NAME

groff_sanitize – filter unwanted constructs out of groff data streams

DESCRIPTION

The **groff_sanitize** auxiliary macro package provides a collection of filters, all of which are accessed through a common **.sanitize** macro call, for substitution, or removal of particular **groff**(7) constructs, from copies of selected fragments of the document source input data stream. Such filtering of the input data may be useful, for example, when creating a PDF document outline, or similar, derived from some content within the document body, whence any specified formatting controls may not be appropriate for inclusion within the outline specification.

USAGE

The **groff_sanitize** macros *may* be loaded using a command line option, as prescribed in compliance with the conventional syntax for **groff**(1) and **pdfroff**(1) commands:

groff [*–option* ...] **–m sanitize** [*–option* ...] [*file* ...]
pdfroff [*–option* ...] **–m sanitize** [*–option* ...] [*file* ...]

However, these macros may be more commonly loaded from within document source, or, perhaps even more commonly still, from within another dependent macro package, using a request of the form:

.mso sanitize.tmac

After the **groff_sanitize** macros have been loaded, the entire gamut of their associated filters may be applied, to some specific text, by a macro call of the form:

.sanitize <*varname*> <*text* ...>

in which the <*varname*> argument is required; it *must* represent a valid **groff**(7) name for a string, in which the filtered value of the <*text* ...> argument is to be stored.

In practice, there is little to be gained by calling **.sanitize** *directly* from the top level of any document source; more practical usage encapsulates a **.sanitize** macro call within another macro, (which may be either user-defined, or provided by another macro package), such that the original text is used, in two or more distinct contexts, with at least one context using the filtered text, while another uses the original unfiltered form; such usage is illustrated in the **EXAMPLES** section of this manual page.

PRINCIPLE OF OPERATION

On entry to the **sanitize** macro, its *first* argument, which is designated as "<*varname*>", and which *must* represent a valid **groff**(7) identifier, is interpreted as the name of a string in which the resultant sanitized text is to be returned to the calling macro; this string is defined, and initialized as an *empty* (i.e. zero-length) string, and is also assigned an internal alias of **sanitize:result**. A further internally named string, **sanitize:residual**, is also defined, and initialized with a value which is comprised of the aggregate content of the second, and any additional arguments, (designated as "<*text* ...>"); when more than one argument is incorporated into the aggregate which forms "<*text* ...>"), then, before the content of each additional argument, a token for one word-space is introduced into **sanitize:residual**, to separate the content of that argument from that of its predecessor.

Following this initialization, **sanitize** enters a cyclic processing phase, wherein one token is removed from the beginning of **sanitize:residual** in each cycle, with additional cycles continuing until no further tokens remain. The token which is removed, within each cycle, is examined to determine how it should be processed in that particular cycle; the token-specific processing actions are:

- Any regular character token is simply appended to **sanitize:result**, and processing continues with the next cycle.
- A special token, which corresponds to one of **groff**(7)'s escape sequences with a single-token representation, and for which an associated **groff_sanitize** filter, (see the **FILTER ACTIONS** section), has been specified, will either be discarded, or it will be replaced by some designated *substitute text*, which is appended to **sanitize:result**, before commencing the next cycle.
- Any instance of the **groff**(7) *escape character* token will initiate an intermediate *look-ahead* processing cycle; this will examine subsequent tokens, within **sanitize:residual**, to identify a particular **groff**(7) escape sequence, and any arguments which may be associated with it. When any such escape sequence has been identified, and it is associated with a **groff_sanitize** filter, *all* tokens which have been consumed by the *look-ahead* cycles will be removed from **sanitize:residual**, and will either simply be discarded, or some designated *substitute text* will be

appended to **sanitize:result**, depending on which of the particular **FILTER ACTIONS** have been specified, before proceeding to the next principal token analysis cycle.

When the preceding cyclic processing has exhausted *all* of the tokens from **sanitize:residual**, *all* of the **sanitize** macro's internally defined local identifiers, *including* **sanitize:residual**, and **sanitize:result**, will be deleted, leaving the sanitized text, which had been accumulated in **sanitize:result**, in the string named by the <*varname*> argument, for return to the caller.

RESERVED IDENTIFIERS

The **groff_sanitize** macro package reserves regions of both the **groff**(7) string and the numeric register namespaces, in which all internal identifiers begin with the label "**sanitize:**"; with the exception of those macros, strings, or numeric registers which are *explicitly* documented as "*user definable*", or "*user modifiable*", in the **FILTER ACTIONS** section, users are *strongly* advised to avoid defining, or modifying, any macro, string, or numeric register with a name which begins with this label.

FILTER ACTIONS

The collection of filters, which is predefined by the **groff_sanitize** package, comprises:

- **sanitize:scan.reject** <*token-list*> . . .

  This is a token elimination filter. Implemented as a **groff**(7) string, its value represents a sequence of optionally quoted <*token-list*> specifications, each of which takes the form:

  ["]<*opening-delimiter*><*reject-token* . . .><*closing-delimiter*>["]

  Within each such <*token-list*> specification, the <*reject-token*> part comprises a list of one or more **groff**(7) entities, each of which yields a *single input token* when read in copy mode. Each <*reject-token*> list *must* be enclosed within a pair of arbitrary delimiter tokens, the <*opening-delimiter*> and the <*closing-delimiter*>, which *must* be represented by *identically* the same input token; this *must not* appear *anywhere* within the enclosed <*reject-token*> list.

  During **sanitize** macro processing, each token which is abstracted from **sanitize:residual** is compared, in turn, with each token which appears in the aggregate collection of <*token-list*> constituents, which comprise the value of **sanitize:scan.reject**, until either a matching token is found, or all tokens in the aggregate <*token-list*> have been compared, and no matching token has been found. If a matching token *is* found, nothing is added to **sanitize:result**, and the **sanitize** macro moves on, to process the next available token, if any, in **sanitize:residual**.

  By default, **groff_sanitize** defines **sanitize:scan.reject** with an aggregate <*token-list*> which comprises the trio of tokens, "`\&`", "`\%`", and "`\:`", as established by the specification:

  .ds sanitize:scan.reject `"'\&\%\:'\"`

  This specification is *user-modifiable*, either by use of an alternative "**.ds**" request, to redefine the collection of <*token-list*> specifications, (of which there is just the one in the default case), in its entirety, or by use of an "**.as**" request, to append additional <*token-list*> specifications to it; in the latter case, each additional <*token-list*> specification, which is individually defined, *must* conform, to the

  <*opening-delimiter*><*reject-token* . . .><*closing-delimiter*>

  pattern, and *must* be separated from its predecessor by one or more white space tokens, within the definition of the **sanitize:scan.reject** string.

  When **sanitize:scan.reject** is defined to incorporate more than one <*token-list*> specification, while the <*opening-delimiter*> and <*closing-delimiter*> within each *must* be represented by the *same* token, it is *not* necessary to use the same delimiter token for *every* individual <*token-list*> specification; it is permitted, and may be convenient, to employ a token which is included within the <*reject-token*> list of one <*token-list*> specification, as the delimiter for another, or analogously, to include the delimiter token of one <*token-list*> specification in the <*reject-token*> list of another.

  Subject to the restrictions that each *must* be represented by a single **groff**(7) input token, and that the chosen token *must not* appear within any <*reject-token*> list for which it is specified as a delimiter, the choice of delimiter tokens is entirely arbitrary. The ASCII apostrophe, "`'`", is usually a suitable choice. As an alternative, the ASCII double quote character, ‘`"`’, may be considered,

but this is a less convenient choice, for reasons which will be explained later, in the **CAVEATS AND BUGS** section.

- **sanitize:scan.subst** <*substitution-group*> ...

  This is a token substitution filter. Like the **sanitize:scan.reject** filter, this is also implemented as a **groff**(7) string; in this case its value represents a sequence of space separated, optionally quoted token <*substitution-group*> specifications, each of which takes the form:

  ["]<*start-delimiter*><*token* ...><*mid-delimiter*><*substitute-text*><*end-delimiter*>["]

  Processing of this filter occurs *after* the **sanitize:scan.reject** filter, and then *only* if that filter *did not* match the current input token.

  The effect of this filter is fundamentally analogous to that of **sanitize:scan.reject**, insofar as it processes each of its <*substitution-group*> specifications in turn, comparing the current input token from **sanitize:residual** to each individual <*token*> which has been specified between the <*start-delimiter*> and the <*mid-delimiter*> tokens, in turn; if a token match is found, whereas the **sanitize:scan.reject** filter terminates its processing *without* adding *any* further content to **sanitize:result**, the **sanitize:scan.subst** filter appends the content which is specified within the <*substitute-text*> field of the matching <*substitution-group*>, to **sanitize:result**, discards the matching input token from **sanitize:residual**, and terminates its action, with *no* consideration of any further <*substitution-group*> comparisons for the input token.

  Analogously to the choice of <*opening-delimiter*> and <*closing-delimiter*> tokens, within the definition of the **sanitize:scan.reject** filter, the <*start-delimiter*>, <*mid-delimiter*>, and <*end-delimiter*> tokens, within each individual <*substitution-group*> within the definition of the **sanitize:scan.subst** filter, *must all* be represented by *the same* token, but it is permitted to choose different delimiter tokens, in distinct <*substitution-group*> specifications.

  By default, **groff_sanitize** provides a definition of the **sanitize:scan.subst** filter, comprising *two* initial <*substitution-group*> specifications, thus:

  .ds sanitize:scan.subst "'\-'-' "'\ \~' '"\"

  the effect of which is to request replacement of any instance of a "\-" input token with an ASCII hyphen-minus token, and any instance of either a "\<SP>" input token, or a "\~" input token, with an ASCII space (i.e. "<SP>") token, in the resultant sanitized text.

  Notice that, in *both* of these default <*substitution-group*> specifications, the ASCII apostrophe is employed as the delimiter token. Also note that the second of these <*substitution-group*> specifications, in its entirety, is enclosed within a pair of ASCII double quote tokens. This is necessary, because the specification for this <*substitution-group*> contains white space tokens; the rationale for this requirement is explained later, in the **CAVEATS AND BUGS** section.

  As is the case for the **sanitize:scan.reject** filter, the **sanitize:scan.subst** filter may be modified by the user, either by redefining the specification string in its entirety, or by appending desired additional <*substitution-group*> specifications to the end of the existing definition. Note that, when modifying the specification string, individual <*substitution-group*> specifications *must* be separated by white space; within each <*substitution-group*>, all three delimiters *must* be represented by *identically* the *same* token, and any <*substitution-group*>, in which white space is included, *must* be enclosed, within the specification string, between a pair of ASCII double quote character tokens.

  In addition to the <*substitution-group*> sequences, which are included within the default definition of **sanitize:scan.subst**, as specified above, **groff_sanitize** defines *one* further substitution, with an effect which is equivalent to:

  .as sanitize:scan.subst " '\e'\\\\'\"

  The intent of this is to insert a reverse-solidus escape character into the sanitized output text, when an "\e" token is encountered in the input text, primarily for the purpose of introducing an octal-encoded special character into the sanitized output text. This filter *does* deliver the intended effect; however, unlike any other **sanitize:scan.subst** filter definition, this is hard-coded into the **groff_sanitize** macro implementation, and *cannot* be overridden, or redefined by the user.

- **sanitize:esc-char.subst** <*substitution-group*> ...

  This is an analogue of the **sanitize:scan.subst** filter, except that, whereas the latter specifies substitutions for escape sequences such as "\~", and "\<SP>", each of which is represented by a *single input token*, the **sanitize:esc-char.subst** filter supports defined substitutions for any syntactically similar, but *semantically different* escape sequence, such as "\0", for which the **groff**(7) representation comprises *two separate input tokens*.

  The form of <*substitution-group*> specifications for the **sanitize:esc-char.subst** filter:

  ["]<*start-delimiter*><*escape* ...><*mid-delimiter*><*substitute-text*><*end-delimiter*>["]

  may *appear* to be *syntactically* similar to that for the **sanitize:scan.subst** filter:

  ["]<*start-delimiter*><*token* ...><*mid-delimiter*><*substitute-text*><*end-delimiter*>["]

  However, the two differ *semantically* insofar as, whereas the <*token*> list specification for the **sanitize:scan.subst** filter is expected to comprise only entities which are each represented by a single **groff**(7) input token, the corresponding <*escape*> list specification, which is expected by the **sanitize:esc-char.subst** filter, should comprise one or more **groff**(7) escape sequences, each of which is represented by *exactly two* input tokens, the first of which should be the **groff**(7) escape character, while the second is any other input token which is representative of any valid **groff**(7) escape sequence, which does *not* take an argument.

  Apart from this semantic difference, in the expression of their respective <*substitution-group*> specifications, the **sanitize:esc-char.subst** filter exhibits, fundamentally, the same behaviour as the **sanitize:scan.subst** filter: when any escape sequence which is represented in an <*escape*> list specification is encountered, within the input text, it is discarded, and its corresponding <*substitute-text*> is appended to the resultant sanitized text, in its place.

  The **groff_sanitize** default definition of **sanitize:esc-char.subst** is:

  .ds sanitize:esc-char.subst `""'\0' '" '\,\/''\"`

  which results in replacement, within sanitized text, of the "\0" *input* escape sequence by a simple ASCII space, and *removal* of both the "\," and "\/" escape sequences, (effectively achieved by replacing each by *nothing*).

  Just as the **sanitize:scan.reject** and **sanitize:scan.subst** filters may be modified, or redefined, to handle additional single-token escape sequences, the **sanitize:esc-char.subst** filter may be modified, or redefined, to accommodate additional dual-token sequences; as before, when any <*substitution-group*> specification includes white space, that specification *must* be enclosed in ASCII double quotes, within the string definition, while all individual <*substitution-group*> specifications *must* be separated by *unquoted* white space.

  Notice that there is no *direct* analogue of the **sanitize:scan.reject** filter, for the removal of dual-token escape sequences; however, an equivalent effect may be achieved by use of the **sanitize:esc-char.subst** filter, as is illustrated within it default definition for the removal of the "\," and "\/" escape sequences, by definition of a <*substitution-group*> specification with *nothing* in the <*substitute-text*> field.

- **sanitize:esc–(***??* <*substitute-string*>

  This is a generic template for a special character substitution filter; it may be instantiated, as required, by defining strings of the form:

  .ds sanitize:esc–(*??* "*substitute-string*\"

  in which the "*??*" place-holder is replaced by any two-character special character name, as documented in **groff_char**(7), to implement a substitution filter for the corresponding named special character escape sequence, such that, when this escape sequence is encountered by the **sanitize** macro, while processing its local **sanitize:residual** string, the value which is specified as <*substitute-string*> will be appended to the **sanitize:result** buffer, in place of the original escape sequence.

groff_sanitize defines *five* default instances of the **sanitize:esc–(***??* filter, namely:

.ds sanitize:esc–(hy "–\"
.als sanitize:esc–(mi sanitize:esc–(hy

which result in the substitution of a single ASCII hyphen-minus character, in sanitized text, in place of each "`\(hy`", or "`\(mi`" input sequence, and:

.ds sanitize:esc–(em "\e204\"
.ds sanitize:esc–(en "\e205\"
.ds sanitize:esc–(rs "\e134\"

which substitute the PDFDocEncoding octal representation equivalent to each of "`\(em`", "`\(en`", and "`\(rs`", respectively.

It may be observed that, whereas any instance of a filter, which is derived from this template, is *always* defined in a format which may seem to be indicative of **troff**'*s* traditional "`\(??`" representation of two-character special character escape sequences, the **sanitize** macro will recognize **groff**(7)'s alternative "`\[??]`" representation as being equivalent, and will process it accordingly, and so, also append the assigned value, corresponding to <*substitute-string*> to **sanitize:result**, in place of this alternative representation of the escape sequence.

- **sanitize:esc–generic**

  This is a generic template for an escape sequence elimination filter; it may be instantiated to recognize **groff**(7) escape sequences in any of the three forms, "`\`*?c*", "`\`*?* (*cc*", or "`\`*?*`[...]`", in which the "*?*" placeholder represents any single-character *function identifier* for an escape sequence which takes a single-character argument, "*c*", a two-character argument, "*cc*", or an arbitrary length argument, "`[...]`". It may be instantiated for any specific escape sequence, with *function identifier* "*?*", by defining an alias of the form:

  .als sanitize:esc–*?* **sanitize:esc–generic**

  which then has the effect of removing any instance of "`\`*?*", together with its argument, in any of the three supported formats, from **sanitize:residual**, while adding *nothing* to the sanitized text which is to be returned through **sanitize:result**.

  groff_sanitize defines *two* default instances of the **sanitize:esc–generic** filter, namely:

  .als sanitize:esc–f sanitize:esc–generic
  .als sanitize:esc–F sanitize:esc–generic

  the combined effect of which prevents the propagation of any "`\f`*c*", "`\f(`*cc*", "`\f[...]`", "`\F`*c*", "`\F(`*cc*", or "`\F[...]`", escape sequence into sanitized text; users may wish to extend the effect of this filter, by defining additional aliases, modelled on this default pair, to suppress propagation of other syntactically similar escape sequences.

- **sanitize:esc–delimited**

  This is a complement for the "`\`*?*`[...]`" form of the **sanitize:esc–generic** filter; like the latter, it eliminates a functional escape sequence, in which the *function identifier* token is followed by an arbitrary length argument, from the resultant sanitized text; it differs from the latter in the form in which that argument is expressed.

  Whereas the **sanitize:esc–generic** filter expects an arbitrary length escape sequence argument to be expressed in the form "`[...]`", the **sanitize:esc–delimited** filter expects the argument to the escape sequence to have the form:

  <*opening-delimiter*><*argument-text*><*closing-delimiter*>

  with the <*opening-delimiter*> and the <*closing-delimiter*> being represented by *identically* the same arbitrarily chosen input token; it is assumed that this arbitrarily chosen delimiter token does *not* appear *anywhere* within <*argument-text*>.

  By default, **groff_sanitize** defines *two* instances of the **sanitize:esc-delimited** filter, namely:

  .als sanitize:esc–s sanitize:esc–delimited
  .als sanitize:esc–v sanitize:esc–delimited

  Users may add additional filters, similar to these, to support any other escape sequences which exhibit similar semantics to this default pair.

Although the **sanitize:esc−generic** and **sanitize:esc−delimited** filters are both implemented, primarily, as elimination filters, both support a mechanism for configuration of any particular instance of either as a substitution filter. Given that any instance of either of these filters is invoked *exclusively* through an alias name of the form "**sanitize:esc−***?*", in which "*?*" represents any single ordinary character, which uniquely identifies the particular **groff**(7) escape sequence to which the filter instance relates, a particular instance of either filter, "**sanitize:esc−***?*", may be configured as a substitution filter by the addition of a further user-supplied definition of the form:

.ds sanitize:esc−*?***.subst** <*substitute-text*>

(in which "*?*" represents *identically the same* ordinary character as the escape sequence identifier in the filter instance to which the "<*substitute-text*>" definition relates). Such "<*substitute-text*>" definitions may be specified on an ad-hoc basis, and will remain in effect for all invocations of any particular filters to which they relate, following definition, until redefined, or deleted.

FILES

/usr/local/share/groff/site-tmac/sanitize.tmac

Implements the **sanitize** macro, and its supporting predefined **groff_sanitize** filters.

CAVEATS AND BUGS

Inclusion of any escape sequence, which lacks an associated **groff_sanitize** filter action assignment, within text which is to be sanitized, may have unpredictable, and undesirable effects.

Passing input text, which includes any use of the "`\s`" escape sequence, in any of its supported forms other than the "`\s`<*delimiter*><*expression*><*delimiter*>" form, to the **sanitize** macro, will confuse the **sanitize:esc−s** filter, producing unpredictable, and probably undesirable, results in the sanitized text.

If redefining either the **sanitize:scan.reject**, or the **sanitize:scan.subst** filter, their associated <*token*> list specifications are interpreted *strictly* as sequences of single-token entities, each of which nominally represents a special escape sequence, with no associated argument; inclusion of any token sequence, which does *not* represent such an entity, will have unpredictable, and most likely undesirable results.

Conversely, the <*escape*> list specifications for the **sanitize:esc-char.subst** filter *must* comprise *only* sequences of dual-token escape sequences, *none* of which accept any argument; inclusion of any tokens which are not paired, with the escape character token as the first of the pair, will not be interpreted as required, to deliver the intended behaviour of this filter.

When redefining, or otherwise modifying, any of the **sanitize:scan.reject**, **sanitize:scan.subst**, or **sanitize:esc-char.subst** filter specifications, it is important to understand how any ASCII double quote tokens will be interpreted, within these specification strings. Effectively, each of these strings may be passed, *unquoted*, within an argument list to an internal **groff_sanitize** macro, and thus, the argument grouping effect of the ASCII double quote token will override any other intended effect. Consequently, while it is certainly possible to work around the limitation, the choice of the ASCII double quote as a <*substitution-group*> delimiter token may be less convenient than some alternative choice.

It is *strongly* recommended that the "`\e`" escape should *not* be used to represent a reverse-solidus escape character, when this is to be propagated into sanitized text; the preferred representation for this character should be the "`\(rs`" escape sequence, (or its alternative "`\[rs]`" representation).

EXAMPLES

The **sanitize** macro is not, typically, called *directly* from any user's **groff**(7) document source; it *is*, however, often incorporated into higher level macros, such as the following example, which inserts text, the specification of which may incorporate some arbitrary format controlling escape sequences, as a heading, into a PDF document body, while placing a sanitized copy of the same text into an associated PDF document outline:

```
.de H
.\" Usage: .H <level> <text> ...
.\"
.\" Save the heading level argument.
.\"
.   nr \\$0.level \\$1
.
.\" Set each new heading one paragraph space below any
.\" text which precedes it.
.\"
.   sp \\n(PDu
.
.\" Reduce arguments to heading text, and copy this to the
.\" PDF document outline, at the specified nesting level.
.\"
.   shift
.   sanitize \\$0.text \\$@
.   pdfhref O \\n[\\$0.level] -- \\*[\\$0.text]
.
.\" Write a formatted copy of the heading text to the body
.\" of the PDF document.
.\"
.   ft B
.   nop \&\\$*
.   ft 1
.
.\" Clean up temporary local storage.
.\"
.   rr \\$0.level
.   rm \\$0.text
..
```

This heading macro might be invoked by a call such as:

```
.H 1 An Example PDF Heading with \F[C]sanitize\F[] Requirement
```

which, in the absence of the **sanitize** macro call within the **H** macro definition, would result in artefacts of the embedded change of font family escape sequences infiltrating the corresponding PDF document outline reference.

AUTHORS

The **groff_sanitize** macros are provided by the *groff-pdfmark* auxiliary package, which was written by Keith Marshall <keith@address.hidden>; this is maintained independently of *GNU roff*, at Keith's *groff-pdfmark* project hosting web-site <https://savannah.nongnu.org/projects/groff-pdfmark/>, whence it is also distributed, and the latest version may *always* be obtained.

SEE ALSO

groff(1), **pdfroff**(1), **groff**(7), **groff_char**(7)

More comprehensive documentation, on the use of the *groff-pdfmark* macro suite may be found, in PDF format, in the reference guide "*Portable Document Format Publishing with GNU Troff*", which has also been written by Keith Marshall; the most recently published version of this guide may be read online, by following the appropriate document reference link on the *groff-pdfmark* project hosting web-site <https://savannah.nongnu.org/projects/groff-pdfmark/>, whence a copy may also be downloaded.