infile (fixed format) — Import text data in fixed format with a dictionary

Description Quick start Menu Syntax
Options Remarks and examples Also see

Description

`infile using` reads a dataset that is stored in text form. `infile using` does this by first reading *dfilename*—a "dictionary" that describes the format of the data file—and then reads the file containing the data. The dictionary is a file you create with the Do-file Editor or an editor outside Stata.

Strings containing plain ASCII or UTF-8 are imported correctly. Strings containing extended ASCII will not be imported (that is, displayed) correctly; you can use Stata's `replace` command with the `ustrfrom()` function to convert extended ASCII to UTF-8. If `ebcdic` is specified, the data will be converted from EBCDIC to ASCII as they are imported. The dictionary in all cases must be ASCII.

If `using` *filename* is not specified, the data are assumed to begin on the line following the closing brace. If `using` *filename* is specified, the data are assumed to be located in *filename*.

The data may be in the same file as the dictionary or in another file. `infile` with a dictionary can import both numeric and string data. Individual strings may be up to 100,000 bytes long. Strings longer than 2,045 bytes are imported as `strL`s (see **[U] 12.4.8 strL**).

Another variation on `infile` omits the intermediate dictionary; see [D] **infile (free format)**. This variation is easier to use but will not read fixed-format files. On the other hand, although `infile` with a dictionary will read free-format files, `infile` without a dictionary is even better at it.

An alternative to `infile using` for reading fixed-format files is `infix`; see [D] **infix (fixed format)**. `infix` provides fewer features than `infile using` but is easier to use.

Stata has other commands for reading data. If you are not certain that `infile using` will do what you are looking for, see [D] **import** and **[U] 22 Entering and importing data**.

Quick start

For dictionary file `mydata.dct` that reads `int`-type `v1` and `str10`-type `v2`

```
dictionary {
  int   v1
  str10  v2
}
```

Import data from `mydata.raw` with instructions for reading the data contained in dictionary file `mydata.dct`

```
infile using mydata.dct, using(mydata.raw)
```

Same as above

```
infile using mydata, using(mydata)
```

Same as above, but import data from `mydata.txt`

```
infile using mydata, using(mydata.txt)
```

Same as above, but read only the first 10 observations

```
infile using mydata in 1/10, using(mydata.txt)
```

Read only observations where catvar is equal to 4 or 5

```
infile using mydata if catvar==4 | catvar==5, using(mydata.txt)
```

Menu

File > Import > Text data in fixed format with a dictionary

Syntax

infile using *dfilename* [*if*] [*in*] [, *options*]

If *dfilename* is specified without an extension, .dct is assumed. If *dfilename* contains embedded spaces, remember to enclose it in double quotes.

options	Description
Main	
using(*filename*)	text dataset filename
clear	replace data in memory
Options	
automatic	create value labels from nonnumeric data
ebcdic	treat text dataset as EBCDIC

A dictionary is a text file that is created with the Do-file Editor or an editor outside Stata. This file specifies how Stata should read fixed-format data from a text file. The syntax for a dictionary is

begin dictionary file

```
[infile] dictionary [using filename] {

        * comments may be included freely
        _lrecl(#)
        _firstlineoffile(#)
        _lines(#)
        _line(#)
        _newline[(#)]
        _column(#)

        _skip[(#)]
        [type] varname [:lblname] [%infmt] ["variable label"]
}
(your data might appear here)
```

end dictionary file

where %*infmt* is { %[#[.#]]{f|g|e} | %[#]s | %[#]S }

Options

Main

`using(`*filename*`)` specifies the name of a file containing the data. If `using()` is not specified, the data are assumed to follow the dictionary in *dfilename*, or if the dictionary specifies the name of some other file, that file is assumed to contain the data. If `using(`*filename*`)` is specified, *filename* is used to obtain the data, even if the dictionary says otherwise. If *filename* is specified without an extension, `.raw` is assumed.

If *filename* contains embedded spaces, remember to enclose it in double quotes.

`clear` specifies that it is okay for the new data to replace what is currently in memory. To ensure that you do not lose something important, `infile using` will refuse to read new data if other data are already in memory. `clear` allows `infile using` to replace the data in memory. You can also drop the data yourself by typing `drop _all` before reading new data.

Options

`automatic` causes Stata to create value labels from the nonnumeric data it reads. It also automatically widens the display format to fit the longest label.

`ebcdic` specifies that the data be stored using EBCDIC character encoding rather than the default ASCII encoding and that the data be converted from EBCDIC to ASCII as they are imported.

Dictionary directives

`*` marks comment lines. Wherever you wish to place a comment, begin the line with a `*`. Comments can appear many times in the same dictionary.

`_lrecl(#)` is used only for reading datasets that do not have end-of-line delimiters (carriage return, line feed, or some combination of these). Such files are often produced by mainframe computers and are either coded in EBCDIC or have been translated from EBCDIC into ASCII. `_lrecl()` specifies the logical record length. `_lrecl()` requests that `infile` act as if a line ends every # bytes.

`_lrecl()` appears only once, and typically not at all, in a dictionary.

`_firstlineoffile(#)` (abbreviation `_first()`) is also rarely specified. It states the line of the file where the data begin. You do not need to specify `_first()` when the data follow the dictionary; Stata can figure that out for itself. However, you might specify `_first()` when reading data from another file in which the first line does not contain data because of headers or other markers.

`_first()` appears only once, and typically not at all, in a dictionary.

`_lines(#)` states the number of lines per observation in the file. Simple datasets typically have `_lines(1)`. Large datasets often have many lines (sometimes called records) per observation. `_lines()` is optional, even when there is more than one line per observation because `infile` can sometimes figure it out for itself. Still, if `_lines(1)` is not right for your data, it is best to specify the correct number through `_lines(#)`.

`_lines()` appears only once in a dictionary.

`_line(#)` tells `infile` to jump to line # of the observation. `_line()` is not the same as `_lines()`. Consider a file with `_lines(4)`, meaning four lines per observation. `_line(2)` says to jump to the second line of the observation. `_line(4)` says to jump to the fourth line of the observation. You

may jump forward or backward. infile does not care, and there is no inefficiency in going forward to _line(3), reading a few variables, jumping back to _line(1), reading another variable, and jumping forward again to _line(3).

You need not ensure that, at the end of your dictionary, you are on the last line of the observation. infile knows how to get to the next observation because it knows where you are and it knows _lines(), the total number of lines per observation.

_line() may appear many times in a dictionary.

_newline[(#)] is an alternative to _line(). _newline(1), which may be abbreviated _newline, goes forward one line. _newline(2) goes forward two lines. We do not recommend using _newline() because _line() is better. If you are currently on line 2 of an observation and want to get to line 6, you could type _newline(4), but your meaning is clearer if you type _line(6).

_newline() may appear many times in a dictionary.

_column(#) jumps to column # (in bytes) of the current line. You may jump forward or backward within a line. _column() may appear many times in a dictionary.

_skip[(#)] jumps forward # columns on the current line. _skip() is just an alternative to _column(). _skip() may appear many times in a dictionary.

[*type*] *varname* [:*lblname*] [%*infmt*] ["*variable label*"] instructs infile to read a variable. The simplest form of this instruction is the variable name itself: *varname*.

At all times, infile is on some column of some line of an observation. infile starts on column 1 of line 1, so pretend that is where we are. Given the simplest directive, '*varname*', infile goes through the following logic:

If the current column is blank, it skips forward until there is a nonblank column (or until the end of the line). If it just skipped all the way to the end of the line, it stores a missing value in *varname*. If it skipped to a nonblank column, it begins collecting what is there until it comes to a blank column or the end of the line. These are the data for *varname*. Then it sets the current column to wherever it is.

The logic is a bit more complicated. For instance, when skipping forward to find the data, infile might encounter a quote. If so, it then collects the characters for the data by skipping forward until it finds the matching quote. If you specified a %*infmt*, then infile skips the skipping-forward step and simply collects the specified number of bytes. If you specified a %S *infmt*, then infile does not skip leading or trailing blanks. Nevertheless, the general logic is (optionally) skip, collect, and reset.

Remarks and examples

Remarks are presented under the following headings:

Introduction
Reading free-format files
Reading fixed-format files
Numeric formats
String formats
Specifying column and line numbers
Examples of reading fixed-format files
Reading fixed-block files
Reading EBCDIC files

Introduction

`infile using` follows a two-step process to read your data. You type something like `infile using descript`, and

1. `infile using` reads the file `descript.dct`, which tells `infile` about the format of the data; and
2. `infile using` then reads the data according to the instructions recorded in `descript.dct`.

`descript.dct` (the file could be named anything) is called a dictionary, and `descript.dct` is just a text file that you create with the Do-file Editor or an editor outside Stata.

As for the data, they can be in the same file as the dictionary or in a different file. It does not matter.

Reading free-format files

Another variation of `infile` for reading free-format files is described in [D] **infile (free format)**. We will refer to this variation as `infile` without a dictionary. The distinction between the two variations is in the treatment of line breaks. `infile` without a dictionary does not consider them significant. `infile` with a dictionary does.

A line, also known as a record, physical record, or physical line (as opposed to observations, logical records, or logical lines), is a string of characters followed by the line terminator. If you were to type the file, a line is what would appear on your screen if your screen were infinitely wide. Your screen would have to be infinitely wide so that there would be no possibility that one line could take more than one line of your screen, thus fooling you into thinking that there are multiple lines when there is only one.

A logical line, on the other hand, is a sequence of one or more physical lines that represent one observation of your data. `infile` with a dictionary does not spontaneously go to new physical lines; it goes to a new line only between observations and when you tell it to. `infile` without a dictionary, on the other hand, goes to a new line whenever it needs to, which can be right in the middle of an observation. Thus consider the following little bit of data, which is for three variables:

```
5 4
1 9 3
2
```

How do you interpret these data?

Here is one interpretation: There are 3 observations. The first is 5, 4, and missing. The second is 1, 9, and 3. The third is 2, missing, and missing. That is the interpretation that `infile` with a dictionary makes.

Here is another interpretation: There are 2 observations. The first is 5, 4, and 1. The second is 9, 3, and 2. That is the interpretation that `infile` without a dictionary makes.

Which is right? You would have to ask the person who entered these data. The question is, are the line breaks significant? Do they mean anything? If the line breaks are significant, you use `infile` with a dictionary. If the line breaks are not significant, you use `infile` without a dictionary.

The other distinction between the two `infile`s is that `infile` with a dictionary does not process comma-separated–value format. If your data are comma-separated, tab-separated, or otherwise delimited, see [D] **import delimited** or [D] **infile (free format)**.

Example 1: A simple dictionary with data

Outside Stata, we have typed into the file `highway.dct` information on the accident rate per million vehicle miles along a stretch of highway, the speed limit on that highway, and the number of access points (on-ramps and off-ramps) per mile. Our file contains

begin `highway.dct`, example 1

```
infile dictionary {
        acc_rate  spdlimit acc_pts
}
4.58 55 4.6
2.86  60 4.4
1.61 . 2.2
3.02 60 4.7
```

end `highway.dct`, example 1

This file can be read by typing the commands below. Stata displays the dictionary and reads the data:

```
. infile using highway
infile dictionary {
        acc_rate  spdlimit acc_pts
}
(4 observations read)
. list
```

	acc_rate	spdlimit	acc_pts
1.	4.58	55	4.6
2.	2.86	60	4.4
3.	1.61	.	2.2
4.	3.02	60	4.7

Example 2: Specifying variable labels

We can include variable labels in a dictionary so that after we `infile` the data, the data will be fully labeled. We could change `highway.dct` to read

begin `highway.dct`, example 2

```
infile dictionary {
* This is a comment and will be ignored by Stata
* You might type the source of the data here.
        acc_rate  "Acc. Rate/Million Miles"
        spdlimit  "Speed Limit (mph)"
        acc_pts   "Access Pts/Mile"
}
4.58 55 4.6
2.86  60 4.4
1.61 . 2.2
3.02 60 4.7
```

end `highway.dct`, example 2

Now when we type `infile using highway`, Stata not only reads the data but also labels the variables.

Example 3: Specifying variable storage types

We can indicate the variable types in the dictionary. For instance, if we wanted to store `acc_rate` as a `double` and `spdlimit` as a `byte`, we could change `highway.dct` to read

begin highway.dct, example 3

```
infile dictionary {
* This is a comment and will be ignored by Stata
* You might type the source of the data here.
 double acc_rate  "Acc. Rate/Million Miles"
 byte   spdlimit  "Speed Limit (mph)"
        acc_pts   "Access Pts/Mile"
}
4.58 55 4.6
2.86  60 4.4
1.61 . 2.2
3.02 60 4.7
```

end highway.dct, example 3

Because we do not indicate the variable type for `acc_pts`, it is given the default variable type `float` (or the type specified by the `set type` command).

◁

▷ Example 4: Reading string variables

By specifying the types, we can read string variables as well as numeric variables. For instance,

begin emp.dct

```
infile dictionary {
* data on employees
  str20 name        "Name"
        age         "Age"
    int sex         "Sex coded 0 male 1 female"
}
"Lisa Gilmore" 25 1
Branton 32 1
'Bill Ross' 27 0
```

end emp.dct

The strings can be delimited by single or double quotes, and quotes may be omitted altogether if the string contains no blanks or other special characters.

◁

▷ Example 5: Specifying value labels

You may attach value labels to variables in the dictionary by using the colon notation:

begin emp2.dct

```
infile dictionary {
* data on name, sex, and age
  str16 name        "Name"
        sex:sexlbl  "Sex"
    int age         "Age"
}
"Arthur Doyle" Male 22
"Mary Hope" Female 37
"Guy Fawkes" Male 48
"Karen Cain" Female 25
```

end emp2.dct

If you want the value labels to be created automatically, you must specify the `automatic` option on the `infile` command. These data could be read by typing `infile using emp2, automatic`, assuming the dictionary and data are stored in the file `emp2.dct`.

◁

Example 6: Separate the dictionary and data files

The data need not be in the same file as the dictionary. We might leave the highway data in `highway.raw` and write a dictionary called `highway.dct` describing the data:

begin `highway.dct`, example 4

```
infile dictionary using highway {
* This dictionary reads the file highway.raw.  If the
* file were called highway.txt, the first line would
* read "dictionary using highway.txt"
        acc_rate  "Acc. Rate/Million Miles"
        spdlimit  "Speed Limit (mph)"
        acc_pts   "Access Pts/Mile"
}
```

end `highway.dct`, example 4

Example 7: Ignoring the top of a file

The `firstlineoffile()` directive allows us to ignore lines at the top of the file. Consider the following raw dataset:

begin `mydata.raw`

```
The following data were entered by Marsha Martinez.  It was checked by
Helen Troy.
id income educ sex age
1024 25000 HS Male 28
1025 27000 C Female 24
```

end `mydata.raw`

Our dictionary might read

begin `mydata.dct`

```
infile dictionary using mydata {
        _first(4)
        int id "Identification Number"
        income "Annual income"
        str2 educ "Highest educ level"
        str6 sex
        byte age
}
```

end `mydata.dct`

Example 8: Data spread across multiple lines

The `_line()` and `_lines()` directives tell Stata how to read our data when there are multiple records per observation. We have the following in `mydata2.raw`:

begin mydata2.raw

```
id income educ sex age
1024 25000 HS
Male
28
1025 27000 C
Female
24
1035 26000 HS
Male
32
1036 25000 C
Female
25
```

end mydata2.raw

We can read this with a dictionary `mydata2.dct`, which we will just let Stata list as it simultaneously reads the data:

```
. infile using mydata2, clear
infile dictionary using mydata2 {
    _first(2)                       * Begin reading on line 2
    _lines(3)                       * Each observation takes 3 lines.
    int id "Identification Number"  * Since _line is not specified, Stata
    income "Annual income"          * assumes that it is 1.
    str2 educ "Highest educ level"
    _line(2)                        * Go to line 2 of the observation.
    str6 sex                        * (values for sex are located on line 2)
    _line(3)                        * Go to line 3 of the observation.
    int age                         * (values for age are located on line 3)
}
(4 observations read)
. list
```

	id	income	educ	sex	age
1.	1024	25000	HS	Male	28
2.	1025	27000	C	Female	24
3.	1035	26000	HS	Male	32
4.	1036	25000	C	Female	25

Here is the really good part: we read these variables in order, but that was not necessary. We could just as well have used the dictionary:

begin mydata2p.dct

```
infile dictionary using mydata2 {
        _first(2)
        _lines(3)
        _line(1)   int   id     "Identification number"
                         income "Annual income"
                   str2  educ   "Highest educ level"
        _line(3)   int   age
        _line(2)   str6  sex
}
```

end mydata2p.dct

We would have obtained the same results just as quickly, the only difference being that our variables in the final dataset would be in the order specified: `id`, `income`, `educ`, `age`, and `sex`.

◁

❑ Technical note

You can use _newline to specify where breaks occur, if you prefer:

begin highway.dct, example 5

```
infile dictionary {
         acc_rate  "Acc. Rate/Million Miles"
         spdlimit  "Speed Limit (mph)"
_newline acc_pts   "Access Pts/Mile"
}
4.58 55
4.6
2.86  60
 4.4
1.61 .
2.2
3.02 60
 4.7
```

end highway.dct, example 5

The line reading '1.61 .' could have been read 1.61 (without the period), and the results would have been unchanged. Because dictionaries do not go to new lines automatically, a missing value is assumed for all values not found in the record.

❑

Reading fixed-format files

Values in formatted data are sometimes packed one against the other with no intervening blanks. For instance, the highway data might appear as

begin highway.raw, example 6

```
4.58554.6
2.86604.4
1.61  2.2
3.02604.7
```

end highway.raw, example 6

The first four columns of each record represent the accident rate; the next two columns, the speed limit; and the last three columns, the number of access points per mile.

To read these data, you must specify the %*infmt* in the dictionary. Numeric %*infmt*s are denoted by a leading percent sign (%) followed optionally by a string of the form w or $w.d$, where w and d stand for two integers. The first integer, w, specifies the width of the format. The second integer, d, specifies the number of digits that are to follow the decimal point. d must be less than or equal to w. Finally, a character denoting the format type (f, g, or e) is appended. For example, %9.2f specifies an f format that is nine characters wide and has two digits following the decimal point.

Numeric formats

The f format indicates that infile is to attempt to read the data as a number. When you do not specify the %*infmt* in the dictionary, infile assumes the %f format. The width, w, being missing means that infile is to attempt to read the data in free format.

As it starts reading each observation, infile reads a record into its buffer and sets a column pointer to 1, indicating that it is currently on the first column. When infile processes a %f format, it moves the column pointer forward through white space. It then collects the characters up to the next occurrence of

white space and attempts to interpret those characters as a number. The column pointer is left at the first occurrence of white space following those characters. If the next variable is also free format, the logic repeats.

When you explicitly specify the field width w, as in `%`w`f`, `infile` does not skip leading white space. Instead, it collects the next w characters starting at the column pointer and attempts to interpret the result as a number. The column pointer is left at the old value of the column pointer plus w, that is, on the first character following the specified field.

▷ Example 9: Specifying the width of fields

If the data above were stored in `highway.raw`, we could create the following dictionary to read the data:

begin `highway.dct`, example 6

```
infile dictionary using highway {
        acc_rate   %4f  "Acc. Rate/Million Miles"
        spdlimit   %2f  "Speed Limit (mph)"
        acc_pts    %3f  "Access Pts/Mile
}
```

end `highway.dct`, example 6

When we explicitly indicate the field width, `infile` does not skip intervening characters. The first four columns are used for the variable `acc_rate`, the next two for `spdlimit`, and the last three for `acc_pts`.

◁

❑ Technical note

The d specification in the `%`w`.`d`f` indicates the number of *implied* decimal places in the data. For instance, the string `212` read in a `%3.2f` format represents the number 2.12. Do *not* specify d unless your data have elements of this form. The w alone is sufficient to tell `infile` how to read data in which the decimal point is explicitly indicated.

When you specify d, Stata takes it only as a suggestion. If the decimal point is explicitly indicated in the data, that decimal point always overrides the d specification. Decimal points are also not implied if the data contain an `E`, `e`, `D`, or `d`, indicating scientific notation.

Fields are right-justified before implying decimal points. Thus '`2 `', '` 2 `', and '`  2`' are all read as 0.2 by the `%3.1f` format.

❑

❑ Technical note

The `g` and `e` formats are the same as the `f` format. You can specify any of these letters interchangeably. The letters `g` and `e` are included as a convenience to those familiar with Fortran, in which the `e` format indicates scientific notation. For example, the number 250 could be indicated as `2.5E+02` or `2.5D+02`. Fortran programmers would refer to this as an `E7.5` format, and in Stata, this format would be indicated as `%7.5e`. In Stata, however, you need specify only the field width w, so you could read this number by using `%7f`, `%7g`, or `%7e`.

The `g` format is really a Fortran output format that indicates a freer format than `f`. In Stata, the two formats are identical.

Throughout this section, you may freely substitute the `g` or `e` formats for the `f` format.

❑

❑ Technical note

Be careful to distinguish between %*fmts* and %*infmts*. %*fmts* are also known as *display* formats—they describe how a variable is to look when it is displayed; see **[U] 12.5 Formats: Controlling how data are displayed**. %*infmts* are also known as *input* formats—they describe how a variable looks when you input it. For instance, there is an output date format, %td, but there is no corresponding input format. (See **[U] 25 Working with dates and times** for recommendations on how to read dates.) For the other formats, we have attempted to make the input and output definitions as similar as possible. Thus we include g, e, and f %*infmts*, even though they all mean the same thing, because g, e, and f are also %*fmts*.

❑

String formats

The s and S formats are used for reading strings. The syntax is %ws or %wS, where the w is optional. If you do not specify the field width, your strings must either be enclosed in quotes (single or double) or not contain any characters other than letters, numbers, and "_".

This may surprise you, but the s format can be used for reading numeric variables, and the f format can be used for reading string variables! When you specify the field width, w, in the %wf format, all embedded blanks in the field are removed before the result is interpreted. They are not removed by the %ws format.

For instance, the %3f format would read "- 2", "-2 ", or " -2" as the number -2. The %3s format would not be able to read "- 2" as a number, because the sign is separated from the digit, but it could read " -2" or "-2 ". The %wf format removes blanks; datasets written by some Fortran programs separate the sign from the number.

There are, however, some side effects of this practice. The string "2 2" will be read as 22 by a %3f format. Most Fortran compilers would read this number as 202. The %3s format would issue a warning and store a *missing* value.

Now consider reading the string "a b" into a string variable. Using a %3s format, Stata will store it as it appears: a b. Using a %3f format, however, it will be stored as ab—the middle blank will be removed.

%wS is a special case of %ws. A string read with %ws will have leading and trailing blanks removed, but a string read with %wS will not have them removed.

Examples using the %s format are provided below, after we discuss specifying column and line numbers.

Specifying column and line numbers

_column() jumps to the specified column. For instance, the documentation of some dataset indicates that the variable age is recorded as a two-digit number in column 47. You could read this by coding

```
_column(47) age %2f
```

After typing this, you are now at column 49, so if immediately following age there were a one-digit number recording sex as 0 or 1, you could code

```
_column(47) age %2f
            sex %1f
```

or, if you wanted to be explicit about it, you could instead code

```
_column(47) age %2f
_column(49) sex %1f
```

It makes no difference. If at column 50 there were a one-digit code for `race` and you wanted to read it but skip reading the `sex` code, you could code

```
_column(47) age %2f
_column(50) race %1f
```

You could equivalently skip forward using `_skip()`:

```
_column(47) age %2f
_skip(1)    race %1f
```

One advantage of `column()` over `_skip` is that it lets you jump forward or backward in a record. If you wanted to read `race` and then `age`, you could code

```
_column(50) race %1f
_column(47) age %2f
```

If the data you are reading have multiple lines per observation (sometimes said as multiple records per observation), you can tell `infile` how many lines per record there are by using `_lines()`:

```
_lines(4)
```

`_lines()` appears only once in a dictionary. Good style says that it should be placed near the top of the dictionary, but Stata does not care.

When you want to go to a particular line, include the `_line()` directive. In our example, let's assume that `race`, `sex`, and `age` are recorded on the second line of each observation:

```
_lines(4)
_line(2)
    _column(47) age %2f
    _column(50) race %1f
```

Let's assume that `id` is recorded on line 1.

```
_lines(4)
_line(1)
    _column(1)  id  %4f
_line(2)
    _column(47) age %2f
    _column(50) race %1f
```

`_line()` works like `_column()` in that you can jump forward or backward, so these data could just as well be read by

```
_lines(4)
_line(2)
    _column(47) age %2f
    _column(50) race %1f
_line(1)
    _column(1)  id  %4f
```

Remember that this dataset has four lines per observation, and yet we have never referred to `line(3)` or `line(4)`. That is okay. Also, at the end of our dictionary, we are on line 1, not line 4. That is okay, too. `infile` will still get to the next observation correctly.

❑ Technical note

Another way to move between records is _newline(). _newline() is to _line() as _skip() is to _column(), which is to say, _newline() can only go forward. There is one difference: _skip() has its uses, whereas _newline() is useful only for backward capability with older versions of Stata.

_skip() has its uses because sometimes we think in columns and sometimes we think in widths. Some data documentation might include the sentence, "At column 54 are recorded the answers to the 25 questions, with one column allotted to each." If we want to read the answers to questions 1 and 5, it would indeed be natural to code

```
_column(54) q1 %1f
_skip(3)
            q5 %1f
```

Nobody has ever read data documentation with the statement, "Demographics are recorded on record 2, and two records after that are the income values." The documentation would instead say, "Record 2 contains the demographic information and record 4, income." The _newline() way of thinking is based on what is convenient for the computer, which does, after all, have to move past a certain number of records. That, however, is no reason for making you think that way.

Before that thought occurred to us, Stata users specified _newline() to go forward a number of records. They still can, so their old dictionaries will work. When you use _newline() and do not specify _lines(), you must move past the correct number of records so that, at the end of the dictionary, you are on the last record. In this mode, when Stata reexecutes the dictionary to process the next observation, it goes forward one record.

❑

Examples of reading fixed-format files

▷ Example 10: A file with two lines per observation

In this example, each observation occupies two lines. The first 2 observations in the dataset are

```
John Dunbar                    10001  101 North 42nd Street
1010111111
Sam K. Newey Jr.               10002  15663 Roustabout Boulevard
0101000000
```

The first observation tells us that the name of the respondent is John Dunbar; that his ID is 10001; that his address is 101 North 42nd Street; and that his answers to questions 1–10 were yes, no, yes, no, yes, yes, yes, yes, yes, and yes.

The second observation tells us that the name of the respondent is Sam K. Newey Jr.; that his ID is 10002; that his address is 15663 Roustabout Boulevard; and that his answers to questions 1–10 were no, yes, no, yes, no, no, no, no, no, and no.

To see the layout within the file, we can temporarily add two rulers to show the appropriate columns:

```
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8
John Dunbar                    10001  101 North 42nd Street
1010111111
Sam K. Newey Jr.               10002  15663 Roustabout Boulevard
0101000000
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8
```

Each observation in the data appears in two physical lines within our text file. We had to check in our editor to be sure that there really were new-line characters (for example, "hard returns") after the address. This is important because some programs will wrap output for you so that one line may appear as many lines. The two seemingly identical files will differ in that one has a hard return and the other has a soft return added only for display purposes.

In our data, the name occupies columns 1–32; a person identifier occupies columns 33–37; and the address occupies columns 40–80. Our worksheet revealed that the widest address ended in column 80.

The text file containing these data is called `fname.txt`. Our dictionary file looks like this:

begin `fname.dct`

```
infile dictionary using fname.txt {
*
* Example reading in data where observations extend across more
* than one line.  The next line tells infile there are 2 lines/obs:
*
_lines(2)
*
                str50   name    %32s       "Name of respondent"
_column(33)     long    id      %5f        "Person id"
_skip(2)        str50   addr    %41s       "Address"
_line(2)
_column(1)      byte    q1      %1f        "Question 1"
                byte    q2      %1f        "Question 2"
                byte    q3      %1f        "Question 3"
                byte    q4      %1f        "Question 4"
                byte    q5      %1f        "Question 5"
                byte    q6      %1f        "Question 6"
                byte    q7      %1f        "Question 7"
                byte    q8      %1f        "Question 8"
                byte    q9      %1f        "Question 9"
                byte    q10     %1f        "Question 10"
}
```

end `fname.dct`

Up to five pieces of information may be supplied in the dictionary for each variable: the location of the data, the storage type of the variable, the name of the variable, the input format, and the variable label.

Thus the `str50` line says that the first variable is to be given a storage type of `str50`, called `name`, and is to have the variable label "Name of respondent". The `%32s` is the input format, which tells Stata how to read the data. The `s` tells Stata not to remove any embedded blanks; the 32 tells Stata to go across 32 columns when reading the data.

The next line says that the second variable is to be assigned a storage type of `long`, named `id`, and be labeled "Person id". Stata should start reading the information for this variable in column 33. The `f` tells Stata to remove any embedded blanks, and the 5 says to read across five columns.

The third variable is to be given a storage type of `str50`, called `addr`, and be labeled "Address". The `_skip(2)` directs Stata to skip two columns before beginning to read the data for this variable, and the `%41s` instructs Stata to read across 41 columns and not to remove embedded blanks.

`line(2)` instructs Stata to go to line 2 of the observation.

The remainder of the data is 0/1 coded, indicating the answers to the questions. It would be convenient if we could use a shorthand to specify this portion of the dictionary, but we must supply explicit directives.

❑ Technical note

In the preceding example, there were two pieces of information about location: where the data begin for each variable (the _column(), _skip(), _line()) and how many columns the data span (the %32s, %5f, %41s, %1f). In our dictionary, some of this information was redundant. After reading name, Stata had finished with 32 columns of information. Unless instructed otherwise, Stata would proceed to the next column—column 33—to begin reading information about id. The _column(33) was unnecessary.

The _skip(2) was necessary, however. Stata had read 37 columns of information and was ready to look at column 38. Although the address information does not begin until column 40, columns 38 and 39 contain blanks. Because these are leading blanks instead of embedded blanks, Stata would just ignore them without any trouble. The problem is with the %41s. If Stata begins reading the address information from column 38 and reads 41 columns, Stata would stop reading in column 78 ($78 - 41 + 1 = 38$), but the widest address ends in column 80. We could have omitted the _skip(2) if we had specified an input format of %43s.

The _line(2) was necessary, although we could have read the second line by coding _newline instead.

The _column(1) could have been omitted. After the _line(), Stata begins in column 1.

See the next example for a dataset in which both pieces of location information are required.

❑

▷ Example 11: Manipulating the column pointer

The following file contains six variables in a variety of formats. In the dictionary, we read the variables fifth and sixth out of order by forcing the column pointer.

begin example.dct

```
infile dictionary {
                          first   %3f
                 double   second  %2.1f
                          third   %6f
    _skip(2)     str4     fourth  %4s
    _column(21)           sixth %4.1f
    _column(18)           fifth %2f
}
1.2125.7e+252abcd 1 .232
1.3135.7   52efgh2    5
1.41457    52abcd 3 100.
1.5155.7D+252efgh04 1.7
16 16 .57  52abcd 5 1.71
```

end example.dct

Assuming that the above is stored in a file called example.dct, we can infile and list it by typing

```
. infile using example
infile dictionary {
                          first   %3f
                 double   second  %2.1f
                          third   %6f
    _skip(2)     str4     fourth  %4s
    _column(21)           sixth %4.1f
    _column(18)           fifth %2f
}
(5 observations read)
```

```
. list
```

	first	second	third	fourth	sixth	fifth
1.	1.2	1.2	570	abcd	.232	1
2.	1.3	1.3	5.7	efgh	.5	2
3.	1.4	1.4	57	abcd	100	3
4.	1.5	1.5	570	efgh	1.7	4
5.	16	1.6	.57	abcd	1.71	5

◁

Reading fixed-block files

❑ Technical note

The `_lrecl(#)` directive is used for reading datasets that do not have end-of-line delimiters (carriage return, line feed, or some combination of these). Such datasets are typical of IBM mainframes, where they are known as fixed block, or FB. The abbreviation LRECL is IBM mainframe jargon for logical record length.

In a fixed-block dataset, each # characters are to be interpreted as a record. For instance, consider the data

```
1 21
2 42
3 63
```

In fixed-block format, these data might be recorded as

begin `mydata.ibm`

```
1 212 423 63
```

end `mydata.ibm`

and you would be told, on the side, that the LRECL is 4. If you then pass along that information to `infile`, it can read the data:

begin `mydata.dct`

```
infile dictionary using mydata.ibm {
        _lrecl(4)
        int     id
        int     age
}
```

end `mydata.dct`

When you do not specify the `_lrecl(#)` directive, `infile` assumes that each line ends with the standard text EOL delimiter (which can be a line feed, a carriage return, a line feed followed by a carriage return, or a carriage return followed by a line feed). When you specify `_lrecl(#)`, `infile` reads the data in blocks of # characters and then acts as if that is a line.

A common mistake in processing fixed-block datasets is to use an incorrect LRECL value, such as 160 when it is really 80. To understand what can happen, pretend that you thought the LRECL in your data was 6 rather than 4. Taking the characters in groups of 6, the data appear as

```
1 212
423 63
```

Stata cannot verify that you have specified the correct LRECL, so if the data appear incorrect, verify that you have the correct number.

The maximum LRECL `infile` allows is 524,275.

❑

Reading EBCDIC files

In the previous section, we discussed the `_lrecl(#)` directive that is often necessary for files that originated on mainframes and do not have end-of-line delimiters.

Such files sometimes are not even plain text files. Sometimes, these files have an alternate character encoding known as extended binary coded decimal interchange code (EBCDIC). The EBCDIC encoding was created in the 1960s by IBM for its mainframes.

Because EBCDIC is a different character encoding, we cannot even show you a printed example; it would be unreadable. Nevertheless, Stata can convert EBCDIC files to ASCII (see [D] **filefilter**) and can read data from EBCDIC files.

If you have a data file encoded with EBCDIC, you undoubtedly also have a description of it from which you can create a dictionary that includes the LRECL of the file (EBCDIC files do not typically have end-of-line delimiters) and the character positions of the fields in the file. You create a dictionary for an EBCDIC file just as you would for a plain text file, using the Do-file Editor or another text editor, and being sure to use the `_lrecl()` directive in the dictionary to specify the LRECL. You then simply specify the `ebcdic` option for `infile`, and Stata will convert the characters in the file from EBCDIC to ASCII on the fly:

```
. infile using mydict, ebcdic
```

Also see

[D] **infile (free format)** — Import unformatted text data

[D] **infix (fixed format)** — Import text data in fixed format

[D] **export** — Overview of exporting data from Stata

[D] **import** — Overview of importing data into Stata

[U] 22 Entering and importing data