Title

stata.com

unicode translate — Translate files to Unicode

Description Syntax Options Remarks and examples Also see

Description

`unicode translate` translates files containing extended ASCII to Unicode (UTF-8).

Extended ASCII is how people got accented Latin characters such as "á" and "à" and got characters from other languages such as "Я", "Θ", and "わたし" before the advent of Unicode or, in this context, before Stata became Unicode aware.

> If you have do-files, ado-files, `.dta` files, etc., from Stata 13 or earlier—and those files contain extended ASCII—you need to use the `unicode translate` command to translate the files from extended ASCII to Unicode.

The `unicode translate` command is also useful if you have text files containing extended ASCII that you wish to read into Stata.

Syntax

Analyze files to be translated

```
unicode analyze filespec [, redo nodata]
```

Set encoding to be used during translation

```
unicode encoding set ["]encoding["]
```

Translate or retranslate files

```
unicode translate filespec [, invalid[(escape|mark|ignore)]
    transutf8 nodata]

unicode retranslate filespec [, invalid[(escape|mark|ignore)]
    transutf8 replace nodata]
```

Restore backups of translated files

```
unicode restore filespec [, replace]
```

Delete backups of translated files

```
unicode erasebackups, badidea
```

filespec is a single filename or a file specification containing * and ? specifying one or more files, such as

```
*.dta
*.do
*.*
*
myfile.*
year??data.dta
```

unicode analyzes and translates .dta files and text files. It assumes that filenames with suffix .dta contain Stata datasets and that all other suffixes contain text. Those other suffixes are .ado, .do, .mata, .txt, .csv, .sthlp, .class, .dlg, .idlg, .ihlp, .smcl, and .stbcal.

Files with suffixes other than those listed are ignored. Thus "*.*" would ignore any .docx files or files with other suffixes. If such files contain text, they can be analyzed and translated by specifying the suffix explicitly, such as info.README and *.README.

Options

redo is allowed with unicode analyze. unicode analyze remembers results from one run to the next so that it does not repeat results for files that have been previously analyzed and determined not to need translation. Thus unicode analyze's output focuses on the files that remain to be translated. redo specifies that unicode analyze show the analysis for all files specified.

nodata is used with unicode analyze, translate, and retranslate. It specifies that the contents of the str# and strL variables in .dta files are not to be translated. The contents of the variables are to be left as is. The default behavior is to translate if necessary.

If option nodata is specified, only the meta data—variable names, dataset label, variable labels, value labels, and characteristics—are analyzed and perhaps translated.

This option is provided for two reasons.

nodata is included for those who do not trust automated software to modify the most vital part of their datasets, the data themselves. We emphasize to those users that unicode backs up files, and so translated files are easily restored to their original status.

The other reason nodata is included is for those datasets that include string variables in which some variables (observations) use one encoding and other variables (observations) use another. Such datasets are rare and called mixed-encoding datasets. One could arise if dataset result.dta was the result of merging input1.dta and input2.dta, and input1.dta encoded its string variables using ISO-8859-1, whereas input2.dta used JIS-X-0208. Such datasets are rare because if this had occurred, you would have noticed when you produced result.dta. The two extended ASCII encodings are simply not compatible, and one group or another of characters would have displayed incorrectly.

invalid and invalid() are allowed with unicode translate and retranslate. They specify how invalid characters are to be handled. Invalid characters are not supposed to arise, and when they do, it is a sign that you have set the wrong extended ASCII encoding. So let's assume that you have indeed set the right encoding and that still one or a few invalid characters do arise. The stories on how this might happen are long and technical, and all of them involve you playing sophisticated font games, or they involve you using a proprietary extended ASCII encoding that is no longer available, and so you are using an encoding that is close to the actual encoding used.

By default, `unicode` will not translate files containing invalid characters. `unicode` instead warns you so that you can specify the correct extended ASCII encoding.

`invalid` specifies the invalid characters are to be shown with an escape sequence. If a string contained "A@B", where @ indicates an invalid character, after translation, the string might contain "A%XCDB", which is say, %XCD was substituted for @. In general, invalid characters are replaced with `%X##`, where ## is the invalid character's hex value. The substitution is admittedly ugly, but it ensures that distinct strings remain distinct, which is important if the string is used as an identifier when you use the data.

`invalid(escape)` is a synonym for `invalid`.

`invalid(mark)` specifies that the official Unicode replacement character be substituted for invalid characters. That official character is `\ufffd` in Unicode speak and how it looks varies across operating systems. On Windows, the Unicode replacement character looks like a square; on Mac and Unix, it looks like a question mark in a hexagon.

`invalid(ignore)` indicates that the invalid character simply be removed. "A@B" becomes "AB".

`transutf8` is allowed with `unicode translate` and `retranslate`. `transutf8` specifies that characters that look as if they are UTF-8 already should nonetheless be translated according to the extended ASCII encoding. Do not specify this option unless `unicode` suggests it when you translate the file without the option, and even then, specify the option only after you have examined the translated file and determined that you agree.

For most of us, this issue arises when two extended ASCII characters appear next to each other, such as a German word containing "üß", or a French word containing "àö". Even when extended ASCII characters are adjacent, that is not necessarily sufficient to mimic valid UTF-8 characters, but some combinations do mimic UTF-8.

Adjacent UTF-8 characters that mimic UTF-8 characters are actually likely when you are using a CJK extended ASCII encoding. CJK stands for Chinese, Japanese, and Korean.

In any case, if `unicode analyze` reports when valid UTF-8 strings appear and if the file needs translating because it is not all ASCII plus UTF-8, you may need to specify `transutf8` when you translate the file. If you are unsure, proceed by translating the file without specifying `transutf8`, inspect the result, and retranslate if necessary.

`replace` has nothing to do with translation and is allowed with `unicode retranslate` and `restore`. It has to do with the restoration of original, untranslated files from the backups that `unicode translate` and `retranslate` make. Option `replace` should not be specified unless `unicode` suggests it.

`unicode` keeps backups of your originals. When you restore the originals or retranslate files (which involves restoring the originals), `unicode` checks that the previously translated file is unchanged from when `unicode` last translated it. It does this because if you modified the translated file since translation, those changes might be important to you and because if `unicode` restored the original from the backup, you would lose those changes. `replace` specifies that it is okay to change the previously translated file even though it has changed.

`badidea` is used with `unicode erasebackups` and is not optional. Erasing the backups of original files is usually a bad idea. We recommend you keep them for six months or so. Eventually, however, you will want to delete the backups. You are required to specify option `badidea` to show that you realize that erasing the backups is a bad idea if done too soon.

Remarks and examples

stata.com

Remarks are presented under the following headings:

What is this about?

Stata 14 and later use UTF-8, a form of Unicode, to encode strings. Stata 13 and earlier used ASCII. Datasets, do-files, ado-files, help files, and the like may need translation to display properly in Stata 15.

Files containing strings using only plain ASCII do not need translation. Plain ASCII provides the following characters:

Latin letters:	A–Z, a–z
Digits:	0–9
Symbols:	! " # $ % & ' ( ) * + , – . /
	: ; < = > ? @ [\ ] ^ _ ‘
	{ | } ~

If the variable names, variable labels, value labels, and string variables in your `.dta` files and the lines in your do-files, ado-files, and other Stata text files contain only the characters above, there is nothing you need to do.

On the other hand, if your `.dta` files, do-files, ado-files, etc., contain accented characters such as

á è ô ü ý ...

or symbols such as

£ ¥ ...

or characters from other alphabets,

знать

こんにちは

then the files do need translating so that the characters display correctly.

`unicode analyze` will tell you whether you have such files, and `unicode translate` will translate them.

You first use `unicode analyze`. It may turn out that no files need translating, and in that case, you are done.

If you do have files that need translating, you will use `unicode translate`. `unicode translate` makes a backup of your file before translating it.

If you do have files that need translating, `unicode translate` will translate them. Before you can use `unicode translate`, you must set the extended ASCII encoding that your files used. You do this with `unicode encoding set`. Encodings go by names such as ISO-8859-1, Windows-1252,

Big5, ISO-2022-KR, and about a thousand other names. However, there are only 231 encodings. Most of the names are aliases (synonyms). ISO-8859-1, for instance, is also known as ISO-Latin1, Latin1, and other names.

See `help encodings` for more information on encodings. Some of you will find the appropriate encoding name immediately. Others will be able only to narrow down the alternatives. Even so, all is not lost. `unicode translate` makes it easy to translate and retranslate a file over and over again until you find the encoding that works best. Once you find that encoding, it is likely that all of your files are using the same encoding.

Do I need to translate my files?

Can I ignore the issue?

If you are asking whether you can close your eyes and ignore the issue, the answer is maybe and maybe not.

If you have files using extended ASCII, they will not display correctly in Stata 15. We view that as a significant problem, but let's assume that does not concern you. If you used extended ASCII for variable names, you may find it difficult or impossible to type the untranslated name. That would be a problem. Other than that, you are probably okay, or more accurately, we cannot think of a problem even though we have tried. We have tried because if we could think of a problem, we would have fixed it. Stata's data management routines have been modified and certified to work with UTF-8. If they receive extended ASCII, they can mightily mess up what is displayed, but beyond that, they should produce results equivalent to what previous Statas produced.

Our advice is, for safety's sake, do not ignore the problem.

However, you do not need to analyze and translate all of your files today. One day, you will `use` a dataset and results will look odd when you `describe` or `list` the data. You will see unprintable characters and probably mutter a few unprintable words yourself, but having discovered the problem, you can then turn to solving it using `unicode analyze` and `unicode translate`.

However, we recommend that you learn to use `unicode translate` today. Take some files you are working with, determine whether you have a problem, and fix them if you do.

Do my files need translation?

If you are asking whether you have files that contain extended ASCII in hopes that you do not, here is our answer:

If you live and work in an English-speaking country, you probably do not have files containing extended ASCII.

If you live and work outside an English-speaking country but you have limited yourself to the unadorned Latin alphabet, you probably do not have files containing extended ASCII.

Otherwise, you probably do have files containing extended ASCII.

How will I know what to do?

`unicode analyze` will tell you whether you have files containing extended ASCII. `unicode analyze` can look at single files, or it can look at all the files in a directory. And if you do have files containing extended ASCII, `unicode translate` will fix the files.

Overview of the process

You will analyze your files and, if necessary, translate them. You can do this one file at a time by typing

```
. unicode analyze myfile.dta
. unicode encoding set encoding
. unicode translate myfile.dta
```

or you can do this with all of your files at once by typing

```
. unicode analyze *
. unicode encoding set encoding
. unicode translate *
```

Shockingly, we are going to advise you that analyzing and even translating all of your files at once is perfectly safe! That is because

1. `unicode analyze` by default ignores files that are not Stata related.
2. `unicode analyze` reads your files and reports on them; it does not change them.
3. `unicode analyze` might report that no files need translating. In that case, you are done.
4. if you do have files that need translating, before you can use `unicode translate`, you must set the extended ASCII encoding. How you determine the encoding is the topic of the next section.
5. `unicode translate`, just like `unicode analyze`, ignores by default files that are not Stata related. Typing `unicode translate *` is safe.
6. `unicode translate` does not modify files that do not need translation. This does not hinge on your having run `unicode analyze`. Typing `unicode translate *` is safe.
7. `unicode translate` does not modify files in which the translation goes poorly; it discards the translation. Typing `unicode translate *` is safe.
8. `unicode translate` makes backups of the original of any file it does translate successfully. At any time, you can type

   ```
   . unicode restore *
   ```

   and the files in your directory are back to being just as they were when you started. Typing `unicode translate *` is safe.

In the rest of this manual entry, we could discuss what might happen when you run `unicode analyze` and `unicode translate` and offer advice on what you might do about it.

`unicode analyze` and `unicode translate`, however, produce a ream of output, especially if you run them on a group of files. That output is tailored to your files and your situation. That output states what did happen and offers advice. Read it.

How to determine the extended ASCII encoding

We are getting ahead of ourselves because we have not yet determined that any of your files do need translating. Whether translation is necessary can be determined without knowing the extended ASCII encoding.

Determining the encoding can be more difficult than you would wish. Back in the day when the experts were still trying to make the extended ASCII solution work, the cleverest among them went to a lot of effort to hide the encoding from you, and they did a good job.

When the time comes to type

```
. unicode encoding set encoding
```

see `help encodings`. We have advice. In the meantime, allow us to predict how this process will transpire:

Some of you will not be able to determine the encoding your files are using, but you will be able to make guesses and narrow the choices down to a few of them. Then you will experiment to see which works best. We say "see" because that is literally how you are going to do it. You will guess, you will translate, and you will look at the result. And then you will repeat the process with a different encoding. The `unicode` command will make the translation and retranslation part easy.

Many of you will discover the single encoding that works for all of your files. Some of you will discover that one encoding works for most of your files but that there are one or two other encodings that you have to use with other files.

And then there is the issue of mixed UTF-8 and extended ASCII. This will affect only a few of you.

1. `unicode translate` will warn you when a file is a mix of UTF-8 and extended ASCII. It warns you because 1) the file could be exactly what it appears to be, a mix of encodings, or 2) the file is all extended ASCII and a few extended ASCII strings are merely masquerading as UTF-8.

2. By default, `unicode translate` assumes that the file really is a mix. It does not translate the UTF-8 strings; it translates just the strings that are extended ASCII.

   Technical note: Here is how this works. A variable label appearing to be UTF-8 already is not translated, whereas another variable label containing extended ASCII is translated even if a part of it appears to be UTF-8. `unicode translate` assumes that each variable label follows a single encoding. This same logic applies to `str#` and `strL` variables in the data. The variable is assumed to use the same encoding in all observations.

3. The default assumption may be incorrect; the file could be entirely extended ASCII. The default assumption is more likely to be incorrect in the CJK case. You can determine whether the default assumption is correct by looking at the file after translation. If some parts of it look like memory junk, then use `unicode retranslate, transutf8` to retranslate the file, and if you do not like that result, use `unicode retranslate` without `transutf8` to return to the previous result. Or you could use `unicode restore` to return to the original file and start all over again, perhaps with a different encoding.

   Technical note: There is no difference between using `unicode restore` followed by `unicode translate` and using `unicode retranslate`. So if you want to try a different encoding, you can restore, set the new encoding, and translate, or you can set the new encoding and retranslate.

Use of unicode analyze

If the files you want to examine are not in the current directory, change to the appropriate directory:

```
. cd wherever
```

`unicode analyze` and all the rest of the `unicode` commands described in this entry look at files in the current directory and only files in the current directory. `unicode` does not even look in subdirectories of the current directory.

Analyze the file.

```
. unicode analyze myfile.dta
```

`unicode analyze` will report whether the file needs translation and provide other information, too. The output looks something like this:

```
. unicode analyze myfile.dta
File summary (before starting):
      1   file(s) specified
      1   file(s) to be examined ...
File myfile.dta (Stata dataset)
       ----------------------------------------
       File does not need translation
File summary:
    all files okay
```

Or it might look like this:

```
. unicode analyze myfile.dta
File summary (before starting):
      1   file(s) specified
      1   file(s) to be examined ...
File myfile.dta (Stata dataset)
      3 variable names need translation
      2 variable labels need translation
      1 str# variable needs translation
       ----------------------------------------
       File needs translation.
       Use unicode translate on this file
File summary:
    1 file needs translation
```

If you were to now rerun the analysis in the case where the file does not need translation, you would see something like this:

```
. unicode analyze myfile.dta
File summary (before starting):
      1   file(s) specified
      1   file(s) already known to be ASCII in previous runs
      0   file(s) to be examined ...
(nothing to do)
```

If you want to see the detailed output, type `unicode analyze myfile.dta, redo`.

The primary purpose of `unicode analyze` is to get the files that do not need translating out of the way. `unicode analyze` does not change your files; it just dismisses the ones that need no further work.

You can run `unicode analyze` on multiple files, and we recommend that you do that.

```
. unicode analyze *
     30   file(s) specified
      6   file(s) not Stata
      1   file(s) already known to be ASCII in previous runs
      1   file(s) already known to be UTF-8 in previous runs
     22   files(s) to be examined
```

There is more to the output, but before we look at that, note that `unicode analyze` reported that 6 files were not Stata. `unicode analyze` and `unicode translate` ignore non-Stata files unless you explicitly specify them, say, by typing `unicode analyze README` or `unicode analyze *.README`.

Let's now return to the remaining output from `unicode analyze *`:

```
File filename (filetype)
        notes about elements that need translating
        ----------------------------------------------------------
        recommendations
File filename (filetype)
        notes about elements that need translating
        ----------------------------------------------------------
        recommendations
.
.
File filename (filetype)
        notes about elements that need translating
        ----------------------------------------------------------
        recommendations
Files matching * that need translation:
        list of files
File summary:
      2   file(s) skipped (known okay from previous runs)
      8   file(s) need translation
```

`unicode analyze` produced a lot of output. If you are like us, you will want a log of the output and perhaps want to look at it in the Viewer. It is not too late, just remember to specify the `redo` option:

```
. log using output
. unicode analyze *, redo
  (output omitted)
. log close
. view output.smcl
```

If you are really like us, you will instead want a file you can edit in Stata's Do-file Editor:

```
. log using output.log
. unicode analyze *, redo
  (output omitted)
. log close
. doedit output.log
```

Now, you can edit the output to make a to-do list for yourself. We go through the output and delete the parts with which we agree, such as the following:

```
File myfile.do (text file)
        40 line(s) in file
  ------------------------------------------------------------
  File does not need translation.
```

Buried in the output, however, may be something like this:

```
File german.dta (Stata dataset)
-------------------------------------------------------------------------------
  File does not need translation, except ...
  The file appears to be UTF-8 already.  Sometimes, files that need
  translating can look like UTF-8.  Look at these examples:
      variable name "länge"
      variable label "Kofferraumvolumen (Kubikfuß)"
      value-label contents "Ausländisch"
      contents of str# variable marke
  Do they look okay to you?
  If not, the file needs translating or retranslating with the
  transutf8 option.  Type
      . unicode   translate "bill_utf8.dta", transutf8
      . unicode retranslate "bill_utf8.dta", transutf8
```

This file, too, is marked as not needing translation, and we agree based on the evidence presented, but we might not have agreed. Assume that the file was named `japan.dta` and that the examples did not look like Japanese but looked like memory junk. We would want to add this file to our list to translate and remind ourselves to specify option `transutf8` when translating.

It is unlikely that any file that `unicode analyze` reports as purely UTF-8 needs translating unless the file is short, and then you must look at it to determine whether the file really is UTF-8.

Here is a different example. The file, according to `unicode analyze`, needs translation, but it also includes UTF-8:

```
File filter.do (text file)
        40 line(s) in file
        33 line(s) ASCII
         1 line(s) UTF-8
         6 line(s) need translation
-------------------------------------------------------------------------------
  File needs translation.  Use unicode translate on this file.
    There are three possibilities.
    1) The file is exactly what it appears to be, a mix of extended
    ASCII and UTF-8.  Use unicode translate.
    2) The UTF-8 lines are extended ASCII masquerading as UTF-8.
    Use unicode translate, transutf8.
    3) The file is UTF-8 with some invalid characters.  Set the
    encoding to utf8 and then use unicode translate, invalid().
```

`unicode analyze` thinks this file needs translation and speculates about how it should be translated. Read the output. Possibility 3) did not even occur to us. Even so, and even without looking at the file, we would favor possibility 2) because there is only one UTF-8 line and there are 6 lines known to need translation.

You will learn that running `unicode analyze` is optional. The advantage of running `unicode analyze` is that it offers advice.

You can analyze files repeatedly. If you type `unicode analyze` without the `redo` option, the output reappears, but files are skipped that `unicode analyze` previously determined as not needing translation. Specify `redo` and you will see all the files.

`unicode analyze` remembers results from previous runs. Five years from now, `unicode analyze` will remember the files it has examined and determined do not need translation, and it will even know whether the file has changed in the intervening five years and so needs reexamination.

`unicode analyze` remembers from one run to the next by creating a directory named bak.stunicode, where it can put its notes. Ignore the directory and its subdirectories. When we tell you about `unicode translate`, you will learn that bak.stunicode is also where backups of unmodified original files

are stored. Now that you know that, you might be tempted to restore originals from the backups by copying the files. Do not do that because you will confuse `unicode`. Use `unicode restore` to restore originals. We will get to that.

The purpose of `unicode analyze` is to dismiss all the files that do not have problems so you can focus on those that do. When you later use `unicode translate`, it will also skip over files that do not need translating. Using `unicode analyze` is optional, and even if you do not use it, `unicode translate` will never translate a file that does not need it; `unicode translate` runs `unicode analyze` in secret if it needs to.

Use of unicode translate: Overview

Let's assume that we have used `unicode analyze` and learned that the following files need translating:

```
myfile.dta
anotherfile.do
```

Before we can translate the files, we must set the extended ASCII encoding. See `help encodings` when you are translating your files.

Let's just assume right now that we know the encoding for the files is ISO-8859-1, and then we will assume that we were wrong and show you how we get out of that situation.

Step 1. Inform `unicode` of the encoding by typing

```
. unicode encoding set ISO-8859-1
```

Step 2. Translate the files, one at a time by typing

```
. unicode translate myfile.dta
. unicode translate anotherfile.do
```

or both in one command by typing

```
. unicode translate *
```

Specifying `*` or `*.*` or `*.dta` or `m*.*` or any other file specification is perfectly safe. `unicode translate` ignores irrelevant files just as `unicode analyze` does. `unicode translate` also ignores files that do not need translating, and it ignores files that have already been translated. `unicode translate` does not depend on your having run `unicode analyze` previously.

`unicode translate` has another great feature: it makes backups of the files it modifies. If, after translation, you decide you do not like the translation, you can restore the original by typing

```
. unicode restore myfile.dta
```

You can even type

```
. unicode restore *
```

if you want all of your files restored.

You do not have to restore the original just to retranslate it. Use `unicode retranslate` instead:

```
. unicode retranslate myfile.dta
. unicode retranslate *
```

The only reason to run `unicode retranslate`, however, is if you want to specify different options or try a different encoding:

```
. unicode encoding set some_other_encoding
. unicode retranslate *
```

And if you do not like that result, you can still `unicode restore`.

Use of unicode translate: A word on backups

`unicode translate` and `retranslate` automatically make backups when they modify a file and a backup does not already exist. `unicode` calculates and keeps track of checksums calculated on the original and translated files, so it knows whether the files are subsequently changed. `unicode` is thoroughly tested. What could possibly go wrong?

If you are like us, you trust nobody with regard to your files. We do not even trust ourselves. Trust us on this. Make your own back up in whatever way you know before using `unicode translate`. Backup the entire directory. We would make a zip file of it, but if nothing else, just copy all the files to a new, out-of-the-way directory. We predict you will not need the copies, but one never knows for sure.

Even if `unicode` is perfect, the subsequent validity of the backups depends on the bak.stunicode subdirectory not being corrupted by another process or even by you. More than once, we have ourselves damaged files in haste.

After you have translated your files, keep the backups for a while. Eventually, however, there will come a day when the backups are no longer needed. The command to delete the backups of your originals is

```
. unicode erasebackups, badidea
```

You must specify option `badidea`. Think of `badidea` as an abbreviation for `badideaifdone-toosoon`: what you are doing in specifying the option is stating that it is not too soon.

Use of unicode translate: Output

`unicode translate`'s output looks just like `unicode analyze`'s output except that the content varies:

```
. unicode translate *
     30   file(s) specified
      6   file(s) not Stata
      6   file(s) already known to be ASCII in previous runs
      4   file(s) already known to be UTF-8 in previous runs
     14   files(s) to be examined
File filename (filetype)
      notes about the translation
      ------------------------------------------------
      result message
File badfile.ado (textfile)
      40 lines in file
      16 lines ASCII
       2 lines translated
      22 lines w/ invalid chars not translated
      ------------------------------------------------
      File not translated because it contains untranslatable
      characters;
             you need to specify a different encoding or, if you
             are sure that you have the correct encoding, use
             unicode translate with the invalid() option
.
.
```

```
File filename (filetype)
      notes about the translation
      notes about elements that need translating
      ------------------------------------------------
      result message
Files matching * that still need translation:
      badfile.ado
File summary:
     10   file(s) skipped (known okay from previous runs)
     13   file(s) successfully translated
      1   files(s) not translated because they contain
          untranslatable characters
              you need to specify a different encoding or, if you
              are sure that you have the correct encoding, use
              unicode translate with the invalid() option
```

One file still needs translation according to the output. How can files still need translation? The output explains. We had untranslatable characters. The output even says what to do about it. We should specify a different encoding—the fact that we had untranslatable characters is evidence that we are using the wrong encoding—or we should accept that there are invalid characters in our file and tell `unicode translate` how to handle them. It will help us make the decision if we scan up from the file-summary message to find the detailed output for badfile.ado:

```
File badfile.ado (textfile)
      40 lines in file
      16 lines ASCII
       2 lines translated
      22 lines w/ invalid chars not translated
      ------------------------------------------------
      File not translated because it contains untranslatable
      characters;
          you need to specify a different encoding or, if you
          are sure that you have the correct encoding, use
          unicode translate with the invalid() option
```

You can read about the `invalid()` option under *Options*, but this looks like a case where the file needs a different encoding; 2 lines translated with the current encoding, and 22 did not. If we had instead seen that 22 lines translated and that 2 lines had invalid characters, we would be less sure about needing a different encoding. Assume the output had been

```
File badfile.ado (textfile)
      40 lines in file
      38 lines ASCII
       2 lines w/ invalid chars not translated
      ------------------------------------------------
      File not translated because it contains untranslatable
      characters;
          you need to specify a different encoding or, if you
          are sure that you have the correct encoding, use
          unicode translate with the invalid() option
```

That an ado-file is mostly ASCII does not surprise us. The fact that no lines could be translated (given the encoding) speaks volumes. We need a different encoding.

Most of our files were translated. For successful translations, the detailed output for `.dta` files will be something like the following:

```
File trees.dta (Stata dataset)
        9 variable names okay, ASCII
        3 variable names translated
      all data labels okay, ASCII
        8 variable labels okay, ASCII
        4 variable labels translated
      all value-label names okay, ASCII
      all value-label contents translated
      all characteristic names okay, ASCII
      all characteristic contents okay, ASCII
      all str# variables okay, ASCII
      ------------------------------------------------------
      File successfully translated
```

The detailed output for text files might look like the following:

```
File runjob.do (textfile)
    120 lines in file
    101 lines ASCII
     19 lines translated
     ------------------------------------------------------
     File successfully translated
```

Here is an example of a file that translated successfully but produced a lot of output:

```
File northwest.dta (Stata dataset)
    all variable names okay, ASCII
    all data labels okay, ASCII
    all variable labels okay, ASCII
    all value-label names okay, ASCII
    all value-label contents okay, ASCII
    all characteristic names okay, ASCII
    all characteristic contents okay, ASCII
      1 strL variable okay, ASCII
      1 strL variable(s) have binary values
           This concerns strL variable diagnotes.
           StrL variables that contain binary values in even one
           observation are not translated by unicode.  Translating
           binary values is inappropriate.  Rarely, however,
           "binary" values are just text or the variable contains
           binary values in some observations and nonbinary values
           in others.  You translate such variables using generate
           or replace; see translating binary strLs.
      1 strL variable translated
      2 str# variables okay, ASCII
      1 str# variable translated
     ------------------------------------------------------
     File successfully translated
```

The extra output concerns a strL variable that was not translated. The output states that the variable is binary and that translating binary strLs is inappropriate, but maybe not. This is the topic of the next section.

Translating binary strLs

`unicode translate` does not translate binary strLs. That is probably the right decision. StrLs are sometimes used in Stata to record documents, images, and other binary files, and modifying binary files is never a good idea.

Stata marks strL variables as binary on an observation-by-observation basis. As far as `unicode translate` is concerned, however, if there is just one observation in which the strL is marked as binary, it treats all observations as binary and does not translate them. The thinking is that variables hold different realizations of the same underlying type of thing, and if the strL is binary in one observation, it is probably truly binary in all observations.

Perhaps you know differently in your specific application and wish to translate the variable's nonbinary observations or all of its observations. Here is how you do that.

You use string function `ustrfrom()` to obtain a translated string. Assuming the existing strL variable is named `myvar`, you type

```
. generate strL newvar = ustrfrom(myvar, "encoding", #)
```

Specify encoding just as you would with `unicode encoding set` *encoding*. *encoding* might be ISO-8859-1, Windows-1252, Big5, ISO-2022-KR, or any other extended ASCII encoding. Whatever string you specify for *encoding*, make sure it is valid and spelled correctly. Testing the string with `unicode encoding set` is one way to do that.

is specified as 1, 2, 3, or 4 and determines how invalid characters are to be handled. Three of the four values correspond to `unicode`'s `invalid()` option:

1	is equivalent to	`invalid(mark)`
2	is equivalent to	`invalid(ignore)`
4	is equivalent to	`invalid(escape)`

The remaining code, 3, specifies that the function return " " if invalid characters are encountered.

So one way of translating all the values of `myvar` would be

```
. generate strL try = ustrfrom(myvar, "ISO-8859-1", 1)
. browse newvar         // review result
. replace newvar = try
. drop try
```

If you want to translate only the nonbinary values of `myvar`, you could type

```
. gen strL try = ustrfrom(myvar, "ISO-8859-1", 1) if !_strisbinary(myvar)
. replace try = myvar if _strisbinary(myvar)
```

That would use Stata's definition of binary, which is difficult to explain. Another good definition of binary is that the string not contain binary 0:

```
. gen strL try = ustrfrom(myvar, "ISO-8859-1", 1) if !strpos(myvar, char(0))
. replace try = myvar if strpos(myvar, char(0))
```

Also see

[D] **unicode** — Unicode utilities

[U] **12.4.2 Handling Unicode strings**

[U] **12.4.2.6 Advice for users of Stata 13 and earlier**