Title

stata.com

bufio( ) — Buffered (binary) I/O

Description

These functions manipulate buffers (string scalars) containing binary data and, optionally, perform I/O.

`bufio()` returns a control vector, *C*, that you pass to the other buffer functions. *C* specifies the byte order of the buffer and specifies how missing values are to be encoded. Despite its name, `bufio()` opens no files and performs no I/O. `bufio()` merely returns a vector of default values for use with the remaining buffer functions.

`bufbyteorder()` and `bufmissingvalue()` allow changing the defaults in *C*.

`bufput()` and `bufget()` copy elements into and out of buffers. No I/O is performed. Buffers can then be written by using `fwrite()` and read by using `fread()`; see [M-5] **fopen( )**.

`fbufput()` and `fbufget()` do the same, and they perform the corresponding I/O by using `fwrite()` or `fread()`.

`bufbfmtlen(`*bfmt*`)` and `bufbfmtisnum(`*bfmt*`)` are utility routines for processing *bfmts*; they are rarely used. `bufbfmtlen(`*bfmt*`)` returns the implied length, in bytes, of the specified *bfmt*, and `bufbfmtisnum(`*bfmt*`)` returns 1 if the *bfmt* is numeric, 0 if string.

Syntax

```
colvector C  =  bufio()

real scalar     bufbyteorder(C)
void            bufbyteorder(C, real scalar byteorder)
real scalar     bufmissingvalue(C)
void            bufmissingvalue(C, real scalar version)

void            bufput(C, B, offset, bfmt, X)

scalar          bufget(C, B, offset, bfmt)
rowvector       bufget(C, B, offset, bfmt, c)
matrix          bufget(C, B, offset, bfmt, r, c)

void            fbufput(C, fh, bfmt, X)

scalar          fbufget(C, fh, bfmt)
rowvector       fbufget(C, fh, bfmt, c)
matrix          fbufget(C, fh, bfmt, r, c)

real scalar     bufbfmtlen(string scalar bfmt)
real scalar     bufbfmtisnum(string scalar bfmt)
```

where

C: *colvector* returned by `bufio()`

B: *string scalar* (buffer)

offset: *real scalar* (buffer position, starts at 0)

fh: file handle returned by `fopen()`

bfmt: *string scalar* (binary format; see below)

r: *string scalar*

c: *string scalar*

X: value to be written; see *Remarks and examples*

bfmt may contain

bfmt	meaning
%{8|4}z	8-byte floating point or 4-byte floating point
%{4|2|1}b[s|u]	4-, 2-, or 1-byte integer; Stata, signed or unsigned
%#s	text string
%#S	binary string

Remarks and examples

stata.com

If you wish simply to read and write matrices, etc., see `fgetmatrix()` and `fputmatrix()` and the other functions in [M-5] **fopen( )**.

The functions documented here are of interest if

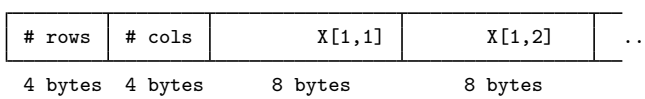
1. you wish to create your own binary-data format because you are writing routines in low-level languages such as FORTRAN or C and need to transfer data between your new routines and Stata, or
2. you wish to write a program to read and write the binary format of another software package.

These are advanced and tedious programming projects.

Remarks are presented under the following headings:

Basics

Let's assume that you wish to write a matrix to disk so you can move it back and forth from FORTRAN. You settle on a file format in which the number of rows and number of columns are first written as 4-byte integers, and then the values of the matrix are written as 8-byte doubles, by row:



One solution to writing matrices in such a format is

```
fh = fopen("filename", "w")
C  = bufio()
fbufput(C, fh, "%4b", rows(X))
fbufput(C, fh, "%4b", cols(X))
fbufput(C, fh, "%8z", X)
fclose(fh)
```

The code to read the matrix back is

```
fh   = fopen("filename", "r")
C    = bufio()
rows = fbufget(C, fh, "%4b")
cols = fbufget(C, fh, "%4b")
X    = fbufget(C, fh, "%8z", rows, cols)
fclose(fh)
```

Another solution, which would be slightly more efficient, is

```
fh = fopen("filename", "w")
C   = bufio()
buf = 8*char(0)
bufput(C, buf, 0, "%4b", rows(X))
bufput(C, buf, 4, "%4b", cols(X))
fwrite(C, buf)
fbufput(C, fh, "%8z", X)
fclose(fh)
```

and

```
fh   = fopen("filename", "r")
C    = bufio()
buf  = fread(fh, 8)
rows = bufget(C, buf, 0, "%4b")
cols = bufget(C, buf, 4, "%4b")
X    = fbufget(C, fh, "%8z", rows, cols)
fclose(fh)
```

What makes the above approach more efficient is that, rather than writing 4 bytes (the number of rows), and 4 bytes again (the number of columns), we created one 8-byte buffer and put the two 4-byte fields in it, and then we wrote all 8 bytes at once. We did the reverse when we read back the data: we read all 8 bytes and then broke out the fields. The benefit is minuscule here but, in general, writing longer buffers results in faster I/O.

In all the above examples, we wrote and read the entire matrix with one function call,

```
fbufput(C, fh, "%8z", X)
```

and

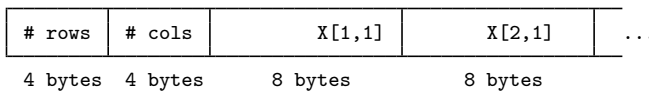
```
X    = fbufget(C, fh, "%8z", rows, cols)
```

Perhaps you would have preferred our having coded

```
for (i=1; i<=rows(X); i++) {
        for (j=1; j<=cols(X); j++) {
                fbufput(C, fh, "%8z", X[i,j])
        }
}
```

and perhaps you would have preferred our having coded something similar to read back the matrix. Had we done so, the results would have been the same.

If you are familiar with FORTRAN, you know that it records matrices in column-dominant order, rather than the row-dominant order used by Mata. It would be a little easier to code the FORTRAN side of things if we changed our file-format design to write columns first:



One way we could do that would be to write the loops ourselves:

```
fh = fopen("filename", "w")
C  = bufio()
fbufput(C, fh, "%4b", rows(X))
fbufput(C, fh, "%4b", cols(X))
for (j=1; j<=cols(X); i++) {
        for (i=1; i<=rows(X); j++) {
                fbufput(C, fh, "%8z", X[i,j])
        }
}
```

and

```
fh   = fopen("filename", "r")
C    = bufio()
rows = fbufget(C, fh, "%4b")
cols = fbufget(C, fh, "%4b")
X    = J(rows, cols, .)
for (j=1; j<=cols(X); i++) {
        for (i=1; i<=rows(X); j++) {
                X[i,j] = fbufget(C, fh, "%8z")
        }
}
```

We could do that, but there are more efficient and easier ways to proceed. For instance, we could simply transpose the matrix before writing and after reading, and if we do that transposition in place, our code will use no extra memory:

```
fh = fopen("filename", "w")
C  = bufio()
fbufput(C, fh, "%4b", rows(X))
fbufput(C, fh, "%4b", cols(X))
_transpose(X)
fbufput(C, fh, "%8z", X)
_transpose(X)
fclose(fh)
```

The code to read the matrices back is

```
fh   = fopen("filename", "r")
C    = bufio()
rows = fbufget(C, fh, "%4b")
cols = fbufget(C, fh, "%4b")
X    = fbufget(C, fh, "%8z", cols, rows)
_transpose(X)
fclose(fh)
```

Argument C

Argument *C* in

`bufput(`*C*`,` *B*`,` *offset*`,` *bfmt*`,` *X*`)`,

`bufget(`*C*`,` *B*`,` *offset*`,` *bfmt*`, ...)`,

`fbufput(`*C*`,` *fh*`,` *bfmt*`,` *X*`)`, and

`fbufget(`*C*`,` *fh*`,` *bfmt*`, ...)`

specifies the control vector. You obtain *C* by coding

```
C = bufio()
```

`bufio()` returns *C*, which is nothing more than a vector filled in with default values. The other buffer routines know how to interpret the vector. The vector contains two pieces of information:

1. The byte order to be used.
2. The missing-value coding scheme to be used.

Some computer hardware writes numbers left to right (for example, Sun), and other computer hardware writes numbers right to left (for example, Intel); see [M-5] **byteorder( )**. If you are going to write binary files, and if you want your files to be readable on all computers, you must write code to deal with this issue.

Many programmers ignore the issue because the programs they write are intended for one computer or on computers like the one they use. If that is the case, you can ignore the issue, too. The default byte order filled in by `bufio()` is the byte order of the computer you are using.

If you intend to read and write files across different computers, however, you will need to concern yourself with byte order, and how you do that is described in *Advanced issues* below.

The other issue you may need to consider is missing values. If you are writing a binary format that is intended to be used outside Stata, it is best if the values you write simply do not include missing values. Not all packages have them, and the packages that do don't agree on how they are encoded. In such cases, if the data you are writing include missing values, change the values to another value such as −1, 99, 999, or −9999.

If, however, you are writing binary files in Stata to be read back in Stata, you can allow Stata's missing values `.`, `.a`, `.b`, ..., `.z`. No special action is required. The missing-value scheme in *C* specifies how those missing values are encoded, and there is only one way right now, so there is in fact no issue at all. *C* includes the extra information in case Stata ever changes the way it encodes missing values so that you will have a way to read and write old-format files. How this process works is described in *Advanced issues*.

Arguments B and offset

Functions

bufput(*C*, *B*, *offset*, *bfmt*, *X*) and

bufget(*C*, *B*, *offset*, *bfmt*, ...)

do not perform I/O; they copy values into and out of the buffer. *B* specifies the buffer, and *offset* specifies the position within it.

B is a string scalar.

offset is an integer scalar specifying the position within *B*. Offset 0 specifies the first byte of *B*.

For bufput(), *B* must already exist and be long enough to receive the result, and it is usual to code something like

```
B = (4 + 4 + rows(X)*cols(X)*8) * char(0)
bufput(C, B, 0, "%4b", rows(X))
bufput(C, B, 4, "%4b", cols(X))
bufput(C, B, 8, "%8z", X)
```

Argument fh

Argument *fh* in

fbufput(*C*, *fh*, *bfmt*, *X*) and

fbufget(*C*, *fh*, *bfmt*, ...)

plays the role of arguments *B* and *offset* in bufput() and bufget(). Rather than copy into or out of a buffer, data are written to, or read from, file *fh*. *fh* is obtained from fopen(); see [M-5] **fopen( )**.

Argument bfmt

Argument *bfmt* in

bufput(*C*, *B*, *offset*, *bfmt*, *X*),

bufget(*C*, *B*, *offset*, *bfmt*, ...),

fbufput(*C*, *fh*, *bfmt*, *X*), and

fbufget(*C*, *fh*, *bfmt*, ...)

specifies how the elements are to be written or read.

bfmts for numeric data

The numeric *bfmts* are

bfmt	Interpretation
%8z	8-byte floating point
%4z	4-byte floating point
%4bu	4-byte unsigned integer
%4bs	4-byte signed integer
%4b	4-byte Stata integer
%2bu	2-byte unsigned integer
%2bs	2-byte signed integer
%2b	2-byte Stata integer
%1bu	1-byte unsigned integer
%1bs	1-byte signed integer
%1b	1-byte Stata integer

A Stata integer is the same as a signed integer, except that the largest 27 values are given the interpretation ., .a, .b, ..., .z.

bfmts for string data

The string *bfmts* are

bfmt	Interpretation
%#s	text string
%#S	binary string

where # represents the length of the string field. Examples include %8s and %639876S.

When writing, it does not matter whether you use %#s or %#S, the same actions are taken:

1. If the string being written is shorter than #, the field is padded with char(0).
2. If the string being written is longer than #, only the first # bytes of the string are written.

When reading, the distinction between %#s and %#S is important:

1. When reading with %#s, if char(0) appears within the first # bytes, the returned result is truncated at that point.
2. When reading with %#S, a full # bytes are returned in all cases.

Argument X

Argument *X* in

bufput(*C*, *B*, *offset*, *bfmt*, *X*) and

fbufput(*C*, *fh*, *bfmt*, *X*)

specifies the value to be written. *X* may be real or string and may be a scalar, vector, or matrix. If *X* is a vector, the elements are written one after the other. If *X* is a matrix, the elements of the first row are written one after the other, followed by the elements of the second row, and so on.

In

X = bufget(*C*, *B*, *offset*, *bfmt*, ...) and

X = fbufget(*C*, *fh*, *bfmt*, ...)

X is returned.

Arguments r and c

Arguments *r* and *c* are optional in the following:

X = bufget(*C*, *B*, *offset*, *bfmt*),

X = bufget(*C*, *B*, *offset*, *bfmt*, *c*),

X = bufget(*C*, *B*, *offset*, *bfmt*, *r*, *c*),

X = fbufget(*C*, *fh*, *bfmt*),

X = fbufget(*C*, *fh*, *bfmt*, *c*), and

X = fbufget(*C*, *fh*, *bfmt*, *r*, *c*).

If *r* is not specified, results are as if $r = 1$.

If *c* is not specified, results are as if $c = 1$.

Thus

X = bufget(*C*, *B*, *offset*, *bfmt*) and

X = fbufget(*C*, *fh*, *bfmt*)

read one element and return it, whereas

X = bufget(*C*, *B*, *offset*, *bfmt*, *c*) and

X = fbufget(*C*, *fh*, *bfmt*, *c*)

read *c* elements and return them in a column vector, and

X = bufget(*C*, *B*, *offset*, *bfmt*, *r*, *c*) and

X = fbufget(*C*, *fh*, *bfmt*, *r*, *c*)

read $r * c$ elements and return them in an $r \times c$ matrix.

Advanced issues

A properly designed binary-file format includes a signature line first thing in the file:

```
fh = fopen(filename, "w")
fwrite(fh, "MyFormat For Mats v. 1.0")
         /* ----+----1----+----2---- */
```

and

```
fh = fopen(filename, "r")
if (fread(fh, 24) != "MyFormat For Mats v. 1.0") {
        errprintf("%s not My-Format file\n", filename)
        exit(610)
}
```

If you are concerned with byte order and mapping of missing values, you should write the byte order and missing-value mapping in the file, write in natural byte order, and be prepared to read back in either byte order.

The code for writing is

```
fh = fopen(filename, "w")
fwrite(fh, "MyFormat For Mats v. 1.0")

C = bufio()
fbufput(C, fh, "%1bu", bufbyteorder(C))
fbufput(C, fh, "%2bu", bufmissingvalue(C))
```

and the corresponding code for reading the file is

```
fh = fopen(filename, "r")
if (fread(fh, 24) != "MyFormat For Mats v. 1.0") {
        errprintf("%s not My-Format file\n", filename)
        exit(610)
}

C = bufio()
bufbyteorder(C, fbufget(C, "%1bu"))
bufmissingvalue(C, fbufget(C, "%2bu"))
```

All we write in the file before recording the byte order are strings and bytes. This way, when we read the file later, we can set the byte order before reading any 2-, 4-, or 8-byte fields.

`bufbyteorder(`*C*`)`—`bufbyteorder()` with one argument—returns the byte-order encoding recorded in *C*. It returns 1 (meaning HILO) or 2 (meaning LOHI).

`bufbyteorder(`*C*`,` *value*`)`—`bufbyteorder()` with two arguments—resets the byte order recorded in *C*. Once reset, all buffer functions will automatically reverse bytes if necessary.

`bufmissingvalue()` works the same way. With one argument, it returns a code for the encoding scheme recorded in *C* (said code being the Stata release number multiplied by 100). With two arguments, it resets the code. Once the code is reset, all buffer routines used will automatically take the appropriate action.

Conformability

`bufio()`:

result:	*colvector*	

`bufbyteorder(`*C*`)`:

C:	*colvector*	made by `bufio()`
result:	1×1	containing 1 (HILO) or 2 (LOHI)

`bufbyteorder(`*C*`, `*byteorder*`)`:

C:	*colvector*	made by `bufio()`
byteorder:	1×1	containing 1 (HILO) or 2 (LOHI)
result:	*void*	

`bufmissingvalue(`*C*`)`:

C:	*colvector*	made by `bufio()`
result:	1×1	

`bufmissingvalue(`*C*`, `*version*`)`:

C:	*colvector*	made by `bufio()`
version:	1×1	
result:	*void*	

`bufput(`*C*`, `*B*`, `*offset*`, `*bfmt*`, `*X*`)`:

C:	*colvector*	made by `bufio()`
B:	1×1	
offset:	1×1	
bfmt:	1×1	
X:	$r \times c$	
result:	*void*	

`bufget(`*C*`, `*B*`, `*offset*`, `*bfmt*`)`:

C:	*colvector*	made by `bufio()`
B:	1×1	
offset:	1×1	
bfmt:	1×1	
result:	1×1	

`bufget(`*C*`, `*B*`, `*offset*`, `*bfmt*`, `*r*`)`:

C:	*colvector*	made by `bufio()`
B:	1×1	
offset:	1×1	
bfmt:	1×1	
r:	1×1	
result:	$1 \times c$	

`bufget(`*C*`,` *B*`,` *offset*`,` *bfmt*`,` *r*`,` *c*`)`:

C:	*colvector*	made by `bufio()`
B:	1×1	
offset:	1×1	
bfmt:	1×1	
r:	1×1	
c:	1×1	
result:	$r \times c$	

`fbufput(`*C*`,` *fh*`,` *bfmt*`,` *X*`)`:

C:	*colvector*	made by `bufio()`
fh:	1×1	
bfmt:	1×1	
X:	$r \times c$	
result:	*void*	

`fbufget(`*C*`,` *fh*`,` *bfmt*`)`:

C:	*colvector*	made by `bufio()`
fh:	1×1	
bfmt:	1×1	
result:	1×1	

`fbufget(`*C*`,` *fh*`,` *bfmt*`,` *r*`)`:

C:	*colvector*	made by `bufio()`
fh:	1×1	
bfmt:	1×1	
r:	1×1	
result:	$1 \times c$	

`fbufget(`*C*`,` *fh*`,` *bfmt*`,` *r*`,` *c*`)`:

C:	*colvector*	made by `bufio()`
fh:	1×1	
bfmt:	1×1	
r:	1×1	
c:	1×1	
result:	$r \times c$	

`bufbfmtlen(`*bfmt*`)`:

bfmt:	1×1
result:	1×1

`bufbfmtisnum(`*bfmt*`)`:

bfmt:	1×1
result:	1×1

Diagnostics

`bufio()` cannot fail.

`bufbyteorder(`*C*`)` cannot fail. `bufbyteorder(`*C*`,` *byteorder*`)` aborts with error if *byteorder* is not 1 or 2.

`bufmissingvalue(`*C*`)` cannot fail. `bufmissingvalue(`*C*`,` *version*`)` aborts with error if *version* $<$ 100 or *version* $>$ `stataversion()`.

`bufput(`*C*`,` *B*`,` *offset*`,` *bfmt*`,` *X*`)` aborts with error if *B* is too short to receive the result, *offset* is out of range, *bfmt* is invalid, or *bfmt* is a string format and *X* is numeric or vice versa. Putting a void matrix results in 0 bytes being inserted into the buffer and is not an error.

`bufget(`*C*`,` *B*`,` *offset*`,` *bfmt*`, ...)` aborts with error if *B* is too short, *offset* is out of range, or *bfmt* is invalid. Reading zero rows or columns results in a void returned result and is not an error.

`fbufput(`*C*`,` *fh*`,` *bfmt*`,` *X*`)` aborts with error if *fh* is invalid, *bfmt* is invalid, or *bfmt* is a string format and *X* is numeric or vice versa. Putting a void matrix results in 0 bytes being written and is not an error. I/O errors are possible; use `fstatus()` to detect them.

`fbufget(`*C*`,` *fh*`,` *bfmt*`, ...)` aborts with error if *fh* is invalid or *bfmt* is invalid. Reading zero rows or columns results in a void returned result and is not an error. End-of-file and I/O errors are possible; use `fstatus()` to detect them.

`bufbfmtlen(`*bfmt*`)` and `bufbfmtisnum(`*bfmt*`)` abort with error if *bfmt* is invalid.

Also see

[M-5] **fopen( )** — File I/O

[M-4] **io** — I/O functions