Bitrans – Bi-directional Translation/Substitution Tool – User Manual

R.Zandbergen

Issue 1.5 , 06/12/2025

Contents

1 Purpose

1.1 General

This document describes a tool (bitrans) that can perform text substitutions to any plain text file.

In addition, this tool provides special support for files in the IVTFF format. For the definition of this format, see Reference [R-1].

1.2 Version

This is version 1.5 of the document. It refers to bitrans software version 1.5.

Software version 1.1 emulated the main functionality of the original bitrans written by Jacques Guy in the 1990's.

Software versions 1.3 and higher support, in addition to this, so-called homophonic substitutions.

Software version 1.4 supports Unicode (UTF-8 encoding) both in input and output. It also has a minor improvement in rules file parsing which means that in some cases for context-dependent substitutions the result of using bitrans changes between versions 1.3 and 1.4 (see Section 0). Finally, it allows the user to place separators in the rules file, such that rules are interpreted in blocks (see also Section 0).

Software version 1.5 allows users to verify whether all text of an input file has been substituted, and removes a few bugs.

1.3 References

[R-1] R. Zandbergen: IVTFF – Intermediate Voynich MS Transliteration File Format. Issue 2.0.2 of 08/07/2025. Available via: https://voynich.nu/software/ivtt/IVTFF_format.pdf

[R-3] ivtt user manual, issue 2.4 of 07/09/2025. Available via: https://voynich.nu/software/ivtt/IVTT_manual.pdf

[R-4] Web page: https://voynich.nu/transcr.html

[R-5] Web site: https://voynich.nu/

[R-6] "Read-Me" file for bitrans and ivtt software: https://voynich.nu/software/000_README.txt

2 Introduction

2.1 General

bitrans may be used to convert text files from one character representation to another. It can process simple Ascii or UTF-8 text files, and it can take into account that certain parts of the text are to be left unchanged, by declaring 'comment' delimiters. Thus, it can also process files that have meta-data in comments, such as Voynich transliteration files in the IVTFF format. In that context, it may be used to take a Voynich MS transliteration file in one transliteration alphabet, and convert it to another transliteration alphabet. However, this is not the only use, and it has additional options that make it a flexible substitution tool.

It is invoked by the command:

```
  bitrans
```

followed by a number of arguments. Arguments starting with a minus sign are treated as options. Other arguments are treated as file names.

It reads a transliteration file (or other text file) from standard input or a user-named file, writes the output (processed) file to standard output (or again a user-named file) and it will write some information to standard error output. It may write some additional information to specific named log files.

To do this, it also needs to read a file that defines the translation rules, which will be called the rules file in this document. The format of this rules file is explained in detail in Section 0 below. By default, the rules file will be read from a file named:

```
       bit_rules.txt
```

(The user may specify another rules file using a command line option).

To explain the use of bitrans, we may start by giving a few simple examples.

2.2 Simple example #1

While the full definition of the rules file is given later, at this point it is sufficient to know that, in its simplest form, it consists of a single header record followed by two columns, one containing source or input tokens, and the other output tokens. We will use this example:

```
##BIT1
ea   Q
ee   Q
```

This file is saying: convert all occurrences of "`ea`" to "`Q`" and also convert all occurrences of "`ee`" to "`Q`".

We also have a text file `example1.txt` with the contents:

```
Eamon is twenty-three years old
```

Then the command:

```
        bitrans example1.txt output.txt
```

Will generate the following output to the user console:

```
Bi-directional translation / substitution tool (v 1.5)

Summary of options:
Translation direction 1 – left to right
No alphabet name check
No substitution completion check
No additional debug output

Input file: example1.txt
Output file: output.txt
Rules file: bit_rules.txt

Reading rules file
Rules file enforces direction 1
  0 comments defined
  2 substitution rules

Analysing   2 substitution rules

Starting...

  1 line(s) processed
```

It will also create the file `output.txt` with the following contents:

```
Eamon is twenty-thrQ yQrs old
```

While this may not seem particularly useful, it illustrates a few points.

First of all, the “`Ea`” at the start is not modified, because the tool is case-sensitive and “`Ea`” is distinct from “`ea`”.

Secondly, the substitution is not reversible, because, for the two cases of “Q” in the output file, we cannot see whether they were originally “ee” or “ea”.

2.3 Simple example #2

In a second example, we will use this rules file:

```
##BIT
e    1
ee   2
eee  3
```

Our input file is:

```
aaa be cee ede eee fef
```

In this case, the output will be:

```
aaa b1 c2 1d1 3 f1f
```

bitrans realises that it should not translate “ee” as “11”, but as “2”, and “eee” not as “111” but as “3”. It does not matter in which order these rules appear in the rules file. Bitrans uses ‘greedy’ substitutions, in that it prefers to substitute the largest possible charcter groups as a priority.

(Of course, if one wishes to translate “ee” as “11” instead, it would be sufficient to leave out the rule “ee   2”.)

In this case (and given that the input file did not already include the numbers 1,2 and 3), the translation is reversible, and bitrans can also do the reverse translation from the same rules file, by using the command line option “-2”, which tells bitrans to interpret the rules file from ‘right to left’ instead of ‘left to right’.

3 Bitrans detailed usage

3.1 General

bitrans is a text substitution tool, comparable in many ways to the general Unix tool `sed`.

The user can specify a set of substitution rules, expressed as two columns of text, and bitrans will replace all occurrences of the characters or characters groups in one column by the items in the other column. It can use the same set of rules in both directions. (This is the reason for "bi" in the name of the tool).

The replacement is not cumulative, which means that a piece of the text that has already been replaced, will not be replaced again due to another substitution rule.

The replacement algorithm is 'greedy', which means that it will always prefer to replace the largest possible group of characters first. This was already illustrated by the example in Section 2.3.

The text is processed in a line-by-line manner, and multiple replacements from the same substitution rule in one line are made by processing the line from left to right. Referring to Example #2 above, had the input file included the string "`eeee`", then the output would be: "`31`". First, the initial "`eee`" is substituted, after which only one "`e`" is left for substitution.

3.2 Preventing substitution of some parts of the text

The text file may have areas that should not be affected by the translation. These may be comments or meta-data. bitrans recognises two methods for identifying such areas:

1. Entire lines that start with a single 'comment character', i.e. 'full-line comments'
2. Parts of lines that are delimited by a pair of characters, i.e. 'limited extent comments'

Typical examples are:

- Lines starting with `#` should be left unchanged
- Text contained between `{ }` braces or `< >` tags should be left unchanged

The user may specify up to six such comment definitions, which may be any combination of the above two types. These should be included in the rules file. See Section 0 for the syntax.

3.3 Specification of Unicode characters

bitrans supports Unicode characters in UTF-8 format of up to three-characters, i.e. up to hexadecimal value FFFF. Such characters may be specified in the rules file, both as input tokens or output tokens, by using a 4-character hex code preceded by ampersand (`&`) and followed by a semi-colon (`;`). In the 4-character hex codes:

- Leading zeroes shall be used as necessary
- The alphabetical part shall be in upper case only (`A-F`)
- Any `&` or ; character that is not part of such a combination will be taken as a literal character
- Any pair of `&` and ; characters that are separated by four characters must be a valid Unicode
- Single tokens in rules files may include several Unicode characters

Examples:

`&abc;`	will be treated as plain text
`&01AB;`	will be treated as a Unicode definition
`&20xy;`	will generate an error

The usage of UTF-8 codes may be explained with example #3.

The following rules file:

```
##BIT
Ae   &00C4;
ae   &00E4;
Oe   &00D6;
oe   &00F6;
Ue   &00DC;
ue   &00FC;
```

when applied to the text:

```
Oetzi hat oefters ueble Traeume
```

will convert it to:

```
Ötzi hat öfters üble Träume
```

Beside this very simple example, this feature can be used for transliteration (Romanisation) of complete texts in languages like Russian, Hebrew or Arabic. It requires an initial effort to set up the corresponding rules files, but once that has been done, these can be reused any time.

As a final comment, for substitution prioritisation (as explained in Section 2.3), UTF-8-encoded Unicode characters count as single characters, even if they occupy two or three bytes in the input text.

3.4 Context-dependent substitution

bitrans allows a limited method for making context-dependent substitutions. The context in this case may be either ‘start of word’ or ‘end of word’. This is achieved by including the # marker in the substitution rule, where this marker represents the word boundary.

This word boundary is defined as any of:

- start of line
- end of line
- space character
- period (full stop)
- comma

(Note that this does not include question marks, exclamation marks, or single or double quotes).

In the normal case, such a conditional substitution rule should have one # marker in both elements of the rule.

It is best explained with example #4.

The following rules file:

```
##BIT
 #9     #con
9#    us#
```

when applied to the text:

```
   Equ9 me9 9stipat9 est et incontinens
```

will convert it to:

```
   Equus meus constipatus est et incontinens
```

The first nine is followed by a space (separator) and therefore the second rule applies. The third nine is preceded by a space (separator) and therefore the first rule applies.

Furthermore, the back conversion of the output, using the same rules file, will convert it back to the original text. In this case, the “`con`” included in “`incontinens`” is not affected by the first rule, because it is not preceded by a separator.

If the separator character (`#`) also appears as a character in the input file, it will be copied to the output file unchanged. It will not play a role in any context-dependent substitutions. However, it cannot be part of any substitution rule or any comment definition rule. If that is desired, the user may re-define the separator character as used in the fules file from # to another symbol. How to do this is explained in Section 0.

It should be noted that the separator is not considered to be replaced, as the result of any substitution. This means that the same separator may play a role in two substitution rules, one that concerns the character(s) preceding it, and one that concerns the character(s) that follows it.

It is also allowed to have context-dependent substitution rules that do not have exactly one separator character on both sides. In this case, the following rules apply:

1. If the source string has no separator character, but the replacement string has one or more, then all output separators will be:
    a. A period/full stop in case the file is in the IVTFF format
    b. A space character if it is not
2. If the source string has one separator character, and the replacement string has one or more, then all output separators will be the same as the input separator
3. If the source string has more than one separator character, and the replacement string has one or more, then all output separators will be the same as the rightmost input separator.

Since input files (IVTFF or other) may have a combination of space, period and comma separators, the result of such complicated substitutions may not always be as desired, and they should be used with great care.

The user may use ivtt to easily fix such issues, either running ivtt before bitrans, or after.

For the ivtt user guide, see [R-3].

The following example (example #5) demonstrates these rules for a file that is not in IVTFF format:

rules file:

```
##BIT
i       #I#
#a#     #n#
y#      (us)#y#
```

Input file:

```
whoami a groovy,trendy guy.
```

Output file:

```
whoam I   n groov(us),y,trend(us)  y gu(us).y.
```

An important warning:

The user should be very careful when using rules that modify the number of separators. This is not the intended use of the word boundary marker, and in some situations the result may become unpredictable (even when it is always deterministic).

Among others, the following things may happen:

1. A newly introduced word boundary marker (a ‘space’ character) may play a role in a later context-dependent substitution, and in such a case, the result depends on the order in which the strings are replaced, which is typically unpredictable for the user.
2. A deleted word boundary marker may have played a role in an earlier context-dependent substitution, and again the result depends on the order in which the strings are replaced

3.5 Bi-directional vs. uni-directional

While bitrans can interpret any rules file in both directions, it is possible that the combined rules in a rules file are not reversible. Section 2.2 already provided an example of this. Since two source strings map to the same output string, the back conversion is not possible. This rules file can only be interpreted from left to right, but not from right to left.

The user may indicate in the header of the rules file whether it is bi-directional, or can only be used in direction 1 (left-to-right) or direction 2 (right-to-left).

For each bitrans execution, bitrans compares the direction requested by the user (as per command line options) with the header in the rules file, and will stop with an error message if the rules file does not allow it.

Furthermore, it is possible that the rules in any rules file are conflicting in one or the other direction, but this is not indicated in the header. In such a case, bitrans will detect the problem and equally stop with an error message.

Finally, it may be that the rules are reversible, while the translation of a text file is not reversible.

The rules are reversible (and bi-directional) if they don’t include any ‘many-to-one’ substitutions.

A good test of a bi-directional rules file is to apply it to a source text, then apply it in reverse direction to the output, and compare the result with the original source text.

This test could fail, even if the rules file is correctly bi-directional. This would happen, for example, if a rules file requests the translation of “`a`” to “`xy`”, while the input text includes occurrences of both “`a`” and “xy”. In this case, the back translation would also affect occurrences of “xy” that were not the result of the first translation.

This is not a problem or bug in the tool, but something that the user should be aware of.

3.6 Support of Voynich transliteration files in IVTFF format

In case the input file to be translated is a file in the IVTFF format, bitrans will detect this from the header record, which must start with the character sequence “#=IVTFF”.

It will then detect the transliteration alphabet that has been used in this file from the IVTFF file’s header record.

In this case, the rules file used to transform this file should have two properties.

First, it should have two comment definition records, as follows (see also Section 0):

```
#(comment)
<(comment)>
```

This allows to leave all meta-data in the transliteration file unchanged.

The header record of the rules file may also specify the transliteration alphabets of the two columns (see header record definition in Section 0). If this is the case, bitrans can substitute the corresponding four-character code in the output file. The precise action depends on whether the user has specified the ‘strict’ option (-s).

If the user has specified the ‘strict’ option:

- If the input file is IVTFF format, the header in the rules file (input column) must match the alphabet code of the IVTFF file.
    - In case it does, bitrans will write the alphabet code of the output column to the output IVTFF file header
    - In case it does not, it is an error and bitrans will stop
- If the input file is not IVTFF format, the ‘strict’ option plays no role and is ignored

If the user has not specified the ‘strict’ option:

- If the input file is IVTFF format, the heading in the rules file (input column) will be compared with the alphabet code of the IVTFF file.
    - In case it matches, bitrans will write the alphabet code of the output column to the output IVTFF file

- In case it does not match, bitrans will write a warning and continue, leaving the alphabet code in the IVTFF file unchanged

In all this, an empty or missing alphabet code will be considered to exist of four space characters, both in the IVTFF file and in the rules file.

Note that bitrans follows the same logic with files that are in the VDBTF format, which is also indicated in their header records. This format is not documented publicly, so no further information is provided here.

Example:

Assume that the input file is a Voynich MS transliteration using the Currier alphabet.

Its header record could say:

```
#=IVTFF Curr 2.0 M 1
```

Now our rules file defines the translation from Eva to Currier, and it could start as follows:

```
##BIT  Eva- Curr
#=%
#(comment)
<(comment)>
a      A
y      9
in     N
iin    M
ch     S
sh     Z

... (etc) ...
```

In this case, the commands:

```
bitrans -2
```

```
bitrans -2 -s
```

would both correctly translate the file from Currier to Eva, replacing the alphabet name by "`Eva-`".

However, the command:

```
   bitrans -1 -s
```

would result in an error message and the tool would stop.

The command:

```
   bitrans -1
```

would only generate a warning, and complete successfully, but it would not create a useful output file.

Finally, the question whether the input file is in IVTFF format or not, affects some specific cases of context-dependent replacements, for which see Section 3.4 above.

3.7 Random substitution from a list

Bitrans allows the use of so-called homophonic substitutions. This term is used in historical cryptology, and may be somewhat misleading. It has nothing to do with the linguistic meaning of homophones (sounding similar). Instead, it refers to a form of encryption, where a particular plaintext character can be replaced by several different cipher characters, and the person doing the encryption can arbitrarily choose which one to use each time.

A rules file record that says: "`A   11`" will cause bitrans to translate all occurrences of "`A`" to "`11`", when used from 'left to right'. It will translate all occurrences of "`11`" to "`A`", when used from 'right to left'.

On the other hand, a rules file record that says: "`A   11   12`", instructs bitrans to translate any "A" arbitrarily into "`11`" or "`12`", for each individual case.

This means that the 'left to right' substitution is not deterministic, and not predictable for the user.

However, the back substitution ('right to left') is allowed with such a file, and is fully deterministic. All cases of "`11`" are converted to "`A`" and so are all cases of "`12`".

In case a rules file includes homophonic substitution rules, the following line will be added to the run summary:

`Rules file encodes ambiguous definitions` (in case translation is left-to-right)

or:

`Rules file decodes ambiguous definitions` (in case translation is right-to-left)

These homophonic substitution rules can have from 3 to 8 entries. In all cases, the first entry belongs to the 'left' column, while all other entries (2 to 7) belong to the 'right column'. bitrans includes a random number generator that ensures that each of the entries in the 'right column' is generated with equal probability, for each such rule.

In case a rule has more than 8 entries, no error is generated, but the additional entries keep over-writing the eighth entry. (This may be changed in a future version).

While, in principle, homophonic substitution rules are reversible, this is no longer true if among the lists of output strings there are common entries. The following case (excerpt of a hypothetical rules file):

```
A    c  d  e
B    f  g
C    a  f  j
```

is valid when used from left to right, but not reversible (used from right to left), because both “`B`” and “`C`” in an input file may generate an “`f`” in the output.

bitrans also detects such cases and will stop with an error message.

A hard-coded random number generator has been included in the bitrans source code, in order to guarantee that all executions of the same bitrans scenario (combination of rules file and input file) will always lead to the same result, independent of the version of the operating system, the version of the compiler suite, and the date of the run.

The following example demonstrates the use of homophonic substitution rules. While all four lines in the input file are the same, the lines in the output file are all different.

rules file:

```
##BIT
A  a  b
B  c  d  e  f
C  g  h  i
D  j  k
E  l  m  n  o
```

Input file:

```
ABBA CD DECA BEDD ACCA BEDA
ABBA CD DECA BEDD ACCA BEDA
ABBA CD DECA BEDD ACCA BEDA
ABBA CD DECA BEDD ACCA BEDA
```

Output file:

```
afdb gj klhb cmjk bhib coja
bdca ij jnha fmkj bggb elka
bdfb ij klia dljj biga flja
acfb ik klha fmkj bgia cljb
```

3.8 Ambiguous substitution rules

It is easy to end up with ambiguous substitution rules. An example is the following pair of rules:

```
AB --> x
BC --> y
```

These rules are not conflicting, and the principle of Section 2.3 will not impose a priority between the two. As a result, the source text "ABC" could end up being converted into "xC" or "Ay".

The original bitrans tool of Jacques Guy provided a disambiguation mechanism for such overlapping rules. One would define that in all cases the priority is either from the left or from the right. This is not (yet) possible in the present bitrans. In the present version (1.4), the proper solution for this specific example would be to also declare a rule for the input string "ABC". In the present implementation, if two input tokens have the same length, they will be processed in the order in which they appear in the rules file.

There is one clear exception. If any token includes a separator character, then this character counts as 'half' a character. Thus, "#A" takes precedence over "A", while "AA" takes precedence over "#A".

The user has one way to influence the sorting of rules, namely by placing a rules separator record, which consists of six minus signs in positions 1 to 6 of the line. In the above example:

```
AB  x
------
BC  y
```

In this case, the user can be certain that "ABC" will be converted into "xC". The user should be aware that the separator record places a barrier between all rules above and all rules below, and not just the two rules immediately surrounding it. In the case of Example 2 in Section 2.3, this rules file:

```
##BIT
e    1
ee   2
------
eee  3
```

Would cause the substitution order to be: “`ee`” before “`e`” before “`eee`”, and as the reader will realise, the rule for “`eee`” will never be applied.

This mechanism should be used with great care, as it is very easy to create irreversible substitutions using it.

3.9 Detection of incomplete substitutions

In some cases, the purpose of a bitrans substitution is to replace only part of the input file. An example of this was used in Section 3.3, where only some characters (here: German *Umlauts*) were intended to be replaced.

In many other cases, the purpose is to replace all text, except for word separator symbols as listed in Section 3.4, or comments as intended in Section 3.2. In such cases, it may be useful to know if the complete susbtitution was successful or not.

This is achieved by adding an option to the command line, as follows:

Option `-w1` will add a single line to standard error output stating how many lines of the intput file led to incomplete substitutions.

Option `-w2` will additionally create a listing of all such lines to a file named `bitrans.log`.

This log file will have three entries for each line with incomplete substitutions:

- The original input line preceded by ‘<’
- The output line precided by ‘>’
- A line placing indicators under each non-substituted character in the output line.

Note that at the start of every execution of bitrans, the file `bitrans.log` is deleted from the current directory, regardless of any `-w` option. This is to avoid misleading information coming from earlier executions of bitrans. The user should rename this file in case it should be preserved for later use.

3.10 Debugging output / verbose option

Most users should never be in a situation where he would want to generate debug output, but this capability has been used extensively during the development and testing of bitrans. Any such debugging output will be written to the named file:

```
bit_debug.txt
```

It is invoked with the `-v` option.

This file is only overwritten when debugging is enabled, and it is not modified when debugging is not enabled.

3.11 Diagnostics

The translation/substitution of the input file will be written to standard output (user console) unless an output file is specified on the command line. The standard output can also be re-directed to a file using the standard Unix/Linux redirection symbol: `>` .

Diagnostics output (with two exceptions, see further below) will be written to standard error output, which will appear on the user console (unless re-directed). In addition, it can be controlled (muted) by command line options of bitrans.

The option `–m1` mutes all informational messages, but does not affect warnings or error messages. Effectively, the entire user console output of the first example in Section 2.2 would be suppressed.

The option `–m2` mutes all standard error output. This may be useful when bitrans is used in pipes, and mixed standard error outputs would be confusing. In such a situation, the user will have no indication whether the execution was successful or not.

4 Options

bitrans allows a few user options, that must be specified as arguments on the command line.

All options must start with – (minus). The possible options and their effects are listed in Table 1 below.

Table 1: bitrans options

Option	Meaning	Comment
`-1`	Translation direction 1: left to right	This is the default direction, so this option is in fact superfluous
`-2`	Translation direction 2: right to left	
`-f`	Re-define the name of the rules file	The rules file name must immediately follow the `-f` , separated by at least one space
`-m0`	Normal output to `<stderr>`.	Default, so superfluous
`-m1`	Suppress output to `<stderr>`, except for warnings and error messages.	
`-m2`	Suppress all output to `<stderr>`.	
`-s`	Strict enforcement of transliteration alphabets	This option only affects the processing of files in IVTFF format. See Section 3.6 for the detailed rules.
`-v0`	No additional debugging output	Default, so superfluous
`-v1`	Debugging output is generated in relation with parsing the rules file	This output is written to the named file `bit_debug.txt`
`-v2`	Debugging output is generated in relation with the substitutions of the first 10 lines of the input file	(same as above)
`-v3`	The combination of options `-v1` and `-v2`	(same as above)
`-w0`	No warnings for incomplete substitutions	Default, so superfluous. See Section 3.9.
`-w1`	Write the number of lines, for which substitution was not complete, to standard error output	See Section 3.9.
`-w2`	Write the number of lines, for which substitution was not complete, to standard error output. In addition, write a detailed summary of these lines to named file `bitrans.log`	See Section 3.9.

bitrans arguments that start with a character other than – (minus) are taken as file names. If any such arguments are given, the first will be used as the name of the input file. If this does not appear, then standard input (`<stdin>`) will be used for the input text. A second such argument will be used as the name of the output file. If this does not appear, then standard output (`<stdout>`) will be used for the output text.

Example:

```
   bitrans file1.txt –f file2.txt file3.txt
```

In this case, the input file will be “`file1.txt`”, the output file will be “`file3.txt`” and the rules file will be “`file2.txt`”.

5 Rules file format

The rules file is a normal Ascii file, of which all lines should not be more than 64 characters long.

It is required to have one header record, which has an obligatory part and a few optional fields.

The rules file can have an optional record to redefine the separator character (see Section 3.3). If this appears, it must be the second record in the file.

It will then have any combination of:

- Comment definition records (see Section 3.2)
- Substitution rule records

While these may be freely mixed, it is recommended to put all comment definition records before the substitution rule records.

There can be at most 512 substitution rules in any rules file (this may be increased in a future version).

The header record can look like any of the following:

```
##BIT
##BIT1
##BIT  Curr Eval
```

The first five characters must say: "`##BIT`".

Position six should be blank, or have the character "`1`" or "`2`".

If there is any other character, it is the same as if there was a blank.

If it has a "1", it means that the file is uni-directional, and can only be used left-to-right. (This was the case in the example in Section 2.2).

If it has a "2", it means that the file is uni-directional, and can only be used right-to-left.

Positions 7 to 16 may or may not exist. If they exist, then:

- Positions 7 and 12 should be blank
- Positions 8-11 and 13-16 should each have a four-character code

These four-character codes are only needed in case the rules file is used to convert a Voynich MS transliteration file in IVTFF or VDBTF format. In that case, the four-character codes should correspond with the transliteration alphabets representing the two columns in the file. Following are some of the codes that are pre-defined, but the user is free to define his own codes for this own alphabets.

(This table may be compared with Table 4 in Ref R-1).

Table 2: Some pre-defined codes for Voynich MS transliteration alphabets

Code	Meaning
`Curr`	The alphabet used by Prescott Currier
`Eva-`	Eva (either basic or extended Eva)
`FSG-`	The alphabet agreed by Friedman and his team
`v101`	The voynich-101 alphabet by Glen Claston
`STA1`	The Super-Transliteration Alphabet

If these codes do not appear in the file, bitrans will use four spaces for each of them.

The separator re-definition record, if it exists, shall be the second record in the rules file, and have exactly three characters.

The first two shall say: "`#=`".

The third character shall define the character used to indicate the separator in any context-dependent substitution rules in this file (see Section 3.3). The character "`&`" is not allowed as a separator character.

Example:

```
##BIT
#=%
 %9    %con
9%  us%
```

Comment definition records must have:

- A single character in position 1
- The string "`(comment)`" in positions 2-10
- A single character in position 11

If the character in position 11 is a space character, the record defines a full-line comment. This space should exist in the file. If it does not exist, i.e. the record ends after the closing "`)`", a warning will be issued, and a space will be assumed.

Otherwise, it defines a limited-extent comment.

For the meaning of these comments, see Section 3.2.

Examples:

```
{(comment)}
*(comment)
```

Substitution rule records form the core of the rules file. They are free-formatted records that must have at least 2, and not more than 8 non-blank fields.

There may be any number (including zero) of leading blanks, any number of blanks between the various fields, and any number (including zero) of blanks after the last field.

The record may not be more than 64 characters wide (including all spaces).

The maximum nr. of such records is presently set at 512.

Examples:

```
a 1
  bla   blabla
 77    x
zero  0  null
```

There is a second limitation, namely that the total combined length of all items in each column (with an additional terminator character per item) shall not exceed 4096. Am error message will be generated in case this occurs

6 Diagnostics messages

6.1 Information messages

All diagnostics messages are written to `<stderr>`. Information messages may be suppressed using the `-m1` option on the command line.

6.2 Warning messages

Warning messages may appear in case something irregular occurs, which allows the tool to continue. They can be suppressed using the `-m2` option.

All warnings start with the character sequence: “`W: `“

6.3 Errors

Errors cause the tool to stop. Error messages are usually self-explanatory.

Error messages may be suppressed using the `-m2` option. In such a case, the user must find out using other means whether the tool has completed correctly.

This option may be useful in routine processes, where bitrans is part of a pipe, and the `<stderr>` output could be confusing in combination with output from other processes in the pipe.

All errors start with the character sequence: “`E: `“

Following are the error messages that can be generated by bitrans, with their meaning.

```
E: illegal unicode
```

In case the four-character string between `&` and `;` has invalid hex characters (see Section 3.3).

```
E: too many rules
```

The rules file includes more than 512 rules.

```
E: cannot process rules input token
```
or
```
E: cannot process rules output token
```

Follows the “illegal Unicode” message.

```
E: combined input tokens too long
```
or
```
E: combined output tokens too long
```

This is an error in the rules file. It occurs when the total length of all concatenated strings in either column is greater than 4096. Note that the stored length of each of these strings is one more than the visible characters. If a rules file re-defines strings that are all eight characters long, then bitrans stores 9 characters for each of them, and this error will be generated if there are 456 (or more) such strings (456 * 9 = 4104 > 4096).

```
E: input file does not exist
```

An input file named via a command line option must exist. If it is re-directed from `<stdin>` then a non-existent file is not an error, but is treated like an empty file.

```
E: cannot open output file
```

An output file named via a command line option is write-protected, or in an area where the user has no write permission. If this is the case for a file re-directed via `<stdout>`, a system error message will appear instead, and bitrans will not even start.

```
E: rules file does not exist
```

Either the default rules file, or the user-named rules file, does not exist.

```
E: Record too long.
Line read so far: <input line>
```

When processing the rules file: it has a record of more than 64 characters. When processing the input file: it has a record of more than 2048 characters.

```
E: rules file has no valid header
```

The rules file does not have `##BIT` in the first five characters of the first line. (It probably is some text file rather than a rules file).

```
E: user-requested direction N forbidden
```

Where N is 1 or 2. The rules file header enforces direction 1 or 2, while the user has requested the other direction via command line option (or default).

```
E: invalid line for #= record
```

The separator re-definition record (see Section 0) appears in an incorrect position in the rules file. It may only appear as the second record.

```
E: separator & not allowed
```

The separator character is not allowed to be an ampersand. This is new as from version 1.4.

```
E: too many comment records
```

There may be at most 6 comment records, and the rules file has more than 6.

```
E: single-word record not recognised
```

There is a single-word record in the rules file, which is not a separator re-definition record or a comment record.

```
E: cannot incorporate this rule
```

This follows an earlier message of the type “too many rules” or “combined input/output subs too long”.

```
E: multiple rules for <string>
```

This error means that the rules file has conflicting rules, i.e. the same string is requested to be replaced in several rules. This shows that the rules file is uni-directional, but this was not indicated in the header, and the user is trying to use it in the wrong direction. This error is generated for every conflicting rule in a rules file, before the program stops.

```
E: only two file names allowed
   error parsing command line
```

This error occurs if the command line has three (or more) arguments that do not start with a minus sign. In this count, the file name following the `-f` option is ignored.

```
  error opening file(s)
```

This message follows an earlier error message related to the input, output or rules file (which were listed above).

```
  error reading rules file
```

This message follows an earlier error message related to the rules file (which were listed above).

```
  error found in rules file
```

This message follows earlier error message when analysing the rules file (“`multiple rules for …`”).

```
E: incomplete record before EOF
```

The input file is missing a <newline> at the end of the last record.

```
  error reading line from input
```

This message follows an earlier error message when reading a record from the input file.

```
W: alphabet in IVTFF file: <name>
   does not match rules file: <name>
E: this is an error
```

This means that the user has specified the “strict” option (`-s`), but the transliteration alphabet names do not match. In case the third line is missing, it is only a warning and the program will continue. This happens if the user did not specify the “strict” option.

```
E: error pre-processing line
```

This error should not occur. Contact me if it happens.

```
E: error performing substitution
```

This error should not occur. Contact me if it happens.

7 Operating system

This tool is based on C code written and compiled under Ubuntu Linux. In principle, it can be compiled in other Linux distributions, under Unix and other operating systems as well.

In Linux (and Unix) the standard input and output may be redirected, and bitrans may very effectively be used in pipes.

When used in Linux/Unix, it can process files that have Windows/DOS formatting conventions (CR-LF). It has never been tested in a Windows environment.

If an input file is lacking the last newline at the end of the file, this will be understood for the rules file, but not for the input text file. In that case, an error will be generated.