Title

stata.com

fopen( ) — File I/O

Description

These functions read and write files. First, open the file and get back a file handle (*fh*). The file handle, which is nothing more than an integer, is how you refer to the file in the calls to other file I/O functions. When you are finished, close the file.

Most file I/O functions come in two varieties: without and with an underscore in front of the name, such as `fopen()` and `_fopen()`, and `fwrite()` and `_fwrite()`.

In functions without a leading underscore, errors cause execution to be aborted. For instance, you attempt to open a file for read and the file does not exist. Execution stops. Or, having successfully opened a file, you attempt to write into it and the disk is full. Execution stops. When execution stops, the appropriate error message is presented.

In functions with the leading underscore, execution continues and no error message is displayed; it is your responsibility (1) to verify that things went well and (2) to take the appropriate action if they did not. Concerning (1), some underscore functions return a status value; others require that you call `fstatus()` to obtain the status information.

You can mix and match use of underscore and nonunderscore functions, using, say, `_fopen()` to open a file and `fread()` to read it, or `fopen()` to open and `_fwrite()` to write.

Syntax

real scalar `fopen(`*string scalar fn*`,` *mode*`)`

real scalar `fopen(`*string scalar fn*`,` *mode*`,` *public*`)`

real scalar `_fopen(`*string scalar fn*`,` *mode*`)`

real scalar `_fopen(`*string scalar fn*`,` *mode*`,` *public*`)`

where

mode: *string scalar* containing `"r"`, `"w"`, `"rw"`, or `"a"`

public: optional *real scalar* containing zero or nonzero

In what follows, *fh* is the value returned by `fopen()` or `_fopen()`:

void `fclose(`*fh*`)`
real scalar `_fclose(`*fh*`)`

string scalar `fget(`*fh*`)`
string scalar `_fget(`*fh*`)`
string scalar `fgetnl(`*fh*`)`
string scalar `_fgetnl(`*fh*`)`

string scalar `fread(`*fh*`,` *real scalar len*`)`
string scalar `_fread(`*fh*`,` *real scalar len*`)`

void `fput(`*fh*`,` *string scalar s*`)`
real scalar `_fput(`*fh*`,` *string scalar s*`)`

void `fwrite(`*fh*`,` *string scalar s*`)`
real scalar `_fwrite(`*fh*`,` *string scalar s*`)`

matrix `fgetmatrix(`*fh*`[ ,` *real scalar isstrict* `])`
matrix `_fgetmatrix(`*fh*`[ ,` *real scalar isstrict* `])`
void `fputmatrix(`*fh*`,` *transmorphic matrix X*`)`
real scalar `_fputmatrix(`*fh*`,` *transmorphic matrix X*`)`

real scalar `fstatus(`*fh*`)`

real scalar `ftell(`*fh*`)`
real scalar `_ftell(`*fh*`)`
void `fseek(`*fh*`,` *real scalar offset*`,` *real scalar whence*`)`
real scalar `_fseek(`*fh*`,` *real scalar offset*`,` *real scalar whence*`)`

(*whence* is coded −1, 0, or 1, meaning from start of file, from current position, or from end of file; *offset* is signed: positive values mean after *whence* and negative values mean before)

void `ftruncate(`*fh*`)`
real scalar `_ftruncate(`*fh*`)`

Remarks and examples

stata.com

Remarks are presented under the following headings:

Opening and closing files

Functions

`fopen(`*string scalar fn*`,` *string scalar mode*`)`

`_fopen(`*string scalar fn*`,` *string scalar mode*`)`

`fopen(`*string scalar fn*`,` *string scalar mode*`,` *real scalar public*`)`

`_fopen(`*string scalar fn*`,` *string scalar mode*`,` *real scalar public*`)`

open a file. The file may be on a local disk, a network disk, or even on the web (such as https://www.stata.com/index.html). *fn* specifies the filename, and *mode* specifies how the file is to opened:

mode	Meaning
`"r"`	Open for reading; file must exist and be readable. File may be `"https://..."` file. File will be positioned at the beginning.
`"w"`	Open for writing; file must not exist and the directory be writable. File may not be `"https://..."` file. File will be positioned at the beginning.
`"rw"`	Open for reading and writing; file must either exist and be writable or not exist and directory be writable. File may not be `"https://..."` file. File will be positioned at the beginning (new file) or at the end (existing file).
`"a"`	Open for appending; file must either exist and be writable or not exist and directory be writable. File may not be `"https://..."` file. File will be positioned at the end.

Other values for *mode* cause `fopen()` and `_fopen()` to abort with an invalid-mode error.

Optional third argument *public* specifies whether the file, if it is being created, should be given permissions so that everyone can read it, or if it instead should be given the normal permissions. Not specifying *public*, or specifying *public* as 0, gives the file the normal permissions. Specifying *public* as nonzero makes the file publicly readable. *public* is relevant only when the file is being created, that is, is being opened `"w"`, or being opened `"rw"` and not previously existing.

`fopen()` returns a file handle; the file is opened or execution is aborted.

`_fopen()` returns a file handle or returns a negative number. If a negative number is returned, the file is not open, and the number indicates the reason. For `_fopen()`, there are a few likely possibilities

Negative value	Meaning
−601	file not found
−602	file already exists
−603	file could not be opened
−608	file is read-only
−691	I/O error

and there are many other possibilities. For instance, perhaps you attempted to open a file on the web (say, http://www.newurl.org/upinfo.doc) and the URL was not found, or the server refused to send back the file, etc. See *Error codes* below for a complete list of codes.

After opening the file, you use the other file I/O commands to read and write it, and then you close the file with `fclose()` or `_fclose()`. `fclose()` returns nothing; if the file cannot be closed, execution is aborted. `_fclose` returns 0 if successful, or a negative number otherwise. For `_fclose()`, the likely possibilities are

Negative value	Meaning
−691	filesystem I/O error

Reading from a file

You may read from a file opened `"r"` or `"rw"`. The commands to read are

`fget(`*fh*`)`

`fgetnl(`*fh*`)`

`fread(`*fh*`,` *real scalar len*`)`

and, of course,

`_fget(`*fh*`)`

`_fgetnl(`*fh*`)`

`_fread(`*fh*`,` *real scalar len*`)`

All functions, with or without an underscore, require a file handle be specified, and all the functions return a string scalar or they return `J(0,0,"")`, a 0×0 string matrix. They return `J(0,0,"")` on end of file and, for the underscore functions, when the read was not successful for other reasons. When using the underscore functions, you use `fstatus()` to obtain the status code; see *Error codes* below. The underscore read functions are rarely used because the only reason a read can fail is I/O

error, and there is not much that can be done about that except abort, which is exactly what the nonunderscore functions do.

`fget(`*fh*`)` is for reading text files; the next line from the file is returned, without end-of-line characters. (If the line is longer then 32,768 characters, the first 32,768 characters are returned.)

`fgetnl(`*fh*`)` is much the same as `fget()`, except that the new-line characters are not removed from the returned result. (If the line is longer then 32,768 characters, the first 32,768 characters are returned.)

`fread(`*fh*`, `*len*`)` is usually used for reading binary files and returns the next *len* characters (bytes) from the file or, if there are fewer than that remaining to be read, however many remain. (*len* may not exceed 9,007,199,254,740,991 [*sic*] on 64-bit computers; memory shortage for storing the result will arise long before this limit is reached on most computers.)

The following code reads and displays a file:

```
fh = fopen(filename, "r")
while ((line=fget(fh))!=J(0,0,"")) {
     printf("%s\n", line)
}
fclose(fh)
```

Writing to a file

You may write to a file opened `"w"`, `"rw"`, or `"a"`. The functions are

`fput(`*fh*`, `*string scalar s*`)`

`fwrite(`*fh*`, `*string scalar s*`)`

and, of course,

`_fput(`*fh*`, `*string scalar s*`)`

`_fwrite(`*fh*`, `*string scalar s*`)`

fh specifies the file handle, and *s* specifies the string to be written. `fput()` writes *s* followed by the new-line characters appropriate for your operating system. `fwrite()` writes *s* alone.

`fput()` and `fwrite()` return nothing; `_fput()` and `_fwrite()` return a real scalar equal to 0 if all went well or a negative error code; see *Error codes* below.

The following code copies text from one file to another:

```
fh_in  = fopen(inputname, "r")
fh_out = fopen(outputname, "w")
while ((line=fget(fh_in))!=J(0,0,"")) {
     fput(fh_out, line)
}
fclose(fh_out)
fclose(fh_in)
```

The following code reads a file (binary or text) and changes every occurrence of "a" to "b":

```
fh_in  = fopen(inputname, "r")
fh_out = fopen(outputname, "w")
while ((c=fread(fh_in, 1))!=J(0,0,"")) {
     fwrite(fh_out, (c=="a" ? "b" : c))
}
fclose(fh_out)
fclose(fh_in)
```

Reading and writing in the same file

You may read and write from a file opened "rw", using any of the read or write functions described above. When reading and writing in the same file, one often uses file repositioning functions, too; see *Repositioning in a file* below.

Reading and writing matrices

Functions

fputmatrix(*fh*, *transmorphic matrix X*)

and

_fputmatrix(*fh*, *transmorphic matrix X*)

will write a matrix to a file. In the usual fashion, fputmatrix() returns nothing (it aborts if there is an I/O error) and _fputmatrix() returns a scalar equal to 0 if all went well and a negative error code otherwise.

Functions

fgetmatrix(*fh*[, *real scalar isstrict*])

and

_fgetmatrix(*fh*[, *real scalar isstrict*])

will read a matrix written by fputmatrix() or _fputmatrix(). Both functions return the matrix read or return J(0,0,.) on end of file (both functions) or error (_fgetmatrix() only). Because J(0,0,.) could be the matrix that was written, distinguishing between that and end of file requires subsequent use of fstatus(). fstatus() will return 0 if fgetmatrix() or _fgetmatrix() returned a written matrix, −1 if end of file, or (after _fgetmatrix()) a negative error code.

For a Mata struct or class matrix, a matrix according to the current definition will be created, and the saved matrix will be used to initialize the new matrix based on member-name matching.

Optional argument *isstrict* affects the behavior of the functions if the matrix being read is a Mata struct or class matrix. When the argument is set and not zero, the current struct or class definition in memory will be checked against the saved matrix to ensure that all variable names, variable eltypes, and variable orgtypes match each other.

fputmatrix() writes matrices in a compact, efficient, and portable format; a matrix written on a Windows computer can be read back on a Mac or Unix computer and vice versa.

The following code creates a file containing three matrices,

```
fh = fopen(filename, "w")
fputmatrix(fh, a)
fputmatrix(fh, b)
fputmatrix(fh, c)
fclose(fh)
```

and the following code reads them back:

```
fh = fopen(filename, "r")
a = fgetmatrix(fh)
b = fgetmatrix(fh)
c = fgetmatrix(fh)
fclose(fh)
```

The following code saves a Mata class scalar to file,

```
class sA {
   real scalar r
   string scalar s
   static scalar sr
   void new()
}
a = sA()
a.r = 1
a.s = "sA instance"
fh = fopen(filename, "w")
fputmatrix(fh, a)
fclose(fh)
```

and the following code reads the class matrix back:

```
fh = fopen(filename, "r")
a = fgetmatrix(fh)
fclose(fh)
```

The contents of `a` depends on the current definition of class `sA` in memory. If the definition does not change, `a.r` will be 1 and `a.s` will be "sA instance". Note: Only regular variables are saved and read back; static variables and functions are not saved. Also, the `new()` function will not be called in the created class scalar. If the class definition has been changed,

```
class sA {
   real scalar r
   real scalar b
   static scalar sr
   void new()
}
```

the function will not read the matrix if optional argument *isstrict* is specified and not zero. Otherwise, a class `sA` scalar `a` according to the current definition will be created. (Note: `new()` will not be called.) Member variables with matching names and compatible eltypes and orgtypes will be initialized using the values in the saved matrix. In this example, `a.r` will be `1`, and `a.b` will be missing because `b` is not a member in the class `sA` definition when it was saved.

❑ Technical note

You may even write pointer matrices

```
mats = (&a, &b, &c, NULL)
fh = fopen(filename,"w")
fputmatrix(fh, mats)
fclose(fh)
```

and read them back:

```
fh = fopen(filename, "r")
mats = fgetmatrix(fh)
fclose(fh)
```

When writing pointer matrices, `fputmatrix()` writes `NULL` for any pointer-to-function value. It is also recommended that you do not write self-referential matrices (matrices that point to themselves, either directly or indirectly), although the elements of the matrix may be cross linked and even recursive themselves. If you are writing pointer matrix `p`, no element of `p`, `*p`, `**p`, etc., should contain `&p`. That one address cannot be preserved because in the assignment associated with reading back the matrix (the "*result* =" part of *result* `= fgetmatrix(fh)`), a new matrix with a different address is associated with the contents.

❑

Repositioning in a file

The function

`ftell(`*fh*`)`

returns a real scalar reporting where you are in a file, and function

`fseek(`*fh*`,` *real scalar offset*`,` *real scalar whence*`)`

changes where you are in the file to be *offset* bytes from the beginning of the file (*whence* $= -1$), *offset* bytes from the current position (*whence* $= 0$), or *offset* bytes from the end of the file (*whence* $= 1$).

Functions `_ftell()` and `_fseek()` do the same thing as `ftell()` and `fseek()`, the difference being that, rather than aborting on error, the underscore functions return negative error codes. `_ftell()` is pretty well useless as the only error that can arise is I/O error, and what else are you going to do other than abort? `_fseek()`, however, has a use, because it allows you to try out a repositioning and check whether it was successful. With `fseek()`, if the repositioning is not successful, execution is aborted.

Say you open a file for read:

```
fh = fopen(filename, "r")
```

After opening the file in mode `r`, you are positioned at the beginning of the file or, in the jargon of file processing, at position 0. Now say that you read 10 bytes from the file:

```
part1 = fread(fh, 10)
```

Assuming that was successful, you are now at position 10 of the file. Say that you next read a line from the file

```
line = fget(fh)
```

and assume that `fget()` returns `"abc"`. You are now at position 14 or 15. (No, not 13: `fget()` read the line and the new-line characters and returned the line. `abc` was followed by carriage return and line feed (two characters) if the file was written under Windows and by a carriage return or line feed alone (one character) if the file was written under Mac or Unix).

`ftell(`*fh*`)` and `_ftell(`*fh*`)` tell you where you are in the file. Coding

```
pos = ftell(fh)
```

would store 14 or 15 in `pos`. Later in your code, after reading more of the file, you could return to this position by coding

```
fseek(fh, pos, -1)
```

You could return to the beginning of the file by coding

```
fseek(fh, 0, -1)
```

and you could move to the end of the file by coding

```
fseek(fh, 0, 1)
```

`_fseek(`*fh*`, 0, 0)` is equivalent to `ftell(`*fh*`)`.

Repositioning functions cannot be used when the file has been opened `"a"`.

Truncating a file

Truncation refers to making a longer file shorter. If a file was opened `"w"` or `"rw"`, you may truncate it at its current position by using

```
ftruncate(fh)
```

or

```
_ftruncate(fh)
```

`ftruncate()` returns nothing; `_ftruncate()` returns 0 on success and otherwise returns a negative error code.

The following code shortens a file to its first 100 bytes:

```
fh = fopen(filename, "rw")
fseek(fh, 100, -1)
ftruncate(fh)
fclose(fh)
```

Error codes

If you use the underscore I/O functions, if there is an error, they will return a negative code. Those codes are

Negative code	Meaning
0	all is well
−1	end of file
−2	connection timed out
−601	file not found
−602	file already exists
−603	file could not be opened
−608	file is read-only
−610	file format error
−612	unexpected end of file
−630	web files not supported in this version of Stata
−631	host not found
−632	web file not allowed in this context
−633	may not write to web file
−660	proxy host not found
−661	host or file not found
−662	proxy server refused request to send
−663	remote connection to proxy failed
−665	could not set socket nonblocking
−669	invalid URL
−670	invalid network port number
−671	unknown network protocol
−672	server refused to send file
−673	authorization required by server
−674	unexpected response from server
−675	server reported server error
−676	server refused request to send
−677	remote connection failed
−678	could not open local network socket
−679	unexpected web error
−691	I/O error
−699	insufficient disk space
−3601	invalid file handle
−3602	invalid filename
−3603	invalid file mode
−3611	too many open files
−3621	attempt to write read-only file
−3622	attempt to read write-only file
−3623	attempt to seek append-only file
−3698	file seek error

Other codes in the −600 to −699 range are possible. The codes in this range correspond to the negative of the corresponding Stata return code; see [P] **error**.

Underscore functions that return a real scalar will return one of these codes if there is an error.

If an underscore function does not return a real scalar, then you obtain the outcome status using `fstatus()`. For instance, the read-string functions return a string scalar or `J(0,0,"")` on end of file. The underscore variants do the same, and they also return `J(0,0,"")` on error, meaning error looks like end of file. You can determine the error code using the function

```
fstatus(fh)
```

`fstatus()` returns 0 (no previous error) or one of the negative codes above.

`fstatus()` may be used after any underscore I/O command to obtain the current error status.

`fstatus()` may also be used after the nonunderscore I/O commands; then `fstatus()` will return -1 or 0 because all other problems would have stopped execution.

Conformability

`fopen(`*fn*`,` *mode*`,` *public*`)`, `_fopen(`*fn*`,` *mode*`,` *public*`)`:

fn:	1×1	
mode:	1×1	
public:	1×1	(optional)
result:	1×1	

`fclose(`*fh*`)`:

fh:	1×1	
result:	*void*	

`_fclose(`*fh*`)`:

fh:	1×1	
result:	1×1	

`fget(`*fh*`)`, `_fget(`*fh*`)`, `fgetnl(`*fh*`)`, `_fgetnl(`*fh*`)`:

fh:	1×1	
result:	1×1	or 0×0 if end of file

`fread(`*fh*`,` *len*`)`, `_fread(`*fh*`,` *len*`)`:

fh:	1×1	
len:	1×1	
result:	1×1	or 0×0 if end of file

`fput(`*fh*`,` *s*`)`, `fwrite(`*fh*`,` *s*`)`:

fh:	1×1	
s:	1×1	
result:	*void*	

`_fput(`*fh*`,` *s*`)`, `_fwrite(`*fh*`,` *s*`)`:

fh:	1×1	
s:	1×1	
result:	1×1	

`fgetmatrix(`*fh*`)`, `_fgetmatrix(`*fh*`)`:

fh:	1×1	
result:	$r \times c$	or 0×0 if end of file

`fputmatrix(`*fh*`, `*X*`)`:

fh:	1×1
X:	$r \times c$
result:	*void*

`_fputmatrix(`*fh*`, `*X*`)`:

fh:	1×1
X:	$r \times c$
result:	1×1

`fstatus(`*fh*`)`:

fh:	1×1
result:	1×1

`ftell(`*fh*`)`, `_ftell(`*fh*`)`:

fh:	1×1
result:	1×1

`fseek(`*fh*`, `*offset*`, `*whence*`)`:

fh:	1×1
offset:	1×1
whence:	1×1
result:	*void*

`_fseek(`*fh*`, `*offset*`, `*whence*`)`:

fh:	1×1
offset:	1×1
whence:	1×1
result:	1×1

`ftruncate(`*fh*`)`:

fh:	1×1
result:	*void*

`_ftruncate(`*fh*`)`:

fh:	1×1
result:	1×1

Diagnostics

`fopen(`*fn*`, `*mode*`)` aborts with error if *mode* is invalid or if *fn* cannot be opened or if an attempt is made to open too many files simultaneously.

`_fopen(`*fn*`, `*mode*`)` aborts with error if *mode* is invalid or if an attempt is made to open too many files simultaneously. `_fopen()` returns the appropriate negative error code if *fn* cannot be opened.

All remaining I/O functions—even functions with leading underscore—abort with error if *fh* is not a handle to a currently open file.

Also, the functions that do not begin with an underscore abort with error when a file was opened read-only and a request requiring write access is made, when a file is opened write-only and a request requiring read access is made, etc. See *Error codes* above; all problems except code -1 (end of file) cause the nonunderscore functions to abort with error.

Finally, the following functions will also abort with error for the following specific reasons:

`fseek(`*fh*`,` *offset*`,` *whence*`)` and `_fseek(`*fh*`,` *offset*`,` *whence*`)` abort with error if *offset* is outside the range ±9,007,199,254,740,991 on 64-bit computers or if *whence* is not −1, 0, or 1.

Also see

[M-5] **bufio( )** — Buffered (binary) I/O

[M-5] **cat( )** — Load file into string matrix `sprintf()` in

[M-5] **printf( )** — Format output

[M-4] **IO** — I/O functions