file — Read and write text and binary files

Description	Syntax	Options	Remarks and examples
Stored results	Reference	Also see	

## Description

file is a programmer's command and should not be confused with import delimited (see [D] import delimited), infile (see [D] infile (free format) or [D] infile (fixed format)), and infix (see [D] infix (fixed format)), which are the usual ways that data are brought into Stata. file allows programmers to read and write both text and binary files, so file could be used to write a program to input data in some complicated situation, but that would be an arduous undertaking.

Files are referred to by a file *handle*. When you open a file, you specify the file handle that you want to use; for example, in

. file open myfile using example.txt, write

myfile is the file handle for the file named example.txt. From that point on, you refer to the file by its handle. Thus

. file write myfile "this is a test" \_n

would write the line "this is a test" (without the quotes) followed by a new line into the file, and

. file close myfile

would then close the file. You may have multiple files open at the same time, and you may access them in any order.

# **Syntax**

#### Open file

```
file open handle using filename, \{\underline{r}ead | \underline{w}rite | \underline{r}ead \underline{w}rite \}
```

```
\left[\left[\underline{t}ext | \underline{b}inary\right] \left[replace | append \right] all\right]
```

#### Read file

file <u>r</u>ead handle [specs]

#### Write to file

file write handle [specs]

Change current location in file

file seek  $handle \{ query | tof | eof | \# \}$ 

Set byte order of binary file

file set *handle* byteorder { hilo | lohi | 1 | 2 }

Close file

```
file close { handle | _all }
```

List file type, status, and name of handle

file query

where specs for text output is

"string" or '"string"' (exp) (parentheses are required) %fmt(exp) (see [D] format about %fmt) \_skip(#) \_\_column(#) \_\_newline[(#)] \_\_char(#) (0 ≤ # ≤ 255) \_tab[(#)] \_\_page[(#)] \_\_dup(#) specs for text input is localmacroname,

*specs* for binary output is

%{8 4}z	(exp)	
$\{4 2 1\}b[s u]$	(exp)	
%#s	"text"	$(1 \leq \# \leq \texttt{max\_macrolen})$
%#s	<i>'"text"</i>	
%#s	(exp)	

and specs for binary input is

%{8 4}z	scalarname	
$\{4 2 1\}b[s u]$	scalarname	
%#s	localmacroname	$(1 \le \# \le max\_macrolen)$

collect is allowed with file query, file read, and file seek; see [U] 11.1.10 Prefix commands.

# Options

read, write, or read write is required; they specify how the file is to be opened. If the file is opened read, you can later use file read but not file write; if the file is opened write, you can later use file write but not file read. If the file is opened read write, you can then use both.

read write is more flexible, but most programmers open files purely read or purely write because that is all that is necessary; it is safer and it is faster.

When a file is opened read, the file must already exist, or an error message will be issued. The file is positioned at the top (tof), so the first file read reads at the beginning of the file. Both local files and files over the net may be opened for read.

When a file is opened write and the replace or append option is not specified, the file must not exist, or an error message will be issued. The file is positioned at the top (tof), so the first file write writes at the beginning of the file. Net files may not be opened for write.

When a file is opened write and the replace option is also specified, it does not matter whether the file already exists; the existing file, if any, is erased beforehand.

When a file is opened write and the append option is also specified, it also does not matter whether the file already exists; the file will be reopened or created if necessary. The file will be positioned at the append point, meaning that if the file existed, the first file write will write at the first byte past the end of the previous file; if there was no previous file, file write begins writing at the first byte in the file. file seek may not be used with write append files.

When a file is opened read write, it also does not matter whether the file exists. If the file exists, it is reopened. If the file does not exist, a new file is created. Regardless, the file will be positioned at the top of the file. You can use file seek to seek to the end of the file or wherever else you desire. Net files may not be opened for read write.

Before opening a file, you can determine whether it exists by using confirm file; see [P] confirm.

text and binary determine how the file is to be treated once it is opened. text is the default. With text, files are assumed to be composed of lines of characters, with each line ending in a line-end character. The character varies across operating systems, being line feed under Unix, carriage return under Mac, and carriage return/line feed under Windows. file understands all the ways that lines might end when reading and assumes that lines are to end in the usual way for the computer being used when writing.

The alternative to text is binary, meaning that the file is to be viewed merely as a stream of bytes. With binary, there is an issue of byte order; consider the number 1 written as a 2-byte integer. On some computers (called hilo), it is written as "00 01", and on other computers (called lohi), it is written as "01 00" (with the least significant byte written first). There are similar issues for 4-byte integers, 4-byte floats, and 8-byte floats.

file assumes that the bytes are ordered in the way natural to the computer being used. file set can be used to vary this assumption. file set can be issued immediately after file open, or later, or repeatedly.

- replace and append are allowed only when the file is opened for write (which does not include read write). They determine what is to be done if the file already exists. The default is to issue an error message and not open the file. See the description of the options read, write, and read write above for more details.
- all is allowed when the file is opened for write or for read write. It specifies that, if the file needs to be created, the permissions on the file are to be set so that it is readable by everybody.

### Text output specifications

- "string" and '"string"' write string into the file, without the surrounding quotes.
- (*exp*) evaluates the expression *exp* and writes the result into the file. If the result is numeric, it is written with a %10.0g format, but with leading and trailing spaces removed. If *exp* evaluates to a string, the resulting string is written, with no extra leading or trailing blanks.
- % fmt (exp) evaluates expression exp and writes the result with the specified % fmt. If exp evaluates to a string, % fmt must be a string format, and, correspondingly, if exp evaluates to a real, a numeric format must be specified. Do not confuse Stata's standard display formats with the binary formats %b and %z described elsewhere in this entry. file write here allows Stata's display formats described in [D] format and allows the centering extensions (for example, %~20s) described in [P] display.
- \_skip(#) inserts # blanks into the file. If  $\# \le 0$ , nothing is written;  $\# \le 0$  is not considered an error.
- \_column(#) writes enough blanks to skip forward to column # of the line; if # refers to a prior column, nothing is displayed. The first column of a line is numbered 1. Referring to a column less than 1 is not considered an error; nothing is displayed then.
- \_newline [(#)], which may be abbreviated \_n[(#)], outputs one end-of-line character if # is not specified or outputs the specified number of end-of-line characters. The end-of-line character varies according to your operating system, being line feed under Unix, carriage return under Mac, and the two characters carriage return/line feed under Windows. If  $\# \le 0$ , no end-of-line character is output.
- \_char(#) outputs one ASCII character, being the one given by the ASCII code # specified. # must be between 0 and 255, inclusive.
- \_tab[(#)]outputs one tab character if # is not specified or outputs the specified number of tab characters. Coding \_tab is equivalent to coding \_char(9).

\_page[(#)]outputs one page feed character if # is not specified or outputs the specified number of page feed characters. Coding \_page is equivalent to coding \_char(12). The page feed character is often called Control-L.

\_dup(#) specified that the next directive is to be executed (duplicated) # times. # must be greater than or equal to 0. If # is equal to zero, the next element is not displayed.

## **Remarks and examples**

Remarks are presented under the following headings:

Use of file Use of file with tempfiles Writing text files Reading text files Use of seek when writing or reading text files Writing and reading binary files Writing binary files Reading binary files Use of seek when writing or reading binary files Appendix A.1. Useful commands and functions for use with file Appendix A.2. Actions of binary output formats with out-of-range values

## Use of file

file provides low-level access to file I/O. You open the file, use file read or file write repeatedly to read or write the file, and then close the file with file close:

file open ...
...
file read or file write ...
file read or file write ...
...
file close ...

Do not forget to close the file. Open files tie up system resources. Also, for files opened for writing, the contents of the file probably will not be fully written until you close the file.

Typing file close \_all will close all open files, and the clear all command (see [D] clear) closes all files as well. These commands, however, should not be included in programs that you write; they are included to allow the user to reset Stata when programmers have been sloppy.

If you use file handles obtained from tempname (see [P] macro), the file will be automatically closed when the ado-file terminates:

tempname myfile
file open 'myfile' using ...

This is the only case when not closing the file is appropriate. Use of temporary names for file handles offers considerable advantages because programs can be stopped because of errors or because the user presses *Break*.

#### Use of file with tempfiles

In the rare event that you file open a tempfile, you must obtain the handle from tempname; see [P] **macro**. Temporary files are automatically deleted when the ado- or do-file ends. If the file is erased before it is closed, significant problems are possible. Using a tempname will guarantee that the file is properly closed beforehand:

tempname myfile
tempfile tfile
file open 'myfile' using "'tfile'"...

### Writing text files

This is easy to do:

file open handle using filename, write text
file write handle ...
...
file close handle

The syntax of file write is similar to that of display; see [P] **display**. The significant difference is that expressions must be bound in parentheses. In display, you can code

#### display 2+2

but using file write, you must code

file write *handle* (2+2)

The other important difference between file write and display is that display assumes you want the end-of-line character output at the end of each display (and display provides \_continue for use when you do not want this), but file write assumes you want an end-of-line character only when you specify it. Thus rather than coding "file write *handle* (2+2)", you probably want to code

file write handle (2+2) \_n

Because Stata outputs end-of-line characters only where you specify, coding

file write *handle* "first part is " (2+2) \_n

has the same effect as coding

file write handle "first part is "
file write handle (2+2) \_n

or even

file write *handle* "first part is " file write *handle* (2+2) file write *handle* n

There is no limit to the line length that file write can write because, as far as file write is concerned, \_n is just another character. The \_col(#) directive, however, will lose count if you write lines of more than 2,147,483,646 characters (\_col(#) skips forward to the specified column). In general, we recommend that you do not write lines longer than 165,199 characters because reading lines longer than that is more difficult using file read.

We say that \_n is just another character, but we should say character or characters. \_n outputs the appropriate end-of-line character for your operating system, meaning the two-character carriage return followed by line feed under Windows, the one-character carriage return under Mac, and the one-character line feed under Unix.

### **Reading text files**

The commands for reading text files are similar to those for writing them:

file open *handle* using *filename*, read text file read handle localmacroname file close *handle* 

The file read command has one syntax:

file read handle localmacroname

One line is read from the file, and it is put in *localmacroname*. For instance, to read a line from the file myfile and put it in the local macro line, you code

#### file read myfile line

Thereafter in your code, you can refer to 'line' to obtain the contents of the line just read. The following program will do a reasonable job of displaying the contents of the file, putting line numbers in front of the lines:

```
program ltype
       version 19.5
                          // (or version 19 if you do not have StataNow)
        local 0 "using '0'"
        syntax using/
        tempname fh
        local linenum = 0
        file open 'fh' using '"'using'"', read
        file read 'fh' line
        while r(eof)==0 {
                local linenum = 'linenum' + 1
                display %4.0f 'linenum' _asis '" 'macval(line)'"'
                file read 'fh' line
        file close 'fh'
end
```

In the program above, we used tempname to obtain a temporary name for the file handle. Doing that, we ensure that the file will be closed, even if the user presses *Break* while our program is displaying lines, and so never executes file close 'fh'. In fact, our file close 'fh' line is unnecessary.

We also used r(eof) to determine when the file ends. file read sets r(eof) to contain 0 before end of file and 1 once end of file is encountered; see Stored results below.

We included \_asis in the display in case the file contained braces or SMCL commands. These would be interpreted, and we wanted to suppress that interpretation so that ltype would display lines exactly as written in the file; see [P] smcl. We also used the macval() macro function to obtain what was in 'line' without recursively expanding the contents of line.

### Use of seek when writing or reading text files

You may use file seek when reading or writing text files, although, in fact, it is seldom used, except with read write files, and even then, it is seldom used with text files.

See Use of seek when writing or reading binary files below for a description of file seek—seek works the same way with both text and binary files—and then bear the following in mind:

- The # in "file seek *handle* #" refers to byte position, not line number. "file seek *handle* 5" means to seek to the fifth byte of the file, not the fifth line.
- When calculating byte offsets by hand, remember that the end-of-line character is 1 byte under Mac and Unix but is 2 bytes under Windows.
- Rewriting a line of an text file works as expected only if the new and old lines are of the same length.

#### Writing and reading binary files

Consider whether you wish to read this section. There are many potential pitfalls associated with binary files, and, at least in theory, a poorly written binary-I/O program can cause Stata to crash.

Binary files are made up of binary elements, of which Stata can understand the following:

Element	Corresponding format
single- and multiple-character strings	%1s and %#s
signed and unsigned 1-byte binary integers	%1b, %1bs, and %1bu
signed and unsigned 2-byte binary integers	%2b, %2bs, and %2bu
signed and unsigned 4-byte binary integers	%4b, %4bs, and %4bu
4-byte IEEE floating-point numbers	%4z
8-byte IEEE floating-point numbers	%8z

The differences between all of these types are only of interpretation. For instance, the decimal number 72, stored as a 1-byte binary integer, also represents the character H. If a file contained the 1-byte integer 72 and you were to read the byte by using the format %1s, you would get back the character "H", and if a file contained the character "H" and you were to read the byte by using the format %1s, you would get back the character "H", and if a file contained the character "H" and you were to read the byte by using the format %1s, you would get back the number 72; 72 and H are indistinguishable in that they represent the same bit pattern. Whether that bit pattern represents 72 or H depends on the format you use, meaning the interpretation you give to the field.

Similar equivalence relations hold between the other elements. A binary file is nothing more than a sequence of unsigned 1-byte integers, where those integers are sometimes given different interpretations or are grouped and given an interpretation. In fact, all you need is the format %1bu to read or write anything. The other formats, however, make programming more convenient.

Format	Length	Туре	Minimum	Maximum	Missing values?
%1bu	1	unsigned byte	0	255	no
%1bs	1	signed byte	-127	127	no
%1b	1	Stata byte	-127	100	yes
%2bu	2	unsigned short int	0	65,535	no
%2bs	2	signed short int	-32,767	32,767	no
%2b	2	Stata int	-32,767	32,740	yes
%4bu	4	unsigned int	0	4,294,967,295	no
%4bs	4	signed int	-2,147,483,647	2,147,483,647	no
%4b	4	Stata long	-2,147,483,647	2,147,483,620	yes
%4z	4	float	$-10^{38}$	$10^{38}$	yes
%8z	8	double	$-10^{307}$	$10^{307}$	yes

When you write a binary file, you must decide on the format that you will use for every element that you will write. When you read a binary file, you must know ahead of time the format that was used for each element.

### Writing binary files

As with text files, you open the file, write repeatedly, and then close the file:

file open handle using filename, write binary
file write handle ...
...
file close handle

The file write command may include the following elements:

%{8 4}z	(exp)	
$\{4 2 1\}b[s u]$	(exp)	
%#s	"text"	$(1 \le \# \le \max\_macrolen)$
%#s	<i>'"text"</i> ,	
%#s	(exp)	

For instance, to write "test file" followed by 2, 2 + 2, and  $3 \times 2$  represented in its various forms, you could code

. file write *handle* %9s "test file" %8z (2) %4b (2+2) %1bu (3\*2)

```
. file write handle %9s "test file"
      . file write handle %8z (2) %4b (2+2) %1bu (3*2)
or even
      . file write handle %9s "test file"
      . file write handle %8z (2)
      . file write handle %4b (2+2) %1bu (3*2)
```

etc.

or

You write strings with the %#s format and numbers with the %b or %z formats. Concerning strings, the # in %#s should be greater than or equal to the length of the string to be written in bytes. If # is too small, only that many characters of the string will be written. Thus

. file write *handle* %4s "test file"

would write "test" into the file and leave the file positioned at the fifth byte. There is nothing wrong with coding that (the "test" can be read back easily enough), but this is probably not what you intended to write.

Also concerning strings, you can output string literals—just enclose the string in quotes—or you can output the results of string expressions. Expressions, as for using file write to output text files, must be enclosed in parentheses:

. file write handle %4s (substr(a,2,6))

The following program will output a user-specified matrix to a user-specified file; the syntax of the command being implemented is

mymatout1 *matname* using *filename* [, replace]

and the code is

```
program mymatout1
        version 19.5
                          // (or version 19 if you do not have StataNow)
        gettoken mname 0 : 0
        syntax using/ [, replace]
        local r = rowsof('mname')
        local c = colsof('mname')
        tempname hdl
        file open 'hdl' using '"'using'"', 'replace' write binary
        file write 'hdl' %2b ('r') %2b ('c')
        forvalues i=1(1)'r' {
                  forvalues j=1(1)'c' {
                            file write 'hdl' %8z ('mname'['i','j'])
                  }
        7
        file close 'hdl'
end
```

A significant problem with mymatout1 is that, if we wrote a matrix on our Unix computer (an Intelbased computer) and copied the file to a SPARC-based computer, we would discover that we could not read the file. Intel computers write multiple-byte numbers with the least-significant byte first; SPARCbased computers write the most-significant byte first. Who knows what your computer does? Thus even though there is general agreement across computers on how numbers and characters are written, this byte-ordering difference is enough to stop binary files.

file can handle this problem for you, but you have to insert a little code. The recommended procedure is this: before writing any numbers in the file, write a field saying which byte order this computer uses (see byteorder() in [FN] **Programming functions**). Later, when we write the command to read the file, it will read the ordering that we recorded. We will then tell file which byte ordering the file is using, and file itself will reorder the bytes if that is necessary. There are other ways that we could handle this—such as always writing in a known byte order—but the recommended procedure is better because it is, on average, faster. Most files are read on the same computer that wrote them, and thus the computer wastes no time rearranging bytes then.

The improved version of mymatout1 is

```
program mymatout2
                                  // (or version 19 if you do not have StataNow)
               version 19.5
               gettoken mname 0 : 0
               syntax using/ [, replace]
               local r = rowsof('mname')
               local c = colsof('mname')
               tempname hdl
               file open 'hdl' using '"'using'"', 'replace' write binary
 /* new */
               file write 'hdl' %1b (byteorder())
               file write 'hdl' %2b ('r') %2b ('c')
               forvalues i=1(1)'r' {
                         forvalues j=1(1)'c' {
                                 file write 'hdl' %8z ('mname'['i','j'])
                         7
               }
               file close 'hdl'
end
```

byteorder() returns 1 if the machine is hilo and 2 if lohi, but all that matters is that it is small enough to fit in a byte. The important thing is that we write this number using %1b, about which there is no byte-ordering disagreement. What we do with this number we will deal with later.

The second significant problem with our program is that it does not write a signature. Binary files are difficult to tell apart: they all look like binary junk. It is important that we include some sort of marker at the top saying who wrote this file and in what format it was written. That is called a signature. The signature that we will use is

```
mymatout 1.0.0
```

We will write that 14-byte-long string first thing in the file so that later, when we write mymatin, we can read the string and verify that it contains what we expect. Signature lines should always contain a generic identity (mymatout here) along with a version number, which we can change if we modify the output program to change the output format. This way, the wrong input program cannot be used with a more up-to-date file format.

Our improved program is

```
program mymatout3
               version 19.5
                                 // (or version 19 if you do not have StataNow)
               gettoken mname 0 : 0
               syntax using/ [, replace]
               local r = rowsof('mname')
               local c = colsof('mname')
               tempname hdl
        file open 'hdl' using '"'using'"', 'replace' write binary
               file write 'hdl' %14s "mymatout 1.0.0"
 /* new */
               file write 'hdl' %1b (byteorder())
               file write 'hdl' %2b ('r') %2b ('c')
               forvalues i=1(1)'r' {
                         forvalues j=1(1)'c' {
                                   file write 'hdl' %8z ('mname'['i','j'])
                         7
               }
               file close 'hdl'
end
```

This program works well. After we wrote the corresponding input routine (see Reading binary files below), however, we noticed that our restored matrices lacked their original row and column names, which led to a final round of changes:

```
program mymatout4
                                 // (or version 19 if you do not have StataNow)
               version 19.5
               gettoken mname 0 : 0
               syntax using/ [, replace]
               local r = rowsof('mname')
               local c = colsof('mname')
               tempname hdl
               file open 'hdl' using '"'using'"', 'replace' write binary
 /* changed */ file write 'hdl' %14s "mymatout 1.0.1"
               file write 'hdl' %1b (byteorder())
               file write 'hdl' %2b ('r') %2b ('c')
 /* new */
               local names : rownames 'mname'
 /* new */
               local len : length local names
               file write 'hdl' %4b ('len') %'len's '"'names'"'
/* new */
 /* new */
               local names : colnames 'mname'
 /* new */
               local len : length local names
 /* new */
               file write 'hdl' %4b ('len') %'len's '"'names'"'
               forvalues i=1(1)'r' {
                       forvalues j=1(1)'c' {
                               file write 'hdl' %8z ('mname'['i', 'j'])
                       7
               }
               file close 'hdl'
```

end

In this version, we added the lines necessary to write the row and column names into the file. We write the row names by coding

local names : rownames 'mname' local len : length local names file write 'hdl' %4b ('len') %'len's '"'names'"' and we similarly write the column names. The interesting thing here is that we need to write a string into our binary file for which the length of the string varies. One solution would be

file write 'hdl' %165199s '"'mname'"'

but that would be inefficient because, in general, the names are much shorter than 165,199 bytes. The solution is to obtain the length of the string to be written and then write the length into the file. In the above code, macro 'len' contains the length, we write 'len' as a 4-byte integer, and then we write the string using a %'len's format. Consider what happens when 'len' is, say, 50. We write 50 into the file, and then we write the string using a %50s format. Later, when we read back the file, we can reverse this process, reading the length and then using the appropriate format.

We also changed the signature from "mymatout 1.0.0" to "mymatout 1.0.1" because the file format changed. Making that change ensures that an old read program does not attempt to read a more modern format (and so produce incorrect results).

#### Technical note

You may write strings using %#s formats that are narrower than, equal to, or wider than the length of the string being written. When the format is too narrow, only that many characters of the string are written. When the format and string are of the same width, the entire string is written. When the format is wider than the string, the entire string is written, and then the excess positions in the file are filled with binary zeros.

Binary zeros are special in strings because binary denotes the end of the string. Thus when you read back the string, even if it was written in a field that was too wide, it will appear exactly as it appeared originally.

#### **Reading binary files**

You read binary files just as you wrote them,

```
file open handle using filename, read binary
file read handle ...
...
file close handle
```

When reading them, you must be careful to specify the same formats as you did when you wrote the file.

The program that will read the matrices written by mymatout1, presented below, has the syntax

```
mymatin1 matname filename
```

and the code is

```
program mymatin1
                                 // (or version 19 if you do not have StataNow)
               version 19.5
               gettoken mname 0 : 0
               syntax using/
               tempname hdl
               file open 'hdl' using '"'using'"', read binary
               tempname val
               file read 'hdl' %2b 'val'
               local r = 'val'
               file read 'hdl' %2b 'val'
               local c = 'val'
               matrix 'mname' = J('r', 'c', 0)
               forvalues i=1(1)'r' {
                       forvalues j=1(1)'c' {
                               file read 'hdl' %8z 'val'
                               matrix 'mname'['i', 'j'] = 'val'
                       }
               }
               file close 'hdl'
end
```

When file read reads numeric values, they are always stored into scalars (see [P] scalar), and you specify the name of the scalar directly after the binary numeric format. Here we are using the scalar named 'val', where 'val' is a name that we obtained from tempname. We could just as well have used a fixed name, say, myscalar, so the first file read would read

file read 'hdl' %2b myscalar

We would similarly substitute myscalar everywhere 'val' appears, but that would make our program less elegant. If the user had previously stored a value under the name myscalar, our values would replace it.

In the second version of mymatout, we included the byte order. The improved version of mymatin is

```
program mymatin2
                                 // (or version 19 if you do not have StataNow)
               version 19.5
               gettoken mname 0 : 0
               syntax using/
               tempname hdl
               file open 'hdl' using '"'using'"', read binary
               tempname val
               file read 'hdl' %1b 'val'
 /* new */
 /* new */
               local border = 'val'
               file set 'hdl' byteorder 'border'
/* new */
               file read 'hdl' %2b 'val'
               local r = 'val'
               file read 'hdl' %2b 'val'
               local c = 'val'
               matrix 'mname' = J('r', 'c', 0)
               forvalues i=1(1)'r' {
                       forvalues j=1(1)'c' {
                               file read 'hdl' %8z 'val'
                               matrix 'mname'['i', 'j'] = 'val'
                      }
               }
               file close 'hdl'
```

We simply read back the value we recorded and then file set it. We cannot directly file set *handle* byteorder 'val' because 'val' is a scalar, and the syntax for file set byteorder is

file set *handle* byteorder {hilo|lohi|1|2}

That is, file set is willing to see a number (1 and hilo mean the same thing, as do 2 and lohi), but that number must be a literal (the character 1 or 2), so we had to copy 'val' into a macro before we could use it. Once we set the byte order, however, we could from then on depend on file to reorder the bytes for us should that be necessary.

In the third version of mymatout, we added a signature. In the modification below, we read the signature by using a %14s format. Strings are copied into local macros, and we must specify the name of the local macro following the format:

```
program mymatin3
                                 // (or version 19 if you do not have StataNow)
               version 19.5
               gettoken mname 0 : 0
               syntax using/
               tempname hdl
               file open 'hdl' using '"'using'"', read binary
               file read 'hdl' %14s signature
 /* new */
/* new */
               if "'signature'" != "mymatout 1.0.0" {
/* new */
                       disp as err "file not mymatout 1.0.0"
/* new */
                       exit 610
/* new */
               }
               tempname val
               file read 'hdl' %1b 'val'
               local border = 'val'
               file set 'hdl' byteorder 'border'
               file read 'hdl' %2b 'val'
               local r = 'val'
               file read 'hdl' %2b 'val'
               local c = 'val'
               matrix 'mname' = J('r', 'c', 0)
               forvalues i=1(1)'r' {
                       forvalues j=1(1)'c' {
                               file read 'hdl' %8z 'val'
                               matrix 'mname'['i', 'j'] = 'val'
                       }
               7
               file close 'hdl'
end
```

In the fourth and final version, we wrote the row and column names. We wrote the names by first preceding them with a 4-byte integer recording their width:

```
program mymatin4
               version 19.5
                                  // (or version 19 if you do not have StataNow)
               gettoken mname 0 : 0
               syntax using/
               tempname hdl
               file open 'hdl' using '"'using'"', read binary
               file read 'hdl' %14s signature
 /* changed */ if "'signature'" != "mymatout 1.0.1" {
 /* changed */
                       disp as err "file not mymatout 1.0.1"
                        exit 610
               }
               tempname val
               file read 'hdl' %1b 'val'
               local border = 'val'
               file set 'hdl' byteorder 'border'
               file read 'hdl' %2b 'val'
               local r = 'val'
               file read 'hdl' %2b 'val'
               local c = 'val'
               matrix 'mname' = J('r', 'c', 0)
               file read 'hdl' %4b 'val'
 /* new */
 /* new */
               local len = 'val'
 /* new */
               file read 'hdl' %'len's names
               matrix rownames 'mname' = 'names'
 /* new */
 /* new */
               file read 'hdl' %4b 'val'
               local len = 'val'
 /* new */
               file read 'hdl' %'len's names
 /* new */
 /* new */
               matrix colnames 'mname' = 'names'
               forvalues i=1(1)'r' {
                       forvalues j=1(1)'c' {
                                file read 'hdl' %8z 'val'
                                matrix 'mname'['i', 'j'] = 'val'
                        }
               3
               file close 'hdl'
end
```

#### Use of seek when writing or reading binary files

Nearly all I/O programs are written without using file seek. file seek changes your location in the file. Ordinarily, you start at the beginning of the file and proceed sequentially through the bytes. file seek lets you back up or skip ahead.

file seek *handle* query actually does not change your location in the file; it merely returns in scalar r(loc) the current position, with the first byte in the file being numbered 0, the second 1, and so on. In fact, all the file seek commands return r(loc), but file seek query is unique because that is all it does.

file seek *handle* tof moves to the beginning (top) of the file. This is useful with read files when you want to read the file again, but you can seek to tof even with write files and, of course, with read write files. (Concerning read files: you can seek to top, or any point, before or after the end-of-file condition is raised.)

file seek *handle* eof moves to the end of the file. This is useful only with write files (or read write files) but may be used with read files, too.

file seek *handle* # moves to the specified position. # is measured in bytes from the beginning of the file and is in the same units as reported in r(loc). 'file seek *handle* 0' is equivalent to 'file seek *handle* tof'.

## Technical note

When a file is opened write append, you may not use file seek. If you need to seek in the file, open the file read write instead.

### Appendix A.1. Useful commands and functions for use with file

- When opening a file read write or write append, file's actions differ depending upon whether the file already exists. confirm file (see [P] confirm) can tell you whether a file exists; use it before opening the file.
- To obtain the length of strings when writing binary files, use the macro function length:

local length : length local mystr
file write handle % 'length's '"'mystr'"'

See Macro functions for parsing in [P] macro for details.

• To write portable binary files, we recommend writing in natural byte order and recording the byte order in the file. Then the file can be read by reading the byte order and setting it: Writing:

file write handle %1b (byteorder())

Reading:

tempname mysca
file read handle %1b 'mysca'
local b\_order = 'mysca'
file set handle byteorder 'b\_order'

The byteorder() function returns 1 or 2, depending on whether the computer being used records data in hilo or lohi format. See [FN] **Programming functions**.

#### Appendix A.2. Actions of binary output formats with out-of-range values

Say that you write the number 2,137 with a %1b format. What value will you later get back when you read the field with a %1b format? Here the answer is ., Stata's missing value, because the %1b format is a variation of %1bs that supports Stata's missing value. If you wrote 2,137 with %1bs, it would read back as 127; if you wrote it with %1bu, it would read back as 255.

			Value written v	when value is
Format	Min value	Max value	Too small	Too large
%1bu	0	255	0	255
%1bs	-127	127	-127	127
%1b	-127	100	•	•
%2bu	0	65,535	0	65,535
%2bs	-32,767	32,767	-32,767	32,767
%2b	-32,767	32,740		
%4bu	0	4,294,967,295	0	4,294,967,295
%4bs	-2,147,483,647	2,147,483,647	-2,147,483,647	2,147,483,647
%4b	-2,147,483,647	2,147,483,620		
%4z	$-10^{38}$	$10^{38}$	•	
%8z	$-10^{307}$	$10^{307}$	•	•

In general, in the Stata variation, missing values are supported, and numbers outside the range are written as missing. In the remaining formats, the minimum or maximum is written as

In the above table, if you write a missing value, take that as writing a value larger than the maximum allowed for the type.

If you write a noninteger value with an integer format, the result will be truncated to an integer. For example, writing 124.75 with a %2b format is the same as writing 124.

## Stored results

file read stores the following in r():

Macros r(status) (if text file) win line read; line ended in cr-lf mac line read; line ended in cr	alars r(eof)	1 on end of file, 0 otherwise	
unix line read; line ended in lf split line read; line was too long and so sp none line read; line was not terminated eof line not read because of end of file	acros r(status) (if text file)	win mac unix split none eof	line read; line ended in cr-lf line read; line ended in cr line read; line ended in lf line read; line was too long and so split line read; line was not terminated line not read because of end of file

r(status) = split indicates that c(macrolen)-1(33maxvar+199 for Stata/MP and Stata/SE, 165,199 for Stata/BE) characters of the line were returned and that the next file read will pick up where the last read left off.

r(status) = none indicates that the entire line was returned, that no line-end character was found, and that the next file read will return r(status) = eof.

If r(status) = eof (r(eof) = 1), then the local macro into which the line was read contains "". The local macro containing "", however, does not imply end of file because the line might simply have been empty. file seek stores the following in r():
Scalars
 r(loc) current position of the file
file query stores the following in r():
Scalars
 r(N) number of open files

## Reference

Slaymaker, E. 2005. Using the file command to produce formatted output for other applications. Stata Journal 5: 239-247.

# Also see

- [P] display Display strings and values of scalar expressions
- [D] filefilter Convert ASCII or binary patterns in a file
- [D] hexdump Display hexadecimal report on file
- [D] import Overview of importing data into Stata
- [D] import delimited Import and export delimited text data
- [D] infix (fixed format) Import text data in fixed format
- [M-4] IO I/O functions

Stata, Stata Press, and Mata are registered trademarks of StataCorp LLC. Stata and Stata Press are registered trademarks with the World Intellectual Property Organization of the United Nations. StataNow and NetCourseNow are trademarks of StataCorp LLC. Other brand and product names are registered trademarks or trademarks of their respective companies. Copyright © 1985–2025 StataCorp LLC, College Station, TX, USA. All rights reserved.



For suggested citations, see the FAQ on citing Stata documentation.