)</code>	Same as <code>ShowMessage</code> but with Unicode text support.

Function	Description
<code>DialogYesNo (const Msg: String): Boolean</code>	Shows Yes/No dialog. Returns <code>True</code> if user clicks Yes.
<code>WideDialogYesNo (const Msg: WideString): Boolean</code>	Same as <code>DialogYesNo</code> but with Unicode text support.
<code>InputDialog (const ACaption, APrompt, ADefault: String): String</code>	Shows input dialog with text box. If user clicks OK, returns the entered value, otherwise returns the default value.
<code>InputQuery (const ACaption, APrompt: String; var Value: String): Boolean</code>	Shows input dialog. Returns <code>True</code> if user clicks OK. The entered value is stored in the <code>Value</code> parameter.
<code>WideInputDialog (const ACaption, APrompt, ADefault: WideString): WideString</code>	Same as <code>InputDialog</code> but with Unicode text support.
<code>WideInputQuery (const ACaption, APrompt: WideString; var Value: WideString): Boolean</code>	Same as <code>InputQuery</code> but with Unicode text support.

Application

Function	Description
<code>GetApplicationPath: WideString</code>	Returns full path to ReNamer executable (e.g., "C:\Program Files\ReNamer\ReNamer.exe").
<code>GetApplicationParams: TWideStringArray</code>	Returns array of command line parameters passed to the application at launch.
<code>GetCurrentFileIndex: Integer</code>	Returns index of current file being processed (ranges from 1 to <code>GetTotalNumberOfFiles</code>).
<code>GetTotalNumberOfFiles: Integer</code>	Returns total number of files in the list.
<code>GetCurrentMarkedFileIndex: Integer</code>	Returns index of current file among marked files only (ranges from 1 to <code>GetTotalNumberOfMarkedFiles</code>).
<code>GetTotalNumberOfMarkedFiles: Integer</code>	Returns total number of marked files.
<code>GetAllFiles: TWideStringArray</code>	Returns file paths of all files in the list. <i>Added in v5.74.2 Beta.</i>
<code>GetMarkedFiles: TWideStringArray</code>	Returns file paths of all marked files. <i>Added in v5.74.2 Beta.</i>

Global Variables

Global variables allow efficient storage and exchange of information between script executions.

Variables persist until manually cleared or application terminates. To clear variables at each preview, use [script initialization](#).

Added in v7.4.0.2 Beta.

Function	Description
<code>SetGlobalVar (const Name: String; Value: Variant)</code>	Sets a global variable.
<code>GetGlobalVar (const Name: String): Variant</code>	Gets a global variable. Returns <code>Unassigned</code> value if variable doesn't exist.
<code>GetGlobalVarDef (const Name: String; Default: Variant): Variant</code>	Gets a global variable. Returns <code>Default</code> value if variable doesn't exist.
<code>HasGlobalVar (const Name: String): Boolean</code>	Checks if a global variable exists.
<code>GetGlobalVarCount: Integer</code>	Returns total number of global variables.
<code>GetGlobalVarNames: TAnsiStringArray</code>	Returns names of all global variables.
<code>ClearGlobalVar (const Name: String)</code>	Clears a specific global variable.
<code>ClearGlobalVars</code>	Clears all global variables.

System

Function	Description
<code>WideGetCurrentDir: WideString</code>	Returns current working directory.
<code>WideSetCurrentDir (const Dir: WideString): Boolean</code>	Sets current working directory.
<code>WideGetTempPath: WideString</code>	Returns system temporary directory. Returns empty string if path cannot be determined.
<code>WideGetEnvironmentVar (const VarName: WideString): WideString</code>	Returns environment variable value by name (e.g., "USERNAME", "COMPUTERNAME").
<code>GetClipboardText: WideString</code>	Returns text content from clipboard.
<code>SetClipboardText (const S: WideString)</code>	Sets text content to clipboard.

Mathematics

Function	Description
<code>Sin (const X: Extended): Extended</code>	Returns the sine of angle <code>X</code> , where <code>X</code> is in radians.
<code>Cos (const X: Extended): Extended</code>	Returns the cosine of angle <code>X</code> , where <code>X</code> is in radians.
<code>Sqrt (const X: Extended): Extended</code>	Returns the square root of <code>X</code> . <code>X</code> must be a non-negative value.
<code>Round (const X: Extended): Integer</code>	Rounds <code>X</code> to the nearest integer, with ".5" values rounded to the nearest even number ("banker's rounding").

Function	Description
<code>Trunc (const X: Extended): Integer</code>	Returns the integer part of <code>X</code> , truncating towards zero without rounding.
<code>Int (const X: Extended): Extended</code>	Returns the integer part of <code>X</code> as a floating-point value.
<code>Abs (const X: Extended): Extended</code>	Returns the absolute (non-negative) value of <code>X</code> .
<code>Pi: Extended</code>	Returns the mathematical constant π (3.14159265358979...).
<code>DivMod (Dividend: Integer; Divisor: Word; var Result, Remainder: Word)</code>	Performs integer division and returns both quotient and remainder in a single operation. <code>Dividend</code> — The value being divided <code>Divisor</code> — The value to divide by <code>Result</code> — Receives the quotient (result of division) <code>Remainder</code> — Receives the remainder (the difference between <code>Result * Divisor</code> and <code>Dividend</code>)
<code>MinInt (const A, B: Integer): Integer</code>	Returns smaller of two Integer values. <i>Added in v6.2.0.8 Beta.</i>
<code>MinInt64 (const A, B: Int64): Int64</code>	Returns smaller of two Int64 values. <i>Added in v6.2.0.8 Beta.</i>
<code>MinFloat (const A, B: Extended): Extended</code>	Returns smaller of two Extended values. <i>Added in v6.2.0.8 Beta.</i>
<code>MaxInt (const A, B: Integer): Integer</code>	Returns larger of two Integer values. <i>Added in v6.2.0.8 Beta.</i>
<code>MaxInt64 (const A, B: Int64): Int64</code>	Returns larger of two Int64 values. <i>Added in v6.2.0.8 Beta.</i>
<code>MaxFloat (const A, B: Extended): Extended</code>	Returns larger of two Extended values. <i>Added in v6.2.0.8 Beta.</i>
<code>Randomize</code>	Initializes random number generator. Should only be called once per session.
<code>RandomRange (const AFrom, ATo: Integer): Integer</code>	Returns random integer from <code>AFrom</code> (inclusive) to <code>ATo</code> (exclusive). For example, <code>RandomRange(1, 10)</code> returns a value between 1 and 9, inclusive.
<code>RandomFloat: Extended</code>	Returns random floating point value from 0.0 (inclusive) to 1.0 (exclusive). <i>Added in v6.2.0.8 Beta.</i>
<code>RandomBoolean: Boolean</code>	Returns random boolean value (<code>True</code> or <code>False</code>). <i>Added in v6.2.0.8 Beta.</i>
<code>RandomString (Len: Integer; const Chars: String): String</code>	Generates random string of length <code>Len</code> using characters from <code>Chars</code> . <i>Added in v6.7.0.4 Beta.</i>

Miscellaneous

Function	Description
<code>Error (const Message: String)</code>	Terminates script with error message. <i>Added in v7.2.0.7 Beta.</i>
<code>Sleep (Milliseconds: Cardinal)</code>	Pauses script execution for specified milliseconds.
<code>HexEncode (const S: String): String</code>	Encodes string into its hexadecimal representation, where each byte is represented as a two-character hex value. <i>Added in v7.8.0.4 Beta.</i>
<code>HexDecode (const S: String): String</code>	Decodes a hexadecimal-encoded string back into its original binary representation. <i>Added in v7.8.0.4 Beta.</i>
<code>Base64Encode (const S: String): String</code>	Encodes string to Base64 format. Useful for encoding binary data to minimize the likelihood of corruption when transmitted through systems like email or web.
<code>Base64Decode (const S: String): String</code>	Decodes Base64 encoded string.
<code>URLEncode (const Str: WideString; UsePlusAsSpace: Boolean): String</code>	Encodes string for URL use. The input string is first encoded to UTF-8, then all characters except digits and Latin letters are encoded as <code>%XX</code> hex sequences. If <code>UsePlusAsSpace</code> is <code>True</code> , spaces are encoded as plus signs ("+") instead of "%20" (for compatibility with some decoders like PHP). <i>Added in v5.74.4 Beta.</i>
<code>URLDecode (const Str: String; UsePlusAsSpace: Boolean): WideString</code>	Decodes URL encoded string. All <code>%XX</code> hex sequences are decoded, then the entire string is decoded from UTF-8. If <code>UsePlusAsSpace</code> is <code>True</code> , plus signs ("+") are interpreted as spaces (for compatibility with some encoders like PHP). <i>Added in v5.74.4 Beta. UsePlusAsSpace parameter added in v6.7.0.1 Beta.</i>
<code>GetTickCount: Cardinal</code>	Returns milliseconds since system start (resets after ~49.7 days). Limited precision.
<code>SizeOf (X): Integer</code>	Returns number of bytes used to represent a variable or type. Pass a variable to get its size, or pass a type identifier to get the size of that type.

Pascal Script: Examples

This page provides practical examples demonstrating how to use the [Pascal Script rule](#) in ReNamer. Each example is self-contained and focuses on a specific task or concept. For function signatures and type details, refer to the [Functions](#) and [Types](#) reference articles.

Modifying FileName

The `FileName` variable is a `WideString` representing the new name for the current file, including the extension. Modifying it is the fundamental operation of any script. Assigning an empty string or raising an error causes the file to be skipped during renaming.

To add a prefix to every filename:

```
begin
    FileName := 'MyPrefix_' + FileName;
end.
```

To add a suffix before the extension, split the base name and extension apart first:

```
var
    BaseName, Ext: WideString;
begin
    BaseName := WideExtractBaseName(FileName);
    Ext := WideExtractFileExt(FileName);
    FileName := BaseName + '_backup' + Ext;
end.
```

`WideExtractBaseName` returns the filename without path or extension. `WideExtractFileExt` returns the extension including the leading dot (e.g. `.txt`). This split-and-rebuild pattern appears throughout many scripts whenever the extension must be preserved while transforming only the base name.

Accessing FilePath and path components

The `FilePath` constant holds the original full path to the current file (e.g. `C:\Music\Artist\Album\Track.flac`). It is read-only, but is invaluable for reading file content, extracting metadata, and deriving folder-based names.

To prefix a filename with its immediate parent folder name:

```
begin
    FileName := WideExtractFileName(WideExtractFileDir(FilePath)) + ' - ' + FileName;
end.
```

`WideExtractFileDir` returns the directory without a trailing backslash, and `WideExtractFileName` extracts just the last component — the immediate parent folder name.

For a deeper hierarchy, use `WideSplitString` to break the path into individual folder components:

```
var
    Parts: TWideStringArray;
begin
    Parts := WideSplitString(WideExtractFileDir(FilePath), '\');
    // Parts[Length(Parts) - 1] is the immediate parent folder
    // Parts[Length(Parts) - 2] is one level above, and so on
    if Length(Parts) >= 2 then
        FileName := Parts[Length(Parts) - 2] + ' - ' + Parts[Length(Parts) - 1] + ' - ' +
        FileName;
    end.
```

Note: `TWideStringArray` is 0-based, so the last element is at index `Length(Parts) - 1`.

Case conversion

ReNamer provides several functions for Unicode-aware case conversion, all accepting and returning `WideString`.

To convert the entire filename to uppercase:

```
begin
    FileName := WideUpperCase(FileName);
end.
```

To convert only the base name while preserving the extension:

```

var
  BaseName, Ext: WideString;
begin
  BaseName := WideExtractBaseName(FileName);
  Ext := WideExtractFileExt(FileName);
  FileName := WideUpperCase(BaseName) + Ext;
end.

```

The same pattern applies to all case functions. Given the input "the quick BROWN fox":

Function	Result
WideUpperCase	THE QUICK BROWN FOX
WideLowerCase	the quick brown fox
WideCaseCapitalize	The Quick Brown Fox
WideCaseSentence	The quick brown fox
WideCaseInvert	THE QUICK brown FOX

Partial case change

To apply case changes to specific characters only, iterate character by character using `WideCharUpper` and `WideCharLower`. The built-in case functions operate on the entire string; this approach is needed when the logic depends on position or neighbouring characters.

The example below uppercases the first letter of each word while lowercasing all other letters, but deliberately leaves digits and punctuation untouched:

```

var
  BaseName, Ext, NewName: WideString;
  I: Integer;
  AtWordStart: Boolean;
begin
  BaseName := WideExtractBaseName(FileName);
  Ext := WideExtractFileExt(FileName);
  NewName := '';
  AtWordStart := True;

  for I := 1 to WideLength(BaseName) do
  begin
    if IsWideCharSpace(BaseName[I]) or IsWideCharPunct(BaseName[I]) then
    begin

```

```

    NewName := NewName + BaseName[I];
    AtWordStart := True;
end
else if IsWideCharAlpha(BaseName[I]) then
begin
    if AtWordStart then
        NewName := NewName + WideCharUpper(BaseName[I])
    else
        NewName := NewName + WideCharLower(BaseName[I]);
        AtWordStart := False;
    end
end
else
begin
    NewName := NewName + BaseName[I]; // digits and other chars pass through unchanged
    AtWordStart := False;
end;
end;

FileName := NewName + Ext;
end.

```

Splitting and rearranging parts

`WideSplitString` and `WideJoinStrings` are the primary tools for decomposing a filename around a delimiter and reassembling it in a different order.

Consider files named `"Artist - Title.mp3"` where the goal is to swap artist and title to produce `"Title - Artist.mp3"`:

```

var
    Parts: TWideStringArray;
    BaseName, Ext: WideString;
begin
    BaseName := WideExtractBaseName(FileName);
    Ext := WideExtractFileExt(FileName);

    Parts := WideSplitString(BaseName, ' - ');

    if Length(Parts) = 2 then

```



```
FileName := WideTrim(Parts[1]) + ' - ' + WideTrim(Parts[0]) + Ext;  
end.
```

The delimiter passed to `WideSplitString` can be a multi-character string. `WideTrim` removes any stray whitespace from each part. The `Length(Parts) = 2` guard ensures the script acts only when the expected structure is present.

`WideJoinStrings` reassembles an array with a delimiter inserted between each item:

```
var  
  Parts: TWideStringArray;  
  BaseName, Ext: WideString;  
begin  
  BaseName := WideExtractBaseName(FileName);  
  Ext := WideExtractFileExt(FileName);  
  
  Parts := WideSplitString(BaseName, '_');  
  FileName := WideJoinStrings(Parts, ' ') + Ext;  
end.
```

This replaces all underscores in the base name with spaces.

String manipulation

For targeted operations that do not split the whole name, use `WidePos`, `WideCopy`, `WideInsert`, `WideDelete`, and `WideReplaceStr`.

Finding and replacing text

To replace all occurrences of a substring (case-sensitive):

```
begin  
  FileName := WideReplaceStr(FileName, '_', ' ');  
end.
```

For case-insensitive replacement, use `WideReplaceText` instead.

Moving a portion of the filename

To move the first N characters from the front of the base name to the end — for example, relocating a leading year `"2024 Concert Recording"` → `"Concert Recording 2024"`:

```

var
    BaseName, Ext, Prefix: WideString;
begin
    BaseName := WideExtractBaseName(FileName);
    Ext := WideExtractFileExt(FileName);

    Prefix := WideCopy(BaseName, 1, 5);    // "2024 "
    WideDelete(BaseName, 1, 5);
    FileName := WideTrim(BaseName) + ' ' + WideTrim(Prefix) + Ext;
end.

```

`WideCopy` extracts a substring by start position and character count. `WideDelete` removes characters from the string in place. Both use 1-based indexing.

Inserting text at a specific position

`WideInsert` modifies the target string in place, inserting text at the given 1-based position. A practical use is inserting a separator after a numeric prefix — for example, turning `"007Skyfall"` into `"007 - Skyfall"`:

```

var
    BaseName, Ext: WideString;
    I: Integer;
begin
    BaseName := WideExtractBaseName(FileName);
    Ext := WideExtractFileExt(FileName);

    // Find where the leading digits end
    I := 1;
    while (I <= WideLength(BaseName)) and IsWideCharDigit(BaseName[I]) do
        Inc(I);

    // Insert separator only if digits were found and something follows
    if (I > 1) and (I <= WideLength(BaseName)) then
        WideInsert(' - ', BaseName, I);

    FileName := BaseName + Ext;
end.

```

Separating CamelCase words

This example inserts a space before each uppercase letter that follows a lowercase letter, turning CamelCase names like `"TheEverlyBrothers"` into `"The Everly Brothers"`:

```
var
  BaseName, Ext, NewName: WideString;
  I: Integer;
begin
  BaseName := WideExtractBaseName(FileName);
  Ext := WideExtractFileExt(FileName);
  NewName := '';

  for I := 1 to Length(BaseName) do
    begin
      if I > 1 then
        if IsWideCharUpper(BaseName[I]) and IsWideCharLower(BaseName[I - 1]) then
          NewName := NewName + ' ';
        NewName := NewName + BaseName[I];
      end;

      FileName := NewName + Ext;
    end.
end.
```

The condition inserts a space only when the current character is uppercase and the preceding character is lowercase, targeting genuine CamelCase boundaries.

Initializing variables

Script-level variables persist across file iterations within a single preview run. This is intentional — it makes counters and accumulators possible — but it means any variable that should start fresh with each preview must be explicitly reset.

The recommended approach is a dedicated `Initialized` boolean flag. Since boolean variables default to `False`, the very first execution of the script naturally triggers initialisation, regardless of which file in the list is processed first:

```
var
  Initialized: Boolean;
  Counter: Integer;
  Prefix: WideString;
begin
```

```

if not Initialized then
begin
    Counter := 0;
    Prefix := 'Item';
    Initialized := True;
end;

Inc(Counter);
FileName := Prefix + ' ' + IntToStr(Counter) + ' - ' + FileName;
end.

```

An alternative sometimes seen is checking `GetCurrentFileIndex = 1` (or `GetCurrentMarkedFileIndex = 1`). However, this is unreliable: if the first file in the list is unmarked (excluded from renaming), the script skips it and the index for the first executed file will be greater than 1, so the initialisation block never runs. The `Initialized` flag avoids this problem entirely and is generally preferred.

Script-level variables are reset at the start of every Preview operation — the script is compiled fresh each time Preview runs. They do not persist between previews. For data that must outlast a single preview, see the [Persisting data](#) section.

Serializing files

To add an incrementing index to each filename, use `GetCurrentMarkedFileIndex` combined with `FormatFloat` for zero-padded output:

```

begin
    FileName := FormatFloat('000', GetCurrentMarkedFileIndex) + ' - ' + FileName;
end.

```

`GetCurrentMarkedFileIndex` counts only the files actually being processed (marked for renaming), so the index always reflects the position in the renamed set rather than the full file list.

`FormatFloat('000', ...)` pads the number to at least three digits (e.g. `001`, `012`, `123`). Adjust the number of zeroes to control the minimum width.

To start from a custom index (e.g. 100 rather than 1) and use Roman numerals instead of decimal numbers:

```

var
    Index: Integer;
begin

```

```
Index := GetCurrentMarkedFileIndex + 99; // starts from 100
FileName := IntToRoman(Index) + ' - ' + FileName;
end.
```

Generating random names

Generating a random name can be useful for anonymising files, creating unique temporary identifiers, or testing.

The `RandomString` function builds a string of a given length by picking characters at random from a supplied alphabet. `RandomRange(6, 13)` produces random lengths between 6 and 12 characters. `Randomize` seeds the random number generator and should be called only once per session.

```
const
    Alphabet = 'abcdefghijklmnopqrstuvwxyz0123456789';

var
    Initialized: Boolean;
begin
    if not Initialized then
    begin
        Randomize;
        Initialized := True;
    end;

    FileName := RandomString(RandomRange(6, 13), Alphabet)
                + WideExtractFileExt(FileName);
end.
```

Indexing per folder

When files from multiple folders are loaded, you may want the index to reset for each folder. The script below tracks the active folder and resets a local counter whenever it changes:

```
var
    Initialized: Boolean;
    CurrentFolder: WideString;
    FolderIndex: Integer;
```

```

begin
  if not Initialized then
  begin
    CurrentFolder := '';
    FolderIndex := 0;
    Initialized := True;
  end;

  if WideExtractFileDir(FilePath) <> CurrentFolder then
  begin
    CurrentFolder := WideExtractFileDir(FilePath);
    FolderIndex := 0;
  end;

  Inc(FolderIndex);
  FileName := FormatFloat('00', FolderIndex) + ' - ' + FileName;
end.

```

The `Initialized` block resets the tracking variables at the start of each preview, ensuring the counter restarts correctly every time.

Serializing duplicates

When multiple files would otherwise produce the same new name, a counter suffix can disambiguate them, producing `"Name.ext"`, `"Name (2).ext"`, `"Name (3).ext"`, and so on.

The script maintains an array of names already assigned in the current preview run and checks each candidate against it:

```

var
  Initialized: Boolean;
  SeenNames: TWideStringArray;
  BaseName, Ext, Candidate: WideString;
  Counter, I: Integer;
  Found: Boolean;
begin
  if not Initialized then
  begin
    SetLength(SeenNames, 0);
    Initialized := True;

```

```

end;

BaseName := WideExtractBaseName(FileName);
Ext := WideExtractFileExt(FileName);
Candidate := FileName;
Counter := 1;

repeat
    Found := WideFindStringArray(Candidate, SeenNames, False) > 0;
    if Found then
        begin
            Inc(Counter);
            Candidate := BaseName + ' (' + IntToStr(Counter) + ')' + Ext;
        end;
    until not Found;

    WideAppendStringArray(SeenNames, Candidate);
    FileName := Candidate;
end.

```

`WideFindStringArray` with the last argument `False` performs a case-insensitive search of previously seen filenames, which is appropriate since Windows filenames are case-insensitive.

`WideAppendStringArray` adds the finalised name to the list before moving on to the next file.

Controlling script flow

Use `Exit` to skip renaming for the current file without an error — `FileName` is left unchanged and the file passes through untouched:

```

begin
    if WideExtractFileExt(FileName) = '.tmp' then
        Exit;

        FileName := 'Processed_' + FileName;
    end.

```

To stop processing all subsequent files once a condition is met, use a persistent boolean flag together with `Exit`. The flag persists across iterations within the same preview run:

```

var
  StopProcessing: Boolean;
begin
  if StopProcessing then
    Exit;

  if WideDialogYesNo('Rename "' + FileName + '"?') then
    FileName := 'Processed_' + FileName
  else
    StopProcessing := True;
end.

```

The first time the user clicks No, `StopProcessing` is set to `True` and `Exit` is called for every subsequent file, leaving them all unchanged.

Renaming by file date and time

To rename a file using its last-modified timestamp, read it with `FileTimeModified` and format it with `FormatDateTime`:

```

var
  FileTime: TDateTime;
begin
  FileTime := FileTimeModified(FilePath);
  FileName := FormatDateTime('yyyy-mm-dd hh-nn-ss', FileTime) + WideExtractFileExt(FileName);
end.

```

`FormatDateTime` accepts a format string following the [Date and Time format](#) conventions. Note that minutes use `nn` to distinguish them from months (`mm`). The example above produces names like `2024-06-15 14-32-07.jpg`.

To use the creation time instead, replace `FileTimeModified` with `FileTimeCreated`.

Adjusting dates embedded in filenames

Cameras and recording devices sometimes embed timestamps in filenames (e.g. `2024-06-15 14-32-07.jpg`). When the device clock was set to the wrong timezone, or when DST was not accounted for, every file in a batch needs the same time offset applied.

`TryScanDateTime` parses the embedded date according to a format pattern, `IncHour` shifts it by the required amount — handling all day, month, and year rollovers automatically — and `FormatDateTime` writes it back:

```
const
    InputFormat  = 'yyyy-mm-dd hh-nn-ss';
    OutputFormat = 'yyyy-mm-dd hh-nn-ss';
    HoursToAdd   = -3; // use a negative value to subtract

var
    BaseName, Ext: WideString;
    DateTime: TDateTime;
begin
    BaseName := WideExtractBaseName(FileName);
    Ext := WideExtractFileExt(FileName);

    if TryScanDateTime(InputFormat, BaseName, DateTime) then
    begin
        DateTime := IncHour(DateTime, HoursToAdd);
        FileName := FormatDateTime(OutputFormat, DateTime) + Ext;
    end;
end.
```

`TryScanDateTime` returns `False` without raising an error if the filename does not match the expected pattern, so files with non-conforming names are left unchanged.

The input and output format strings are kept as separate constants, making it straightforward to reformat while adjusting — for example, changing `OutputFormat` to `'yyyy-mm-dd hh.nn.ss'` converts the time separators from dashes to dots in the same pass. See the [Date and Time format](#) reference for format specifiers.

Reading file content

Scripts can read file content directly, which is useful for assigning names based on data inside the file itself.

Reading a fixed fragment

`FileReadFragment` reads a specific number of bytes from a given offset (0-based). This is efficient for sampling file headers without loading the entire file:

```

var
  Header: String;
begin
  Header := FileReadFragment(FilePath, 0, 4);
  if Copy(Header, 1, 2) = 'PK' then
    FileName := '[ZIP] ' + FileName;
end.

```

Reading the full content

`FileReadText` returns the entire file as a `WideString` and automatically detects encoding via byte order mark (BOM), supporting UTF-8 and UTF-16. Files without BOM are treated as ANSI.

A practical use is extracting a value embedded inside the file. The example below renames HTML files using the text content of their `<title>` tag:

```

var
  Content, Title: WideString;
  PosStart, PosEnd: Integer;
begin
  if not WideSameText(WideExtractFileExt(FileName), '.html') then
    Exit;

  Content := FileReadText(FilePath);

  PosStart := WideTextPos('<title>', Content);
  PosEnd := WideTextPos('</title>', Content);

  if (PosStart > 0) and (PosEnd > PosStart) then
    begin
      PosStart := PosStart + 7; // skip past '<title>'
      Title := WideTrim(WideCopy(Content, PosStart, PosEnd - PosStart));
      if Title <> '' then
        FileName := Title + '.html';
    end;
end.

```

`WideTextPos` performs a case-insensitive search, which is appropriate for HTML tags. `WideCopy` extracts the text between the two tag positions, and `WideTrim` cleans up any surrounding whitespace.

Renaming from a list

A plain text file with one new name per line can drive batch renaming. Load the list once on first run, then advance a local counter on each subsequent execution:

```
var
  Initialized: Boolean;
  Names: TWideStringArray;
  Counter: Integer;
begin
  if not Initialized then
  begin
    Names := FileReadTextLines(WideGetEnvironmentVar('USERPROFILE') + '\Desktop\names.txt');
    Counter := 0;
    Initialized := True;
  end;

  if Counter < Length(Names) then
  begin
    FileName := WideTrim(Names[Counter]) + WideExtractFileExt(FileName);
    Inc(Counter);
  end;
end.
```

`FileReadTextLines` handles encoding detection automatically. `Counter` starts at 0 to match the zero-based array index and is incremented after each use. Files beyond the end of the list are left unchanged.

Converting file content encoding

Scripts are not limited to renaming — they can also modify the content of the files being processed. A practical use is bulk-converting text files from the [Windows system code page](#) (ANSI) to UTF-8, which is required when moving content to systems or editors that expect UTF-8.

Warning: This script writes directly to the original file on disk during the Preview stage. The change is immediate and irreversible. Always keep backups before running it on a large batch, and verify the result on a single file first.

The script assembles the UTF-8 BOM (`EF BB BF`) from its hex byte values, checks whether the file already starts with it, and skips the file if so — making the script safe to run more than once on the same set of files:

```
var
  Content, UTF8BOM: String;
```

```

begin
    Content := FileReadContent(FilePath);

    // Define the UTF-8 BOM
    UTF8BOM := Chr($EF) + Chr($BB) + Chr($BF);

    // Skip files that are already UTF-8 encoded (BOM present)
    if Copy(Content, 1, Length(UTF8BOM)) = UTF8BOM then
        Exit;

    // Convert content encoding and add UTF-8 BOM
    Content := UTF8BOM + WinCPToUTF8(Content);

    FileWriteContent(FilePath, Content);
end.

```

`FileReadContent` reads the raw bytes of the file as a `String`. `WinCPToUTF8` re-encodes the string from the active Windows system code page to UTF-8. The BOM is then prepended before writing the result back with `FileWriteContent`, which overwrites the original file.

Note that `FileName` is not modified in this script. The script's sole effect is on the file content. This is unusual behaviour for a Pascal Script rule and worth keeping in mind when combining it with other rules in the same preset.

Interactive dialogs

Dialog functions pause the preview run and prompt the user, making it possible to inject values or confirm decisions at runtime. Because scripts run for each file in sequence, dialogs should generally be shown only once per preview — use the `Initialized` flag pattern for this — unless per-file confirmation is specifically intended.

Displaying messages

`WideShowMessage` is useful during development for inspecting intermediate values:

```

begin
    WideShowMessage('File: ' + FileName + #13#10 + 'Path: ' + FilePath);
end.

```

`#13#10` is a carriage return and line feed, producing a newline inside the message box.

Yes/No confirmation per file

`WideDialogYesNo` pauses for user input and returns `True` if the user clicks Yes:

```
begin
  if WideDialogYesNo('Apply uppercase to:' + #13#10 + FileName) then
    FileName := WideUpperCase(FileName);
  end.
```

Collecting input once per preview

`WideInputDialog` shows a text input dialog and returns the entered text which will be used as a prefix for filenames:

```
var
  Initialized: Boolean;
  Prefix: WideString;

begin
  if not Initialized then
    begin
      Prefix := WideInputDialog('Prefix', 'Enter a prefix to add to all filenames:', '');
      Initialized := True;
    end;

  FileName := Prefix + FileName;
end.
```

The dialog appears once, the user types a prefix, and that value is reused for every file in the list.

Regular expressions

The built-in regex functions bring pattern matching directly into scripts, useful for transformations that are too complex for a simple search-and-replace.

Pattern-based replacement

`ReplaceRegex` works like the Regular Expressions rule but within a script, allowing the result to be further processed:

```
begin
  // Collapse any sequence of whitespace into a single space
  FileName := ReplaceRegex(FileName, '\s+', ' ', False, False);
```

```

    FileName := WideTrim(FileName);
end.

```

With `UseSubstitution` set to `True`, backreferences can be used in the replacement pattern. The example below moves a trailing four-digit year to the front:

```

begin
    // "Concert Recording 2024.mp4" -> "2024 Concert Recording.mp4"
    FileName := ReplaceRegEx(FileName, '^(.*)\s(\d{4})(\.[^.]+)$', '$2 $1$3', True, True);
end.

```

Extracting matches

`MatchesRegEx` returns an array of all substrings in the input that match the given pattern:

```

var
    Matches: TWideStringArray;
    Ext: WideString;
begin
    Ext := WideExtractFileExt(FileName);
    Matches := MatchesRegEx(FileName, '\d+', True);

    if Length(Matches) > 0 then
        FileName := 'Track_' + Matches[0] + Ext;
end.

```

Applying case conversion to matches

`FindRegEx` returns the matches and their positions, allowing programmatic string manipulation and substitution:

```

var
    Positions: TIntegerArray;
    Matches: TWideStringArray;
    I, CountMatches: Integer;
begin
    // Capitalise each word in the base name
    CountMatches := FindRegEx(WideExtractBaseName(FileName), '[a-zA-Z]+', False, Positions,
Matches);
    for I := 0 to CountMatches - 1 do
        begin

```

```

Delete(FileName, Positions[I], Length(Matches[I]));
Insert(WideCaseCapitalize(Matches[I]), FileName, Positions[I]);
end;
end.

```

Persisting data

Script-level variables reset at the start of every Preview. Two mechanisms are available when data must outlast a single preview run.

Global Variables

Global Variables (added in v7.4.0.2) persist across previews for the entire application session — until manually cleared or the application is closed. They are identified by name and can store any value type.

```

var
  Initialized: Boolean;
  RunCount: Integer;
begin
  if not Initialized then
    begin
      if HasGlobalVar('RunCount') then
        RunCount := GetGlobalVar('RunCount')
      else
        RunCount := 0;
        Initialized := True;
      end;

      Inc(RunCount);
      SetGlobalVar('RunCount', RunCount);

      FileName := FormatFloat('0000', RunCount) + '_' + FileName;
    end.
  end.

```

To reset all global variables at the start of each preview, clear them inside the `Initialized` block:

```

begin
  if not Initialized then
    begin

```

```

    ClearGlobalVars;
    Initialized := True;
end;

// ... rest of script
end.

```

File-based persistence

For data that must survive application restarts, write it to a file. Using the system temporary folder keeps the file out of the way:

```

var
    Initialized: Boolean;
    StorageFile: WideString;
    Counter: Integer;
begin
    if not Initialized then
    begin
        StorageFile := WideGetTempPath + 'renamer_counter.txt';
        if WideFileExists(StorageFile) then
            Counter := StrToIntDef(FileReadContent(StorageFile), 0)
        else
            Counter := 0;
        Initialized := True;
    end;

    Inc(Counter);
    FileWriteContent(StorageFile, IntToStr(Counter));

    FileName := FormatFloat('0000', Counter) + '_' + FileName;
end.

```

Delete the storage file manually (or from a cleanup script) to reset the counter. `StrToIntDef` returns the default value `0` if the file contains unexpected content, preventing a script error.

Using meta tags

The `CalculateMetaTag` function extracts metadata from a file using ReNamer's built-in [Meta Tags](#) engine, covering EXIF, ID3, document properties, and more. The full list of available tag names is

on the Meta Tags reference page.

A common use is renaming photos by their EXIF capture date:

```
begin
  FileName := CalculateMetaTag(FilePath, 'EXIF_Date') + WideExtractFileExt(FileName);
end.
```

The date format used here follows the application's default Date and Time format setting. To use a specific format regardless of that setting, use `CalculateMetaTagFormat`:

```
begin
  FileName := CalculateMetaTagFormat(FilePath, 'EXIF_Date', 'yyyy-mm-dd') +
WideExtractFileExt(FileName);
end.
```

It is good practice to guard against an empty result, which occurs when the tag is not present in the file:

```
var
  Tag: WideString;
begin
  Tag := CalculateMetaTag(FilePath, 'ID3_Title');
  if Tag <> '' then
    FileName := Tag + WideExtractFileExt(FileName);
end.
```

User-defined functions

Scripts can define their own functions and procedures (UDFs), which is useful for encapsulating reusable logic and keeping the main code block readable. Declarations must appear before the main code block.

The example below defines a helper that strips characters forbidden in Windows filenames:

```
function StripInvalidChars(const S: WideString): WideString;
var
  I: Integer;
  C: WideChar;
begin
  Result := '';
```

```

for I := 1 to WideLength(S) do
begin
    C := S[I];
    if WidePos(C, '\/:*?"<>|') = 0 then
        Result := Result + C;
    end;
end;

begin
    FileName := StripInvalidChars(FileName);
end.

```

Note that `Result` here is the standard Pascal return-value identifier used inside a function body — this is its correct and intended use. User-defined functions and procedures follow standard Pascal syntax; they can accept parameters, call other UDFs, and use all built-in functions.

Importing external functions

Scripts can call functions from any external DLL by declaring them with the `external` directive followed by an import declaration in one of these formats:

- `function FunctionName(Parameters): ReturnType; external 'FunctionName@Library.dll CallingConvention';`
- `procedure FunctionName(Parameters); external 'FunctionName@Library.dll CallingConvention';`

A minimal example using a Windows API function:

```

function GetTickCount: Cardinal;
    external 'GetTickCount@kernel32.dll stdcall';

begin
    FileName := IntToStr(GetTickCount) + '_' + FileName;
end.

```

The example below retrieves the current Windows username:

```

function GetUserNameA(lpBuffer: PChar; var nSize: Cardinal): Boolean;
    external 'GetUserNameA@advapi32.dll stdcall';

var

```

```

Size: Cardinal;
UserName: String;
begin
  Size := 200;
  SetLength(UserName, Size);
  if GetUserNameA(PChar(UserName), Size) then
    begin
      SetLength(UserName, Size - 1); // Size includes trailing null terminator
      FileName := UserName + '_' + FileName;
    end;
end.

```

The calling convention is `stdcall` for most Windows API functions and `cdecl` for many C library functions. An incorrect declaration or a missing DLL will cause a script error. Test new imports carefully.

For integrations with 3rd party command-line tools such as ExifTool or MediaInfo, `ExecConsoleApp` is often more practical than DLL import — see the [Library](#) article for those scripts.

Conditional processing by file type

Scripts process every file in the list, but sometimes logic should apply only to a subset.

`WideMatchesMask` tests a filename against a single wildcard mask. `WideMatchesMaskList` tests against a semicolon-separated list.

To apply a transformation only to certain image files:

```

begin
  if not WideMatchesMaskList(FileName, '*.jpg;*.jpeg') then
    Exit;

  FileName := CalculateMetaTag(FilePath, 'EXIF_Date') + WideExtractFileExt(FileName);
end.

```

The `Exit` at the top is an efficient way to skip non-matching files early, leaving `FileName` unchanged.

An alternative pattern can be applied whenever different rules should apply to different file types within a single script:

```

begin
  if WideMatchesMask(FileName, '*.mp3') then
    FileName := CalculateMetaTag(FilePath, 'MP3_Title') + '.mp3'
  else if WideMatchesMask(FileName, '*.jpg') then
    FileName := CalculateMetaTag(FilePath, 'EXIF_Date') + '.jpg';
  end.

```

URL-decoding filenames

Files downloaded from the web sometimes retain URL encoding in their names, producing strings like "My%20Document%20%282024%29.pdf". The built-in `URLDecode` function converts them back to readable form:

```

var
  BaseName, Ext: WideString;
begin
  BaseName := WideExtractBaseName(FileName);
  Ext := WideExtractFileExt(FileName);
  FileName := URLDecode(BaseName, False) + Ext;
end.

```

The second parameter, `UsePlusAsSpace`, controls whether `+` signs are treated as spaces. Set it to `False` for standard URL encoding (where spaces are `%20`), or `True` for HTML form encoding (where spaces are `+`).

The extension is decoded separately only when needed, since extensions are typically not URL-encoded.

Pascal Script: Library

This page provides curated, ready-to-use scripts for common renaming tasks and integrations with third-party tools. Each script can be copied directly into the [Pascal Script rule](#) and adapted by adjusting the constants at the top.

Additionally, the [User Forum](#) is a valuable source of community-contributed scripts covering a wider range of use cases — though as with any user-contributed content, scripts should be reviewed and tested carefully before use.

Using console applications

Several scripts in this library work by running an external command-line tool, capturing its output, and parsing the result to produce a new filename. This is done with the `ExecConsoleApp` function:

```
ExecConsoleApp(const Command: String; out Output: String): Cardinal
```

It executes the command string, blocks until the program finishes, captures everything written to standard output into `Output`, and returns the program's exit code. An exit code of `0` indicates success for all tools covered here.

Console applications typically write output using the OEM character encoding (the legacy DOS code page). Unless the tool provides its own UTF-8 output option, convert the output with `OemToWide` before using it in a filename:

```
UnicodeOutput := OemToWide(Output);
```

Always enclose both the executable path and the file path in double quotes within the command string, to handle spaces in paths correctly:

```
Command := '"' + ExePath + '" [options] "' + FilePath + '"';
```

ExifTool

[ExifTool](#) is a powerful, actively maintained tool for reading and writing metadata in a very wide range of file formats, including JPEG, RAW (CRW, CR2, NEF, ARW, and many others), HEIF, TIFF,

PNG, video files, and more. It supports EXIF, IPTC, XMP, and many vendor-specific tag namespaces.

Setup

1. Download the [ExifTool Windows package](#) and extract it into ReNamer's folder.
2. The archive contains a file named `exiftool(-k).exe`. **Rename it to `exiftool.exe`** before use. The `(-k)` suffix causes the tool to wait for the user to press Enter before closing — ReNamer cannot send that input, so the application will freeze if this step is skipped.

Script

The script below renames files using the **Date/Time Original** EXIF tag. Adjust the `TAG` constant to extract any other tag.

```
const
  EXE = 'exiftool.exe -t -charset filename=utf8';
  TAG = 'Date/Time Original' + #9 + '(.*)[\r\n]';

var
  Command, Output: String;
  Matches: TWideStringArray;

begin
  Command := EXE + ' "' + FilePath + '"';
  if ExecConsoleApp(Command, Output) = 0 then
    begin
      Matches := SubMatchesRegEx(Output, TAG, False);
      if Length(Matches) > 0 then
        FileName := Matches[0] + WideExtractFileExt(FileName);
      end;
    end;
end.
```

The `-t` flag produces tab-delimited output (`TagName<tab>Value`), which the `TAG` regex captures reliably. `#9` is the tab character. The `-charset filename=utf8` flag tells ExifTool to interpret filenames as UTF-8, which is necessary for Unicode paths.

Discovering available tags

Run `exiftool.exe <file>` from the command line to list all tags and their values for any given file. Paste the tag name into the `TAG` constant, replacing the example.

The raw tag value is used as the filename and may contain characters that are invalid in Windows filenames (colons are common in date strings, for example). Add a Replace or Clean Up rule after the Pascal Script rule to sanitise the result if needed.

Exiv2

[Exiv2](#) is a lightweight open-source library for reading and writing EXIF, IPTC, and XMP metadata from JPEG and RAW image formats (including CRW, CR2, and many others).

Setup

1. Download the [Exiv2 Windows package](#) and extract its `bin` folder contents into ReNamer's folder. The required file is `exiv2.exe` along with any DLLs it ships with.
2. Exiv2 is distributed in several build variants with different runtime dependencies. If you see errors about missing DLLs, install the matching [Microsoft Visual C++ Redistributable](#).

Script — extract any tag

The script below extracts the **camera model** (`Exif.Image.Model`) and prepends it to the filename. Change the `TAG` constant to extract any other Exiv2 key.

```
const
    TAG          = 'Exif.Image.Model';
    EXECUTABLE   = 'exiv2.exe';
    PARAMETERS   = '-Pt -K ' + TAG;

var
    Command, Output: String;
    UnicodeOutput: WideString;

begin
    Command := ''' + EXECUTABLE + ' ' + PARAMETERS + ' "' + FilePath + '"';
    if ExecConsoleApp(Command, Output) = 0 then
        begin
            UnicodeOutput := WideTrim(OemToWide(Output));
            if UnicodeOutput <> '' then
                FileName := UnicodeOutput + ' ' + FileName;
        end;
    end.
```

The `-Pt` flag prints the tag value only (no label or formatting). The `-K` flag filters output to a specific key, so `Output` contains just the value followed by a newline. `OemToWide` handles the encoding conversion.

Script — extract and reformat EXIF date

EXIF date values are formatted as `yyyy:mm:dd hh:mm:ss`, which contains colons that are illegal in Windows filenames. This variant extracts the date and reformats it cleanly. Adjust `DATE_TAG` and `DATE_FORMAT` as needed.

```
const
    EXECUTABLE = 'exiv2.exe';
    DATE_TAG    = 'Exif.Photo.DateTimeOriginal';
    DATE_FORMAT = 'yyyy-mm-dd hh.nn.ss';

var
    Command, Output: String;
    DateTime: TDateTime;

begin
    Command := '"' + EXECUTABLE + '" -Pt -K ' + DATE_TAG + ' "' + FilePath + '"';
    if ExecConsoleApp(Command, Output) = 0 then
        begin
            Output := WideTrim(OemToWide(Output));
            if TryScanDateTime('yyyy:mm:dd hh:nn:ss', Output, DateTime) then
                FileName := FormatDateTime(DATE_FORMAT, DateTime) + WideExtractFileExt(FileName);
        end;
    end.
```

`TryScanDateTime` parses the EXIF date string according to the format pattern. If parsing fails — for example, because the tag is absent — the script leaves `FileName` unchanged without raising an error.

Discovering available tags

Run `exiv2.exe -pa <file>` to list all metadata keys and their values for any given file. Keys follow the dotted namespace format (`Exif.Image.Model`, `Iptc.Application2.Headline`, etc.).

Tested with ReNamer 7.9 and Exiv2 0.27.2.

MediaInfo

[MediaInfo](#) extracts technical and descriptive metadata from a wide variety of audio and video formats, including MP4, MKV, AVI, MP3, FLAC, and many others. It can report codec names, duration, bitrate, encoding date, track titles, and more.

Setup

1. Download the [MediaInfo CLI package](#) (the command-line interface version, not the GUI) and extract it anywhere on your system.
2. Set the `MediaInfoExe` constant in the script to the full path of the `MediaInfo.exe` executable.

Script

The script below renames files using the **Encoded_Date** tag. Adjust `OutputParameter` to extract any other supported tag.

```
const
    MediaInfoExe    = 'C:\Tools\MediaInfoCLI\MediaInfo.exe';
    OutputParameter = 'General;%Encoded_Date%';

var
    Command: WideString;
    Output: String;

begin
    Command :=
        ''' + MediaInfoExe + ''' +
        ' --output="' + OutputParameter + '"' +
        ' "' + FilePath + '"';
    if ExecConsoleApp(Command, Output) = 0 then
        FileName := WideTrim(0emToWide(Output)) + WideExtractFileExt(FilePath);
end.
```

The `--output` parameter uses a template format: `Section;%TagName%`. The `General` section covers file-level properties; other sections include `Video`, `Audio`, `Text`, and `Image`. For example, `'Video;%Format%'` extracts the video codec name.

Discovering available tags

Use these built-in help commands from the command line to explore available tags:

- `MediaInfo.exe --Help` — general help.
- `MediaInfo.exe --Help-Output` — documentation for the output template format.
- `MediaInfo.exe --Info-Parameters` — full list of supported tag names.

Tested with ReNamer 7.9 and MediaInfo 26.01.

Xpdf

[Xpdf](#) provides a suite of PDF processing tools. The `pdftinfo.exe` utility extracts document properties such as Title, Author, Subject, Creator, and creation date from PDF files.

Setup

1. Download [Xpdf tools](#) and copy `pdftinfo.exe` into ReNamer's folder.

Script

The script below renames PDF files using their **Title** tag. Change the `TAG` constant to extract any other property.

```
const
    EXE = 'pdftinfo.exe -enc UTF-8';
    TAG = 'Title\s*\:\s*(.*?)[\r\n]';

var
    Command, Output: String;
    Matches: TWideStringArray;

begin
    Command := EXE + ' "' + FilePath + '"';
    if ExecConsoleApp(Command, Output) = 0 then
        begin
            Matches := SubMatchesRegEx(Output, TAG, False);
            if Length(Matches) > 0 then
                FileName := Matches[0] + WideExtractFileExt(FileName);
        end;
    end.
```

The `-enc UTF-8` flag instructs `pdftinfo.exe` to write its output in UTF-8, so no `OemToWide` conversion is needed. The `TAG` regex captures the value following the property label and colon; replace `Title` in the pattern to target a different property.

Discovering available tags

Run `pdftinfo.exe <file>` from the command line to list all properties available for a given PDF. Common tags include `Author`, `Subject`, `Keywords`, `Creator`, `CreationDate`, and `ModDate`.

Tested with ReNamer 7.9 and Xpdf tools 4.01.01.

TrID

[TrID](#) identifies the true type of a file by analysing its binary content, independently of the existing extension. It is useful for correcting mislabelled files or assigning extensions to files that have none.

Unlike the other tools in this library, TrID is integrated via **DLL import** rather than as a console application. See [Importing external functions](#) in the Examples article for background on this mechanism.

Unicode limitation: TrID library processes file paths as ANSI strings. Files with non-ASCII characters in their paths will not be handled correctly and are therefore skipped by the script.

Setup

Download and place both files into ReNamer's folder:

1. [TrIDLib.dll](#) — the identification library. See the [TrIDLib product page](#) for version history and additional notes.
2. [TrIDDefs.TRD](#) — the file type definitions database. TrID's identification quality depends on using an up-to-date database file, so re-download it periodically to pick up newly added file types.

Script

The script identifies the file type, applies a confidence cutoff of 20% to filter low-confidence matches, and sets the extension to the best match (or multiple candidates separated by `|` when several types score above the cutoff). It loads the definitions database once at initialisation and errors if the database file is missing.

```
{ TrID - file type detection by content }

const
    // Minimum confidence percentage for the detected
    // file extensions to be included in the result.
    TargetConfidenceCutoff = 20;

function TridLoadDefsPack(szPath: PChar): Integer;
    external 'TrID_LoadDefsPack@TrIDLib.dll stdcall';
function TridSubmitFileA(szFileName: PChar): Integer;
    external 'TrID_SubmitFileA@TrIDLib.dll stdcall';
function TridAnalyze: Integer;
```

```

external 'TrID_Analyze@TrIDLib.dll stdcall';
function TridGetInfo(lInfoType: Integer; lInfoIdx: Integer; sBuf: PChar): Integer;
external 'TrID_GetInfo@TrIDLib.dll stdcall';

const
    TRID_GET_RES_NUM      = 1;
    TRID_GET_RES_FILEEXT = 3;
    TRID_GET_RES_POINTS  = 4;
    TRID_BUFFER_SIZE      = 4096;

function GetExtensions(ConfidenceCutoff: Integer): WideString;
var
    Buffer, Extension: String;
    I, NumMatches, NumPoints, TotalPoints: Integer;
begin
    Result := '';
    SetLength(Buffer, TRID_BUFFER_SIZE);
    NumMatches := TridGetInfo(TRID_GET_RES_NUM, 0, PChar(Buffer));
    if NumMatches > 0 then
    begin
        TotalPoints := 0;
        // Count total points for all matches
        for I := 1 to NumMatches do
            TotalPoints := TotalPoints + TridGetInfo(TRID_GET_RES_POINTS, I, PChar(Buffer));
        // Iterate through all matches
        for I := 1 to NumMatches do
            begin
                // Check if a match passes the minimum confidence level
                NumPoints := TridGetInfo(TRID_GET_RES_POINTS, I, PChar(Buffer));
                if (NumPoints * 100 / TotalPoints) >= ConfidenceCutoff then
                begin
                    // Include the detected file extension
                    if TridGetInfo(TRID_GET_RES_FILEEXT, I, PChar(Buffer)) > 0 then
                    begin
                        Extension := LowerCase(String(PChar(Buffer)));
                        if (Length(Extension) > 0) and (Pos(Extension, Result) = 0) then
                        begin
                            if Length(Result) > 0 then Result := Result + '|';
                            Result := Result + Extension;
                        end;
                    end;
                end;
            end;
        end;
    end;
end;

```

```

        end;
    end;
end;
// Normalize the delimiter between file extensions
if Length(Result) > 0 then
    Result := WideReplaceStr(Result, '/', '|');
end;
end;

var
    Initialized: Boolean;
    AppPath, FileExtensions: WideString;

begin
    if not Initialized then
        begin
            Initialized := True;
            AppPath := WideExtractFilePath(GetApplicationPath);
            if TridLoadDefsPack(AppPath) = 0 then
                Error('TrID database file not found in the application folder.');
            end;

            // Skip Unicode filenames, because TrID library only works with ANSI filenames
            if FilePath <> UTF8Decode(WinCPToUTF8(UTF8ToWinCP(UTF8Encode(FilePath)))) then
                Exit;

            if TridSubmitFileA(PChar(FilePath)) <> 0 then
                begin
                    if TridAnalyze <> 0 then
                        begin
                            FileExtensions := GetExtensions(TargetConfidenceCutoff);
                            if FileExtensions <> '' then
                                FileName := WideStripExtension(FileName) + '.' + FileExtensions;
                            end;
                        end;
                    end;
                end.

```

`TridLoadDefsPack` is called once and receives the application folder path, from which it locates `TrIDDefs.TRD`. `TridSubmitFileA` submits the file for analysis and `TridAnalyze` runs the identification. `GetExtensions` collects all results above the confidence cutoff, converting them to lowercase and joining multiple candidates with `|`, which is an invalid filename character and will require the end

user to remove alternative candidates before renaming using other renaming rules.

Tested with ReNamer 7.9 and TrIDLib 1.0.2.

Renaming folders using file metadata

This script renames **folders** by reading a metadata tag from a representative file inside each folder, and appending the tag value to the folder name. It requires no third-party tools — only ReNamer's built-in `CalculateMetaTag` and `WideScanDirForFiles` functions.

A typical use case is music library organisation: given an album folder named `"Artist - Title"` containing MP3 files with an ID3 Year tag, the script produces `"Artist - Title (1993)"`.

Setup

[Add the folders](#) you want to rename to ReNamer's file list (not the files inside them).

Script

Adjust `MASK` to match the type of files to search for inside each folder, and `TAG` to the metadata tag to extract. The full list of supported tag names is in the [Meta Tags](#) reference.

```
const
    MASK = '*.mp3';
    TAG  = 'MP3_Year';

function FindFirstFile(const Folder: WideString): WideString;
var
    Files: TWideStringArray;
begin
    Result := '';
    SetLength(Files, 0);
    WideScanDirForFiles(Folder, Files, False, False, False, MASK);
    if Length(Files) > 0 then
        Result := Files[0];
end;

var
    MasterFile, TagValue: WideString;

begin
```

```

if not WideDirectoryExists(FilePath) then
    Exit;

MasterFile := FindFirstFile(FilePath);
if MasterFile = '' then
    Exit;

TagValue := WideTrim(CalculateMetaTag(MasterFile, TAG));
if TagValue <> '' then
    FileName := FileName + ' (' + TagValue + ')';
end.

```

`WideScanDirForFiles` scans the folder non-recursively for files matching `MASK` and returns their full paths. The first match is used as the source for the metadata tag. The `WideDirectoryExists` guard ensures the script acts only on folder entries and skips any regular files that may be in the list.

Tested with ReNamer 7.9.

Renaming files to match a reference file

This script renames files by taking the base name of a **reference file** located in the same directory, while preserving each file's own extension. It requires no third-party tools.

A typical use case is keeping companion files in sync. If a folder contains a reference file (`.master` for example) alongside generated or dependent files (data exports, logs, sidecar files), this script renames the companions to match the reference file's base name.

Setup

Add the files to be renamed to ReNamer's file list. The reference file must reside in the same folder as the files being renamed, but does not need to be in the renaming list itself.

Script

Adjust `MASTER_FILE_MASK` to match the reference file pattern.

```

const
    MASTER_FILE_MASK = '*.master';

function FindFirstFile(const Folder, FileMask: WideString): WideString;
var
    Files: TWideStringArray;

```

```

begin
    Result := '';
    SetLength(Files, 0);
    WideScanDirForFiles(Folder, Files, False, False, False, FileMask);
    if Length(Files) > 0 then
        Result := Files[0];
end;

var
    MasterFile: WideString;

begin
    MasterFile := FindFirstFile(WideExtractFileDir(FilePath), MASTER_FILE_MASK);
    if Length(MasterFile) > 0 then
        FileName := WideExtractBaseName(MasterFile) + WideExtractFileExt(FilePath);
end.

```

`WideScanDirForFiles` scans the source file's own folder for the first match against `MASTER_FILE_MASK`. `WideExtractBaseName` takes the base name from the reference file, and `WideExtractFileExt(FilePath)` preserves the original extension of the file being renamed. If no reference file is found, `FileName` is left unchanged.

Tested with ReNamer 7.9.