TEXcount
Technical documentation
Version 3.0

Abstract

The aim of this document is to explain the implementation details of TEXcount with the aim of aiding anyone who wishes to modify the Perl code (including the author). To be of practical use, it will require some knowledge of Perl and familiarity with TEXcount: while full fledged development of TEXcount requires a working knowledge of Perl, code modification may often be done with only limited experience with Perl.

Contents

1 Introduction

1.1 TEXcount versioning

The version number is on the form *major.version.subversion.build*. Main releases contain only the first to terms, implying that subversion and build number are both zero. Minor releases only contain the first three terms. Main as well as minor releases should be functional, tested versions. The subversion number can also be alpha ($\alpha = -2$) or beta ($\beta = -1$) for which the testing has been limited. The build number is used to keep track of changes and versions during development: they may be made available, but are purely for testing.

1.2 Some things you need to know about Perl

TEXcount is written in Perl. The entire script, including macro rules and help texts, is contained in one file. This makes the file somewhat big, and modularisation of the code is therefore not strictly enforced. I have however tried to structure the code somewhat.

Perl has a few build-in data structures which are referenced in somewhat different manners. In this document, it will be important to recognize the difference between three different types of data: regular variables, arrays and hash maps.

$*name*=*value*: The $ at the start indicates that it is a Perl variable. The value can be numbers or strings, or it can be a reference which points to another data object (e.g. array or hash map).

@*name*=(*value*,...): The @ at the start indicates that this is an array. The positions are indexed from 0 to length -1.

%*name*=(*key*=>*value*,...): The % at the start indicates that this is a hash map that maps keys to values. The key will usually be a string, but can also be a number.

sub *name* {...}: This defines a subroutine: function or procedure. Normally, these can be defined anywhere in the script, and I have generally placed them after the main program.

Note that (*value*,...) is a list of values, not an array or a hash: it simply produces a list of values that are used to fill the defined array or hash. Arrays and hashes can also be produced directly by [*value*,...] or {*key*=>*value*,...} respectively. Both these return a reference to the array/hash rather than the array/hash itself.

In much of the code, hashes are passed by reference: e.g. if %hash is a hash, $href=\%hash stores a reference to the hash, where the leading \ causes a reference to be returned rather than the hash itself. Retreiving a value from the hash is done by $hash{*key*} or $href->{*key*} if the hash is accessed by reference. Note that individual values in array and hashes are prefixed by $: i.e. $*array*[*index*] and $*hash*{*key*}.

TEXcount makes extensive use of regular expressions (regex): expressions on the form $*string*=~/*pattern*/ and $*string*=~s/*pattern*/*replace*/. In TEXcount, the main use is to recognize (and remove) tokens (words, macros, spaces, etc.) at the start of a string of LATEX code. Some of these may be fairly simple to understand, while others may be more complex.

2 Overview

2.1 Code structure

TEXcount is written in Perl, and although hardly the best structured and documented code ever seen, I have tried to structure and document it somewhat. In particular, some parts of the code have been written with modifications in mind so that users can make their own changes without in-depth knowledge of Perl or the TEXcount script.

Here's a quick walk-through of the code structure and comments on how easily the code may be modified. Some parts of the code are marked as *CMD specific*. There are two version of the script: the CMD

version intended for command line use, and the CGI version used with the web interface. The one you have is the CMD version.

HEADER AND IMPORTS: *The shebang (#!) and use imports.*

INITIAL SETUP: *These set up global variables prior to execution.*

Settings and setup variables: The start of the script sets of initial settings and variables. Many of these may be modified by command line options, but if you want to change the default behaviour these may be changed. However, note that there is a list @StartupOptions intended for this: initially, it is empty, but this is probably the simplest place the change startup options.

Internal states: As of version 2.3, internal state identifiers (which are numerical codes) have been defined as STATE, TOKEN and CNT variables, and these are also defined here. A few subroutines for interpreting these states have been included here, although most subroutines are defined after the main code, since they are intimately tied to the state's numerical values. None of these are intended to be modified.

Styles: The style definitions basically define which elements to print for each of the verbosity levels. These map element names to ANSI colour codes. When used with HTML, the element names are used as tag classes. If you wish to change the ANSI colour scheme, or change which elements are written in each verbosity option, these may be changed.

Word pattern definitions: This section contains regular expression patterns for identifying words and macro options. In addition, the additional character classes defined by TEXcount are defined here. If you have special needs or wishes, modifying these definitions may be an option.

TEXcount parsing rules: This is the section in which the main rules for interpreting the LATEX code is specified: the exception is a few hard-coded rules that do not follow these general patterns. These are hashes that map the macro or environment name to the macro handling rules. First, the default rules are defined, then packages specific rules are defined.

MAIN: *This is the top-level code which gets executed. All else is done through calls to subroutines.*

Main TEXcount code: This is the main code that is run. It is very simple: just a call to the method MAIN passing the command line options.

SUBROUTINES: *The subroutines are organised into blocks. Subroutines names use capital letters or initials if they are main routines (like public in other languages) to be used at the top-level, lower case if they may be used throughout but are considered to be lower-level subroutines, prefixed by one or two underscores (_) if used only within the block.*

Main routines: The MAIN routine gives the general processing flow. This in turn calls routines to parse to command line options, process/apply the options, parse the TEX/LATEX files, and finally summarise the final results. The main routines are CMD specific.

CMD specific subroutines: These are subroutine versions that are CMD specific, e.g. file inclusion and ANSI colours. Their location is somewhat illogical: logically, they might belong later together with related subroutines, but have been placed this early because they are specific to the CMD (or CGI) version.

Option handling: After parsing the options, the option values are processed using these subroutines. Some of the option handling operations call on global variables, whereas some are more hard-coded. Like the global variables, if you have special wishes or needs, there may be parts here that can be modified quite easily to change default settings or effects of specific options.

TEX object: The main role of the TeX object (which is technically not an object in the ordinary sense but just a hash) is to be a container object which links to the TEX/LATEX code, the word count object, etc. The TeX object pertaining to any parsed TEX/LATEX file is passed along from subroutine to subroutine, usually called $tex. The Main object produced by getMain is a simple substitute for the TeX object for use when none is available, e.g. to catch errors not specific to any particular TeX object.

File reading routines: These are used to read files and STDIN.

Parsing routines: These contain the main routines for parsing the TEX/LATEX code. The main worker method is the _parse_unit which parses a block of code: the *unit*. A unit of code may be the contents of an environment, a {...} group, a macro option or parameter, etc. The parsing of one unit is determined by the parsing state, which is passed to the parsing method, and the end marker which indicates which token marks the end of the unit. Different subroutines are then used to process the different types of code: macros, environments, TC instructions, etc. Amongst these routines are also routines for converting the parsed code into tokens, which is done one token at the time which is then removed from the start of the code.

Count object and routines: The count object contains the counters as an array, plus titles and labels; in addition it can contain a list of subcounts which are themselves count objects. The count object is used for each file, but also to summarise multiple files, and region counts within files (e.g. per section). The TeX object contains an active count object to which newly counted words, equations, etc. get added. However, each TeX object also has a summary count object which will contain the final sum.

Output routines: First, there are some routines for general output, i.e. independent of specific TeX objects. There are then some routines for formatting output, e.g. for the verbose output. There are also routines for printing count summaries in various formats. A special set of routines exist for printing the verbose output itself, and some of these are also involved in the parsing.

Help functions: These routines are used to print help.

HTML functions: These are routines for producing HTML output. In particular, the HTML style is defined here and may be easily modified.

Text data: Some texts are not hard-coded into the script, but added as text data at the end. There are some routines defined to handle the text data, and then the text data itself.

Perl will first process the setup section which defines global variables, arrays and hashes. It then executes the main section (consisting of the call to MAIN), whereafter it exits. The subroutines and text data follow after the exit.

There is a separate CGI version of the TEXcount script for the web service. While this is mostly the same as the regular command line version, there are some differences in how options are set and LATEX documents are read. Occasional differences in the CGI version will be commented on, but the main emphasis will be on the command line version of TEXcount.

2.2 Global variables

A number of globally defined variables, including constants, arrays and hashes, are defined at the start of the program. These fall into a few different categories.

There are a number of variables defined for storing options and settings, many of which can be modified by command line options. In addition, there are few variables for global summaries and statistics, as well as a few for internal states during parsing.

Global constants are defined to represent different states and counters. One set of constants,$CNT_..., specify the position of the different counters in the counting array; parser states are defined as $STATE_...; token types are named $TOKEN_.... For example, the parsing state $STATE_TEXT indicates that a block of LATEX code should be parsed and have words counted as text words. The constants simply take numerical

values, but help make the code more readable. Together with some of these are defined functions for interpreting or transforming these

Alternative settings for different options are defined in a number of hashes, e.g. %STYLES indicating which tokens to print at different levels of verbosity, and %NamedLetterPattern which stores alternative regex rules which may be used to recognize letters.

A special set of global settings are the macro handling rules that are stored in a number of hashes: %TeXmacro, %TeXenvir, etc. as well as similar sets of hashes for package specific rules.

2.3 TEXcount objects

First, note that what is referred to as objects here are just hash maps with a predefined set of values. However, these serve the same purposes as objects. There are no explicit class specifications defining these, just a set of functions returning hashes that contain the required keys, some of which may even be optional. Still, it is useful to think of them as objects, and their main purpose is to encapsulate data so that they can conveniently be passed around.

The *Main* object Each TEXcount session instantiates a singleton *Main* object. This is used as a replacement when no *TeXcode* object is available for capturing (counting and storing) error messages and warnings.

The *TeXcode* object The *TeXcode* object encapsulates the LATEX code that is to be parsed as well as counts and lists of reported errors. In the code, it is generally referred to using the $tex variable.

The *count* object The *count* object is primarily a container for the array of counts: i.e. the array containing word counts and counts of headers, equations, etc. However, it also keeps track of subcounts from contained files, sections, etc.

A more detailed explanation of the different objects is provided in section 4

2.4 Main program flow

The main program consists of a single call to the procedure MAIN. This does the following:

Initialise: Most of the initialisation is done when defining the global variables, but some initalisation required code execution: e.g. OS specific initialisation.

Check_Arguments: Runs an initial check of the command line arguments passed to TEXcount, e.g. for -help, and may exit TEXcount.

Parse_Arguments: Parses the command line arguments, setting option variables, and returns the list of LATEX files to be parsed.

Apply_Options: This applies the options set either in the initial setup or initialisation, or when parsing the arguments. While most options are set directly during the argument parsing, settings that may depend on multiple options or options that should be applied only once, e.g. initialising the output and writing the HTML header, are applied here.

Parse files (or write help or error message): The file parsing calls Parse_file_list with the list of files to be parsed, and this returns the total count object. Apart from this, help, summary output and error reports are produced if required.

Close_Output: This just makes sure the output channel is properly closed, e.g. writing closing HTML code.

In the CGI version of TEXcount, Initialise, Check_Arguments and Parse_Arguments are replaced by a single call to Set_Options. Also, since the CGI version only processes one file, alternatives for parsing and reporting on multiple files are not required and is instead replaced by a single call of parse.

2.5 How TEXcount processes LATEX documents

The parse routine is the entry point for parsing LATEX code of a single file. It takes a *TeXcode* object, the container object of a LATEX document and its corresponding *count* object, performs the parsing of the entire document. The counts are stored in the counter in the *TeXcode* object.

The hierarchy of delegation from MAIN down to parse is as follows:

MAIN calls Parse_file_list with a list of files which return the total count (a count object) for MAIN to report.

Parse_file_list calls parse_file for each file in the provided file list, and for STDIN (identified by $_STDIN_) if the option to parse standard input has been set. It then aggregates the counts returned by parse_file into a total count which it returns.

parse_file calls _add_file, first for the main file, and then again for each included file if file inclusion (-inc) has been set. The aggregation of counts is done by _add_file into a total count object provided by parse_file, and this total count object is then returned by parse_file upon completing the parsing of the main file as well as all included files.

_add_file reads the file into memory, creates a *TeXcode* object which encapsulates the LATEX code and the counts, and calls parse to perform the parsing of this *TeXcode* object. The counts are added directly into the *TeXcode* object, so only the *TeXcode* object reference is being passed around.

In the CGI version, parse is called directly from MAIN since only one document can be parsed and no file inclusion is possible.

2.6 LATEX code parsing by parse

The parse routine takes a *TeXcode* object and parses this to the end. It is, however, only the entry point for parsing the LATEX code: other routines do the main parsing with _parse_unit being the main work horse. In fact, parse only initiates the parsing, calling _parse_unit repeatedly until the end of the file.

The _parse_unit routine is used to parse one unit or block of LATEX code: a unit/block can be the a part of the document enclosed in e.g. {...} or \begin...\end, or based on context enclosed by e.g. [...], or the document at the top level. It is passed the *TeXcode* object to parse, a parsing state instructing it how the block should be parsed, and optinally a block-end token which tells _parse_unit when the block ends. The _parse_unit routine is then called recursively whenever a unit/block is encountered that requires a separate parsing state or closing token.

The parsing state indicates if the block is part of the main text in which words should be counted, a header, equation contents, should be excluded, etc. and is the only state variable of the parser. In addition to the regular states with which the LATEX code is parsed, there are transition states. E.g. $STATE_TO_HEADER indicates that the block should be counted as a header and the contents should then be parsed using $STATE_TEXT_HEADER as specified in %transition2state.

The document is tokenized, and _parse_unit retrieves one token at the time by calling next_token. Depending on the active parsing state and token, different rules (most with their own subroutines) are applied. These rules add to the *count* object of the *TeXcode* object by calling _inc_count and set the presentation style of the verbose output included which tokens to print. The active token and its style is by default stored in the *TeXcode* object and printed to the verbose output by next_token upon retrieving the next token, although this is occasionally overrun by calls to e.g. flush_next.

When _parse_unit encounters a new block/unit, it will determine the state with which this unit should be parsed based the present state and the context that defines the unit.

2.7 Summary statistics

The counts are stored in the *TeXcode* object: subroutines performing the actual parsing increments the appropriate counter upon processing the parsed tokens. The *TeXcode* object contains a main *count* object from which summary output is generated. The *count* object can also contain a list of subcounts, themselves *count* objects, which may also be presented in the summary.

Depending on options set and the number of files parsed, summary output can range from a single number of the total word count, to an extensive summary for each spcified file with separate summaries for each included file, as well as a total summary.

3 Global constants and variables

There are a number of global variables defined at the start of the script for storing options and settings as well as global counters.

In addition, there are sets of global constants, as well as other globally defined variables, hashes and arrays. Here, we outline the main groups.

3.1 Global constants

There are a few sets of global constants. The use of global constants makes the code more readable. The sets of global constants are:

$STATE_...: Parsing states, e.g. $STATE_TEXT for parsing LaTeX code as regular text and $STATE_IGNORE for regious that should not be counted.

$CNT_...: Index pointing to the location in the counter array used for a specific count, e.g. $CNT_WORDS_TEXT=1 indicating that words in text are counted in position 1 of the array.

$TOKEN_...: Token types, e.g. $TOKEN_WORD and $TOKEN_MACRO. When a token is parsed, the *TeX-code* object stores the token type as well as the token, which can then be used to determine how the token should be interpreted.

3.1.1 Counter indices: $CNT_...

The *count* object contains an array with the following counts: number of files, number of words in text, number of words in headers, number of words in captions, number of headers, number of floating object-s/tables/figures, number of inline equations, number of displayed equations. Storing these are the main purpose of the *count* object.

Each count has a fixed position in the array, and the $CNT_... constants provide the positions of each count: e.g. $CNT_WORDS_TEXT=1 indicates that the counter for words in the text is stored in position 1 of the array. Originally, these positions were hard-coded and directly related to the parsing states, but by using these constants, and keeping the counter indices distinct from the parsing states, the code becomes both more readable and more flexible in case of future changes.

3.1.2 Parsing states: $STATE_...

The parsing states fall into two categories.

First there are parsing states used during the parsing of a unit/block: e.g. $STATE_TEXT, $STATE_MATH, $STATE_IGNORE, ldots. In some of the states, words are counted either as text words, header words or captions words; in other states, words are ignored and the state primarily influences how the parsed LaTeX code is styled in the verbose output.

Secondly, there are transitional states: e.g. $STATE_TO_HEADER which indicates the start of a header which should first cause the header count to be incremented and then the contained text to be parsed as header text using the parsing state $STATE_TEXT_HEADER. The handling of the transitional states are encoded in %transition2state and performed by the transition_to_content_state routine which is called by _parse_unit.

Macro handling rules specify how many parameters the macro takes and which parsing states are used to parse each parameter; for environments, it additionally specifies a parsing state for the contents of the environment.

Originally, before implementing the $STATE_... constants, fixed numerical values were hard coded into the Perl code, and these numerical codes were required for adding new rules. For the macro rules specified

within the Perl code of TEXcount, the original numerical codes remain in the initial rule specification. However, from version 2.3 of TEXcount, the intention is that users should no longer use these numerical codes to specify new macro handling rules, but instead use a set of keywords: e.g. text, header, ignore, etc. For this purpose, a hash %key2state is defined which maps keywords to parsing states. The original numerical codes are included in this map in part for backward compatibility, but also because this key-to-state map is applied to the macro handling rule hashes %TeXmacro, %TeXenvir, etc. The %key2state has is set up e.g. with

```
add_keys_to_hash(\%key2state,$STATE_TEXT,1,'word','w','wd');
```

which maps the keys 1, 'word', 'w' and 'wd' all to the value $STATE_TEXT (which need not be 1!). However, this specification, which is used to convert keywords to states during initialisation of the macro handling rules and later if adding new rules, ensures that the original numberical codes will be handled as before: TEXcount will be backward compatibile with respect to using the numberical codes to add new macro handling rules through %TC commands.

Although in theory the parsing state numerical codes could be changed without any effect to the code, there are still a few places where the actual numerical values are used: e.g. the routine state_to_text.

3.1.3 Token types: $TOKEN_...

When the LATEX code is tokenized, i.e. the string containing the LATEX code is converted to tokens like words or macros, not only is the token stored in the *TeXcode* object, but a token type is stored as well indicating if the object is a word, macro, space, symbol, bracket, etc. To make the Perl code more readable, these token type, although just integer values, are represented by constants $TOKEN_....

When _parse_unit parses the LATEX code, it frequently uses the token type stored in the *TeXcode* object rather than the token itself to determine how to interpret the parsed tokens.

3.2 Option alternatives

Some options result in choosing between a number of alternatives for parsing, counting or presentation. These alternatives tend to be defined in arrays or hashes. When an alternative is selected, the corresponding value(s) are copied to a variable, array or hash which may then later be applied or further processed.

%BreakPointOptions: For keywords like section or chapter, this defines which macros indicate a new break point (i.e. initiates a new subcount).

%STYLES, %STYLE: The %STYLES hash contains different sets of style definitions, used to define the style with which tokens are printed, and are used to set the %STYLE hash by Apply_Options after the options have been processed. Each value of the %STYLES is a hash mapping style name to ANSI colour styles. For a given style, only style names defined in the style are printed in the verbose output. If ANSI colour coded output is used, these are the colour codes; otherwise, the ANSI colour styles are not themselves used, but the style name must still be included in the hash to enable the token to be printed.

%NamedLetterPattern, $LetterPattern: Named regex patterns are defined in %NamedLetterPattern where the selected pattern is stored in $LetterPattern. This regex pattern defines what is recognized as letters when parsing LATEX code.

%NamedWordPattern, @WordPatterns, $WordPattern: Named word patterns are defined in %NamedWordPattern. The selected patterns are stored in the array @WordPatterns. Letters are indicated by a special character, and when the options are applied replaced by $LetterPattern and merged into a single regex stored in $WordPattern.

%NamedMacroOptionPattern, $MacroOptionPattern: Named regex patterns are stored in %NamedMacroOptionPattern, and the selected pattern copied to $MacroOptionPattern. This pattern is used to recognize macro options which should be excluded from word counts.

%NamedEncodingGuessOrder: For each named language, this gives an array of encodings to try if none is given.

4 Details of the TEXcount objects

These objects are simply hashes that are created with a given set of keys. Some keys may, however, be optional.

4.1 The *Main* object

The *Main* object is used instead of the *TeXcode* object to capture errors and warnings. It is created by the getMain routine. The values (keys) it contains are:

errorcount: Numerical value, initialised to 0, used to count the number of errors reported.

errorbuffer: Array, initialised to an empty array, used to buffer error messages reported before output is available: e.g. before the header or HTML header has been printed.

warnings: Hash, initially empty, used to store warnings.

When errors are reported through calls to error, they will be stored in the errorbuffer if this exists, otherwise printed immediately. This is used to store errors reported before e.g. the HTML header has been written. After the appropriate headers have been written and the output channel is ready for writing, a call to flush_errorbuffer is made which prints all the errors in the errorbuffer and then deletes it so further errors will be printed immediately rather than buffered.

4.2 The *TeXcode* object

The *TeXcode* object is used to encapsulate the LATEX code and corresponding counts. It is created by the TeXcode routine. The values it contains are:

filename, filepath: The name and path of the parsed LATEX file.

PATH: An array containing the paths to search for included documents. At creation, this is empty, but calls to _add_file will set it; the top level files, initiated from parse_file, will have this set to $workdir.

texcode: Initialised with the LATEX document as a single string. If included files are to be inserted into the document, they will be inserted into the texcode string.

texlength: Counts the total length (in characters) of LATEX code. Initialised with the length of the LATEX document. If included documents are inserted, their length is added to texlength.

line: Initialised to an empty string. During parsing, segment by segment (one paragraph at a time) is moved from texcode to line. Tokens are then read and subsequently removed from line.

next: Initialised to undef, this stored the next token to be processed. Upon tokenization, the token is identified and removed from the start of line and moved to next.

type: Initialised to undef, this contains the token type ($TOKEN_...) of the next token.

style: Initialised to undef, this is used to set the style with which the next token should be presented in the verbose output.

printstate: Initialised to undef, this is used output the active parsing state for use with verbose output (if $showstates is set).

eof: Initialised to 0, this is set to 1 once the end of the document is reached.

countsum: The contains the main *count* object.

subcount: This contains the present subcount which is also a *count* object. These subcount are used to count e.g. section and chapters of the document.

errorcount: Initialised to 0, used to count the number of errors reported during the parsing.

errorbuffer: Undefined at initiation, indicating that errors should be printed instantly rather than stored for later printing. Can be defined as an array which is then used to store error messages so they can be printed later.

warnings: Hash used to store warnings.

When the *TeXcode* object is initialised, the LaTeX document is placed as a single big string in texcode. During parsing, next_token is called on to return the next token, which in turn it delegates to _get_next_token. Instead of operating on the whole document, which was done in older version of TeXcount and was quite slow on large document, more_texcode is called on to move segments (i.e. paragraphs) of LaTeX code from texcode to line, and then it grabs one token at a time from line. This is when next and type are set.

When the tokens are interpreted and counted, inc_count is called which increments the appopriate counter in subcount. If a new subcount is initiated, a call to next_subcount adds subcount to sumcount, including appending the subcount object to the list of subcounts stored with sumcount, and then replaces subcount with a new *count* object.

4.3 The *count* object

The *count* object is used to store the word and text element counters. It is created by new_count. The values it contains are:

title: A string set upon creating to contain a descriptive title of the count.

counts: An array, initialized with 0s, which is used to store the counts. The size of the array is determined by $SIZE_CNT and should reflect the number of $CNT_... indices defined.

subcounts: This is an array, initialised to an empty array, used to store the subcounts.

In addition to the default fields, when used as the sumcount field of a *TeXcode* object, a few additional fields are added:

TeXcode: This is a reference pointing back to the *TeXcode* object in which it is contained.

5 LaTeX code parsing and interpreting

The entry point for parsing a LaTeX document is the parse routine. This simply calls _parse_unit repeatedly using parsing state $STATE_TEXT until the end of the document is reached. Thus, _parse_unit is the main routine for performing the actual parsing.

The _parse_unit routine is called with a *TeXcode* object, a parsing state, and optionally an unit-ending token as arguments. It then calls next_token on the *TeXcode* object until the unit-ending token is reached: if the file ends before this is found, an error is reported. If no unit-ending token is provided, only one unit will be parsed. If the unit-ending token is set to $_PARAM_, indicating that the unit to be parsed is a macro parameter, the $simple_token flag is set and passed to next_token to avoid combining letters into words, and only one token is parsed before returning.

For each token, depending on the token, token type, and active parsing state, _parse_unit decides how the token should be interpreted. In some cases, the interpretation is done within _parse_unit, but in many cases the interpretation is delegated to subroutines like _parse_macro, _parse_math, etc. If new groups ({...} or \begin...\end) are encountered, this causes _parse_unit to be cause recursively with an unit-ending token passed to _parse_unit to identify the group end.

Note that by default, even blocks that are to be ignored are parsed and required balanced units. Different exclude states exist to deal with cases in which the unit should not be completely parsed.

Upon interpreting the parsed tokens, _parse_unit or the subroutines to which it delegates the interpretation control the counter incrementation as well as how the tokens are presented in the verbose output. The counter incrementation is done through calls to inc_count passing as arguments the *TeXcode* object, the appropriate count reference ($CNT_...), and optionally a number if the counter should be increased by a number different from 1. Specifying how the token should be presented in the verbose output is done by deciding on the style, usually set using set_style: the styles are represented by strings that give the style name, which are the same as used as keys in %STYLE and as styles in the HTML output.

If a style for presenting a token is selected which is not in the %STYLE hash, the token is not printed. Thus, the %STYLE hash also filters which tokens are printed to the verbose output.

5.1 Tokenization and token handling

The routine for retrieving the next token is next_token. This first makes sure that the previous token gets printed to the verbose output with the style specified by set_style. It then calls _get_next_token to retrive the next token: this will process comments and line breaks itself until a token is retrieved that it returns.

The _get_next_token routine checks the line field of the *TeXcode* object to determine which is the next token in line. If the line field is empty, it calls more_texcode to move the next segment of LaTeX code from the texcode field of the *TeXcode* object to line. When it has decided on the approriate kind of token, removing it from the start of the line field in the process, it sets the next and type fields of the *TeXcode* object through calls to __set_token or __get_token (for single character tokens).

If the optional $simple_token flag is set, only simple tokens will be returned: i.e. letters will not be combined into words. This is used for parsing macro parameters.

5.2 Processing parameters and options

In _parse_unit, based on the parsing state and parsed token, it is decided how to interpret and process the token. In some cases, this processing is restricted to the parsed token itself: counting or ignoring it as well as deciding on the style with which it should be presented in the verbose output.

In some cases, the token influences the parsing of subsequent text: e.g. macros can take parameters and options. Special subroutines exist to handle parsing of macro parameters, gobble up spaces or macro parameters, or handle ignored regions.

5.3 Verbose output

By default, all parsed code is processed for printing to the verbose output. If it actually gets printed or not depends on whether the set style is included in the %STYLE hash or not.

Upon parsing a token, it is stored in the next field of the *TeXcode* object. If set_style is called during processing, this will set the style field of the *TeXcode* object, but will not itself print the token. The flush_next routine is used to print the next token using the style set in the style field, or provided in the call; this in turn calls print_style which is responsible for the printing. There is an automatic call to flush_next when the next token is retrieved, ensuring that all tokens are sent off for printing. When flush_next is called, the style field is set to $STYLE_BLOCK='-' which blocks further printing (or change in style) of the token; the style field is then set to undef by next_token upon reading the next token.

The tokens are passed to print_style, either directly from the parsing or via next_token, which looks up the style in the %STYLE hash. Only tokens whose style is defined in the %STYLE hash get printed. If colour coded output to text is set, the values %STYLE are used with the ansiprint function to print the token using ANSI colour codes. If output to HTML is chosen, the token will be printed enclosed in a tag using the style as class; the HTML style definitions are then used to determine how these elements will be displayed.

Special style values are $STYLE_EMPTY=' ' which is used for spaces and must be defined in the %STYLE for spaces to be printed, and the $STYLE_BLOCK='-' style value which is not actually a style but a value used to mark that the token has already been printed and block further printing of it.

In addition to the %STYLE hash which specifies which tokens get printed, there is a global variable $printlevel the value of which is taken from the %STYLE which is used to control if verbose output is on (1 or 2) or off (0 or -1). The -1 values indicates the quiet mode in which errors should not be printed; the value 1, as opposed to 2, indicates that multiple ignored lines should be collapsed to make the verbose output more compact, although this is only partially done.

The routines for handling tokens, styles and verbose printing remain from the earliest version of TEXcount and has not undergone much improvements or cleaning up and remains somewhat unstructured. Hence, there may be stray calls to e.g. set_style that no longer have any effect.

6 Regex patters: letters, words, macro options

One of the most important regex definitions in TEXcount is that used to recognize words. This is done in two steps: first a regex for letters is produced, and then this is combined with patterns for words to generate one big pattern.

Another regex defined is the one used to recognize macro options, i.e. [...], that appear together with macros and which should be ignored.

One reason behind the desire to generate one big pattern rather than loop through alternative patterns is to enable Perl to compile each pattern just once. The pattern compilation typically takes longer than the pattern matching, so this can make a big difference.

6.1 The word regex

First note that TEXcount distinguishes between alphabetic words, i.e. words composed of letters, and logograms (e.g. Chinese characters) which are counted per character. When words (or letters) are counted, these are made from characters defined as alphabetic; characters defined as logographic are counted separately character by character.

The regex pattern recognizing a letter is placed in $LetterPattern. This is usually taken from one of the optional patterns in %NamedLetterPattern, but can be modified elsewhere or replaced by undef to signify that no words or letters should be counted.

A number of regex patterns which should be recognized as words are place in the array WordPatterns. This is usually set by using one of the named lists of word patterns defined in %NamedWordPattern, but can be redefine or modified by options. In the word patterns, the character is used to represent a letter, and this is later replaced by $LetterPattern when the options are applied.

After parsing the command line arguments, the options and settings are applied. At this point, through apply_language_options, $LetterPattern is applied to WordPatterns, which are then combined into a single regex: $WordPattern. At this point, patterns for recognizing logograms are also added.

6.2 The macro option regex

After macros and macro parameters, macro options on the form [...] will be ignored. There is a single regex used to recognize and remove these macro options.

For most uses, macro options tend to be short codes which are easily recognized. However, there are also cases where the macro options can be more complex. On the other hand, there are also cases where brackets are used without being macro options, and it is vital that these cases should not be mistaken for macro options: in particular if they contain text that should be counted.

In order to capture most macro options as options without running a risk of ignoring actual text enclosed in brackets, restrictions are placed on what can go inside macro options. The default rule is moderately strict, but can be relaxed to allow more extensive and general macro options.

The different macro option regex patterns are named in %NamedMacroOptionPattern and copied to $MacroOptionPattern when initialised or changed by options.

6.3 Unicode character classes

The user can specify which character classes should be considered alphabetic (i.e. letters) and which should be considered logographic (i.e. counted as indicidual characters). Typical alphabetic characters are the Latin letters. Typical logograms are the Chinese characters. If any of the language options are used, these character classes will automatically be set.

Specifications of alphabets and logograms are done by options -alpha= and -logo= using Unicode character classes. Unicode classes include Latin, Digit, Ideographic, Han, etc. Note that all Unicode character classes start with capital letters.

6.4 Custom made character classes

Some of the Unicode character classes are not defined quite as desired by TEXcount. In particular, the Alphabetic character class includes Ideographic, which would cause e.g. Chinese characters to be allowed as parts of words together with Latin characters rather than force them to be counted as individual characters. To resolve this problem, new character classes are defined in TEXcount that fit our need.

New character classes can be defined within TEXcount through subroutines named Is_*name*. Most notable is the Is_alphabetic character class from which the logographic characters have been excluded. This is now used as the default alphabetic character class.

Presently defined characters classe are named digit, alphabetic, alphanumeric, punctuation, cjk, cjkpunctuation. Note that these are all lower case, and have the prefix Is_ added when referred to in the code.

When adding character classes to the set of alphabetic or logographic characters using -alpha= or -logo=, the names without the prefix Is_ may be used: for character classes starting with a lower case letter, the prefix is added automatically.

Note that the subroutines specifying the character classes must be defined prior in the code to any use: this is unlike other subroutines which may be defined anywhere in the code. Also, to be permitted as character classes by Perl, the subroutines must start with Is_ (or In_ although that is not used by TEXcount), although different versions of Perl need not enforce this.

7 Macro handling rules

While some rules for handling macros are hard-coded into TEXcount, most of the rules are stored in a number of hashes which TEXcount look up whenever a macro is encountered. The general rule is that the keys are either macros (e.g. '\section') or environment names (e.g. 'quote').

%TeXmacro: The keys are macros, or 'begin*name*' where name is an environment name, and the values specify how many parameters the macro (or environmemt) takes and how these should be processed. See the section on parameter handling rules further down.

%TeXenvir: The keys are environment names, and values are the parsing state with which the contents of the environment should be parsed.

%TeXpreamble: These are macro handling rules to be applied in the preamble, i.e. after \documentclass but before \begin{document}. The rules are specified as for %TeXmacro.

%TeXfloatinc: These are macro handling rules to be applied within floating bodies, i.e. tables and figures.

%TeXmacroword: The keys are macros, and the values are numbers representing how many words the macro generates. This is used for macros like %LaTeX which generates text.

%TeXpackageinc: The keys are macros used to include packages. Although included in %TeXmacro, the processing of package inclusion is actually performed by _parse_include_package independent of the hash value. The value should therefore be 1 or [$STATE_IGNORE] since this is how it will be processed by _parse_include_package.

%TeXfileinclude: The keys are macros used to include LaTeX files into the document, the value a keyword or list of keywords telling how file names and paths should be interpreted. Processing of these macros is done by _parse_include_file.

Note that the definition of %TeXmacro starts by including %TeXpreamble, %TeXfloatinc and %TeXpackageinc. After that, the values of %TeXpackageinc are never used. For %TeXpreamble and %TeXfloatinc, however, it is in principle possible to rules within the preamble and floats, respectively, that are different from those defined in %TeXmacro and applied elsewhere in the document.

7.1 Parameter handling rules

A macro can be specified to take a given number of parameters: this will typically be {...} blocks following the macro. For each of these parameters, a separate parsing state can be specified. This is represented by an array with one element for each parameter, the elements being the parsing state ($STATE_...) with which that parameter should be parsed.

In addition to the $STATE_... rules are some modifier/option states, $_STATE_.... The $STATE_OPTION states indicates that the next rule in the list is an optional parameter enclosed in []. By default [] options are ignored, which can be swithed off by $STATE_NOOPTION or on by $STATE_AUTOOPTION.

An alternative specification of a parameter handling rule is to give the number of parameters to ignore. TeXcount will check if the specified rule is an array (as described above) or a number and interpret the rule accordingly.

The hashes %TeXmacro, %TeXpreamble and %TeXfloatinc all take values that are this kind of parameter handling rules, as are q%TeXpackageinc since they are included in %TeXmacro.

Throughout the script, parsing states are referred to using the $STATE_... constants. In previous versions, however, these codes were hard-coded into the script and used both to set up the hashes and to specify new rules through %TC instructions. For backward compatibility, the old numerical state codes remain in the conversions from keywords to $STATE_... constants as stored in %key2state and applied through calls to convert_hash accompanied by keyarray_to_state or key_to_state.

7.2 File inclusion and the %TeXfileinclude hash

The main LaTeX commands for file inclusion are \input and \include, while \bibliography includes the .bbl bibliography file. However, additional packages exist that can also modify the file search path, of which TeXcount has support for the import package.

File inclusion macro rule are stored in the %TeXfileinclude hash. The values are strings which contain one or more keywords (separated by space or comma):

input: This is a special keyword to use with \input. The handling of the parameter values is as file, but the parameter itself is not required to be enclosed in {}.

file: This parameter simply gives the name of or path to a file. If the file is not found, TeXcount will append .tex and try again.

texfile: This parameter gives the name of or path to a file, but .tex will be appended, and is the rule used by \include.

dir: This parameter provides the path of a directory relative to the $workdir, and adds this to the search path before including any files. This is used with the \import macro of the import package.

subdir: This parameter provides the path of a directory relative to the current directory, and adds this to the search path before including any files. This is used with the \subimport macro of the import package.

<bbl>: This is a special keyword to use with \bibliography to specify inclusion of the bibliography file.

The parsing of the macros and parameters is done by _parse_include_file. For each keyword it parses a parameter, unless the parameter is on the form <*keyword*>. The parsing of the input parameter is handled

differently from the rest since it need not be enclosed by {}. It then delegates the processing of macro inclusion rules to include_file.

In include_file, the file is located (based on search path) and either appends the file to the @filelist array of files to be include, or merged immediately into the document by a calls to read_binary and prepend_code.

The @filelist array contains elements which are themselves arrays on the form [file,path,...] where the first element is the path to the file to be included, and the remaining elements are the search paths used to set the PATH values of the *TeXcode* object. For the top level files, i.e. the ones specified on the command line, the search path will contain only $workdir: the directory from which TEXcount is executed unless -dir is used to specify otherwise. If more directories are added to the path, $workdir will remain the last directory of the search path, while the first directory of the search path will be considered the current directory.

File inclusion macros can also take parameters that should be parsed using regular macro parsing rules. The TeXmacro hash will be checked for @pre\macroname and @post\macroname entries which will be applied before and after the file handling rules.

7.3 Package and document class specific rules

Whenever TEXcount encounters a package inclusion, it will check for package specific rules. These are defined in hashes names %PackageTeXmacro etc. which maps the package name to the hash map of rules to be added to %TeXmacro etc. There is an additional %PackageSubpackage which for each package name in the set of keys maps to a list of packages whose rules should automatically be included.

Similarly, rules specific to particular document classes may be implemented by using the key class%*name* instead of the package name, and these will then be added to the set of parsing rules if \documentclass{*name*} is encountered.

Note that rules for including the bibliography is also stored in these hashes under the key %incbib.

8 Presentation of summary statistics

The counts (words, headers, etc.) from a LATEX document are stored as a *count* objects. The main routine for printing the summary statistics from a *count* object is print_count: the routine conditional_print_total which is called from MAIN delegates printing to print_count except if the brief output format is selected. The print_count routine then delegates the printing to one of a number of subroutines depending on the settings.

Word frequencies are store globally in %WordFreq. This gets incremented each time _process_word is called. Summary of word frequencies are produced and printed by print_word_freq which tries to combine words that differ only by capitalization, and also produces subcounts per character class.

A global count of the number of errors reported is stored in $errorcount, while warnings are stored globally in the %warnings hash mapping when added through the warning routine with the warning as key and the number of occurrences as value to ensure each warning is only listed once no matter how many times it is reported. Both warnings and errors are also stored in their respective *Main* or *TeXcode* objects when reported through calls to error or warning.

In MAIN, after processing of the LATEX documents, Report_Errors is called to give a total report on errors and warnings. The exact output depends on the settings.

NB: Processing of errors and warnings requires some improvement. Now, parts of the code handle errors per file, others do so globally.

9 Encodings

The preferred encoding is Unicode UTF-8. From version 2.3 of TEXcount, this is used internally to represent the LATEX code, and Unicode is relied upon to handle different character sets and classes.

When files are read into TEXcount, they may have to be decoded from whatever encoding they are saved in into UTF-8. The file encoding may be specified explicitly using the -enc= option, otherwise TEXcount will try to guess the appropriate encoding.

The output from TEXcount is be default UTF-8. However, if a file encoding is specified using -enc= and output is text, not HTML, this encoding will also be applied to the output. This may be useful when using TEXcount in a pipe, otherwise the documents will be converted to UTF-8.

10 Help routines and text data

A hash, %GLOBALDATA, and hash reference $STRINGDATA are is defined for storing strings used for various outputs. The %GLOBALDATA is set up containing string constants for version number, maintainer name, etc., while $STRINGDATA is initially undefined.

The $STRINGDATA hash is accessed through calls to StringData which initialises the hash if undefined. Initialisation, which is done by STRINGDATA, reads through the __DATA__ section at the end of the script, identifies headers which are used as keys in the hash which maps to and array containing the subsequent text lines. References in the read text on the form ${keyword} are replaced by the corresponding string in %GLOBALDATA: this allows e.g. version information to be inserted into the text.

Headers in the text data consists of three or more colons followed by space(s) and a keyword. Lines containing three or more colons but no keyword have no effect.

Lines starting with are used to format output printed by wprintlines. The two characters '-' and ':' can then be used to indicate indentation tabulators, and subsequent lines will be indented and wrapped: this is used for printing help on command line options. The wprintlines also wraps text: the page with is set by $Text::Wrap::columns.