Stata help for dta

help dta

Title

[P] file formats .dta -- Description of .dta file format

Syntax

The information contained in this highly technical entry probably does not interest you. We describe in detail the format of Stata **.dta** datasets for those interested in writing programs in C or other languages that read and write them.

Remarks

Remarks are presented under the following headings:

1. Introduction

Stata-format datasets record data in a way generalized to work across computers that do not agree on how data are recorded. Thus the same dataset may be used, without translation, on different computers (Windows, Unix, and Macintosh computers). Given a computer, datasets are divided into two categories: native-format and foreign-format datasets. Stata uses the following two rules:

R1. On any computer, Stata knows how to write only native-format datasets.

R2. On all computers, Stata can read foreign-format as well as native-format datasets.

Rules R1 and R2 ensure that Stata users need not be concerned with dataset formats. If you are writing a program to read and write Stata datasets, you will have to determine whether you want to follow the same rules or instead restrict your program to operate on only native-format datasets. Since Stata follows rules R1 and R2, such a restriction would not be too limiting. If the user had a foreign-format dataset, he or she could enter Stata, **use** the data, and then save it again.

2. Versions and flavors of Stata

Stata is continually being updated, and these updates sometimes require changes be made to how Stata records **.dta** datasets. This document documents what are known as **format-114** datasets, the most modern format. Stata itself can read older formats, but whenever it writes a dataset, it writes in **114** format.

There are currently four flavors of Stata available: Small Stata, Stata/IC, Stata/SE, and Stata/MP. The same **114** format is used by all flavors. The difference is that datasets can be larger in some flavors.

3. Representation of strings

1. Strings in Stata may be from 1 to 244 bytes long.

2. Stata records a string with a trailing binary 0 (**\0**) delimiter if the length of the string is less than the maximum declared length. The string is recorded without the delimiter if the string is of the maximum length.

3. Leading and trailing blanks are significant.

4. Strings use ASCII encoding.

4. Representation of numbers

1. Numbers are represented as 1-, 2-, and 4-byte integers and 4- and 8-byte floats. In the case of floats, ANSI/IEEE Standard 754-1985 format is used.

2. Byte ordering varies across machines for all numeric types. Bytes are ordered either least-significant to most-significant, dubbed LOHI, or most-significant to least-significant, dubbed HILO. Pentiums, for instance, use LOHI encoding. Sun SPARC-based computers use HILO encoding. Itanium-based computers are interesting in that they can be either LOHI or HILO depending on operating system. Windows and Linux on Itanium use LOHI encoding. HP-UX on Itanium uses HILO encoding.

3. When reading a HILO number on a LOHI machine or a LOHI number on a HILO machine, perform the following before interpreting the number:

byte	no translation necessary
2-byte int	swap bytes 0 and 1
4-byte int	swap bytes 0 and 3, 1 and 2
4-byte float	swap bytes 0 and 3, 1 and 2
8-byte float	swap bytes 0 and 7, 1 and 6, 2 and 5, 3 and 4

4. For purposes of written documentation, numbers are written with the most significant byte listed first. Thus, **0x0001** refers to a 2-byte integer taking on the logical value 1 on all machines.

5. Stata has five numeric data types. They are

byte	1-byte signed int
int	2-byte signed int
long	4-byte signed int
float	4-byte IEEE float
double	8-byte IEEE float

6. Each type allows for 27 missing value codes, known as **.**, **.a**, **.b**, ..., **.z**. For each type, the range allowed for nonmissing values and the

missing values codes are

byte

minimum nonmissing	-127	(0x80)
maximum nonmissing	+100	(0x64)
code for .	+101	(0x65)
code for **.a**	+102	(0x66)
code for **.b**	+103	(0x67)
...		
code for **.z**	+127	(0x7f)

int

minimum nonmissing	-32767	(0x8000)
maximum nonmissing	+32740	(0x7fe4)
code for .	+32741	(0x7fe5)
code for **.a**	+32742	(0x7fe6)
code for **.b**	+32743	(0x7fe7)
...		
code for **.z**	+32767	(0x7fff)

long

minimum nonmissing	-2,147,483,647	(0x80000000)
maximum nonmissing	+2,147,483,620	(0x7fffffe4)
code for .	+2,147,483,621	(0x7fffffe5)
code for **.a**	+2,147,483,622	(0x7fffffe6)
code for **.b**	+2,147,483,623	(0x7fffffe7)
...		
code for **.z**	+2,147,483,647	(0x7fffffff)

float

minimum nonmissing	-1.701e+38	(-1.fffffeX+7e) *(sic)*
maximum nonmissing	+1.701e+38	(+1.fffffeX+7e)
code for .		(+1.000000X+7f)
code for **.a**		(+1.001000X+7f)
code for **.b**		(+1.002000X+7f)
...		
code for **.z**		(+1.01a000X+7f)

double

minimum nonmissing	-1.798e+308	(-1.fffffffffffffX+3ff)
maximum nonmissing	+8.988e+307	(+1.fffffffffffffX+3fe)
code for .		(+1.0000000000000X+3ff)
code for **.a**		(+1.0010000000000X+3ff)
code for **.b**		(+1.0020000000000X+3ff)
...		
code for **.z**		(+1.01a0000000000X+3ff)

Note that for **float**, all *z*>1.fffffeX+7e, and for **double**, all *z*>1.fffffffffffffX+3fe are considered to be missing values and it is merely a subset of the values that are labeled **.**, **.a**, **.b**, ..., **.z**. For example, a value between **.a** and **.b** is still considered to be missing and, in particular, all the values between **.a** and **.b** are known jointly as **.a_**. Nevertheless, the recording of those values should be avoided.

In the table above, we have used the {**+**|**-**}**1.***<digits>***X**{**+**|**-**}*<digits>* notation. The number to the left of the **X** is to be interpreted as a base-16 number (the period is thus the base-16 point) and the number to the right (also recorded in base 16) is to be interpreted as the power of 2 *(sic)*. For example,

```
1.01aX+3ff = (1.01a) * 2^(3ff)                          (base 16)
           = (1 + 0/16 + 1/16^2 + 10/16^3) * 2^1023     (base 10)
```

The {**+**|**-**}**1.***<digits>***X**{**+**|**-**}*<digits>* notation easily converts to IEEE 8-byte double: the **1** is the hidden bit, the digits to the right of

the hexadecimal point are the mantissa bits, and the exponent is the IEEE exponent in signed (removal of offset) form. For instance, pi = 3.1415927... is

```
                                    8-byte IEEE, HILO
                                   -----------------------
    pi = +1.921fb54442d18X+001 = 40 09 21 fb 54 44 2d 18

                                 = 18 2d 44 54 fb 21 09 40
                                   -----------------------
                                    8-byte IEEE, LOHI
```

Converting {**+**|**-**}**1.***<digits>***X**{**+**|**-**}***<digits>*** to IEEE 4-byte float is more difficult, but the same rule applies: the **1** is the hidden bit, the digits to the right of the hexadecimal point are the mantissa bits, and the exponent is the IEEE exponent in signed (removal of offset) form. What makes it more difficult is that the sign-and-exponent in the IEEE 4-byte format occupy 9 bits, which is not divisible by four, and so everything is shifted one bit. In float:

```
                                4-byte IEEE, HILO
                                   -----------
    pi = +1.921fb60000000X+001 = 40 49 0f db

                                 = db of 49 40
                                   -----------
                                4-byte IEEE, LOHI
```

The easiest way to obtain the above result is to first convert +1.921fb60000000X+001 to an 8-byte double and then convert the 8-byte double to a 4-byte float.

In any case, the relevant numbers are

V	value	HILO	LOHI
m	-1.fffffffffffX+3ff	ffefffffffffffff	fffffffffffefff
M	+1.fffffffffffX+3f3	7fdfffffffffffff	fffffffffffdf7f
.	+1.0000000000000X+3ff	7fe0000000000000	0000000000000e07f
.a	+1.0010000000000X+3ff	7fe0010000000000	0000000000001e07f
.b	+1.0020000000000X+3ff	7fe0020000000000	0000000000002e07f
.z	+1.01a0000000000X+3ff	7fe01a0000000000	00000000001ae07f
m	-1.fffffeX+7e	fefffffff	fffffffe
M	+1.fffffeX+7e	7efffffff	ffffff7e
.	+1.000000X+7f	7f000000	0000007f
.a	+1.001000X+7f	7f000800	0008007f
.b	+1.002000X+7f	7f001000	0010007f
.z	+1.01a000X+7f	7f00d000	00d0007f

5. Dataset format definition

Stata-format datasets contain five components, which are, in order,

1. Header
2. Descriptors
3. Variable Labels
4. Expansion Fields
5. Data
6. Value Labels

5.1 Header

The Header is defined as

Contents	Length	Format	Comments
ds_format	1	byte	contains 114 = 0x72
byteorder	1	byte	0x01 -> HILO, 0x02 -> LOHI
filetype	1	byte	0x01
unused	1	byte	0x01
nvar (number of vars)	2	int	encoded per **byteorder**
nobs (number of obs)	4	int	encoded per **byteorder**
data_label	81	char	dataset label, \0 terminated
time_stamp	18	char	date/time saved, \0 terminated
Total	109		

time_stamp[17] must be set to binary zero. When writing a dataset, you may record the time stamp as blank **time_stamp[0]**=\0), but you must still set **time_stamp[17]** to binary zero as well. If you choose to write a time stamp, its format is

dd Mon yyyy hh:*mm*

dd and *hh* may be written with or without leading zeros, but if leading zeros are suppressed, a blank must be substituted in their place.

5.2 Descriptors

The Descriptors are defined as

Contents	Length	Format	Comments
typlist	**nvar**	byte array	
varlist	33***nvar**	char array	
srtlist	2*(**nvar**+1)	int array	encoded per **byteorder**
fmtlist	49***nvar**	char array	
lbllist	33***nvar**	char array	

typlist stores the type of each variable, 1, ..., nvar. The types are encoded:

type	code
str1	1 = 0x01
str2	2 = 0x02
...	
str244	244 = 0xf4
byte	251 = 0xfb *(sic)*
int	252 = 0xfc
long	253 = 0xfd
float	254 = 0xfe
double	255 = 0xff

Stata stores five numeric types: **double**, **float**, **long**, **int**, and **byte**. If **nvar**==4, a **typlist** of 0xfcfffdfe indicates that variable 1 is an **int**, variable 2 a **double**, variable 3 a **long**, and variable 4 a **float**. Types above 0x01 through 0xf4 are used to represent strings. For example, a string with maximum length 8 would have type **0x08**. If **typlist** is read into the C-array **char typlist[]**, then **typlist[i-1]** indicates the type of variable **i**.

varlist contains the names of the Stata variables 1, ..., **nvar**, each up

to 32 characters in length, and each terminated by a binary zero (\0). For instance, if **nvar**==4,

```
0       33        66          99
|        |         |           |
vbl1\0...myvar\0...thisvar\0...lstvar\0...
```

would indicate that variable 1 is named **vbl1**, variable 2 **myvar**, variable 3 **thisvar**, and variable 4 **lstvar**. The byte positions indicated by periods will contain random numbers (and note that we have omitted some of the periods). If **varlist** is read into the C-array **char varlist[]**, then **&varlist[(i-1)*33]** points to the name of the **i**th variable.

srtlist specifies the sort-order of the dataset and is terminated by an (int) 0. Each 2 bytes is 1 int and contains either a variable number or zero. The zero marks the end of the **srtlist**, and the array positions after that contain random junk. For instance, if the data are not sorted, the first int will contain a zero and the ints thereafter will contain junk. If **nvar**==4, the record will appear as

```
0000.............
```

If the dataset is sorted by one variable **myvar** and if that variable is the second variable in the **varlist**, the record will appear as

```
00020000............  (if byteorder==HILO)
02000000............  (if byteorder==LOHI)
```

If the dataset is sorted by **myvar** and within **myvar** by **vbl1**, and if **vbl1** is the first variable in the dataset, the record will appear as

```
000200010000........  (if byteorder==HILO)
020001000000........  (if byteorder==LOHI)
```

If **srtlist** were read into the C-array **short int srtlist[]**, then **srtlist[0]** would be the number of the first sort variable or, if the data were not sorted, 0. If the number is not zero, **srtlist[1]** would be the number of the second sort variable or, if there is not a second sort variable, 0, and so on.

fmtlist contains the formats of the variables 1, ..., **nvar**. Each format is 49 bytes long and includes a binary zero end-of-string marker. For instance,

```
%9.0f\0.........................................%8.2f\0......
................................%20.0g\0.................
.....................%td\0..............................
..........%tcDDmonCCYY_HH:MM:SS.sss\0.....................
```

indicates that variable 1 has a **%9.0f** format, variable 2 a **%8.2f** format, variable 3 a **%20.0g** format, and so on. Note that these are Stata formats, not C formats.

1. Formats beginning with **%t** or **%-t** are Stata's date and time formats.

2. Stata has an old **%d** format notation and some datasets still have them. Format **%d**... is equivalent to modern format **%td**... and **%-d**... is equivalent to **%-td**...

3. Nondate formats ending in **gc** or **fc** are similar to C's **g** and **f** formats, but with commas. Most translation routines would ignore the ending **c** (change it to **\0**).

4. Formats may contain commas rather than period, such as **%9,2f**, indicating European format.

If **fmtlist** is read into the C-array **char fmtlist[]**, then **&fmtlist[12*(i-1)]** refers to the starting address of the format for the **i**th variable.

lbllist contains the names of the value formats associated with the variables 1, ..., **nvar**. Each value-format name is 33 bytes long and includes a binary zero end-of-string marker. For instance,

```
0   33        66   99
|    |         |    |
\0...yesno\0...\0...yesno\0...
```

indicates that variables 1 and 3 have no value label associated with them, whereas variables 2 and 4 are both associated with the value label named **yesno**. If **lbllist** is read into the C-array **char lbllist[]**, then **&lbllist[33*(i-1)]** points to the start of the label name associated with the **i**th variable.

5.3 Variable labels

The Variable Labels are recorded as

Contents	Length	Format	Comments
Variable 1's label	81	char	\0 terminated
Variable 2's label	81	char	\0 terminated
...			
Variable **nvar**'s label	81	char	\0 terminated
Total	81***nvar**		

If a variable has no label, the first character of its label is \0.

5.4 Expansion fields

The Expansion Fields are recorded as

Contents	Length	Format	Comments
data type	1	byte	coded, only 0 and 1 defined
len	4	int	encoded per **byteorder**
contents	len	varies	
data type	1	byte	coded, only 0 and 1 defined
len	4	int	encoded per **byteorder**
contents	len	varies	
data type	1	byte	code 0 means end
len	4	int	0 means end

Expansion fields conclude with code 0 and len 0; before the termination marker, there may be no or many separate data blocks. Expansion fields are used to record information that is unique to Stata and has no equivalent in other data management packages. Expansion fields are always optional when writing data and, generally, programs reading Stata datasets will want to ignore the expansion fields. The format makes this easy. When writing, write 5 bytes of zeros for this field. When reading, read five bytes; the last four bytes now tell you the size of the next read, which you discard. You then continue like this until you read 5 bytes of zeros.

The only expansion fields currently defined are type 1 records for variable's characteristics. The design, however, allows new types of expansion fields to be included in subsequent releases of Stata without changes in the data format since unknown expansion types can simply be skipped.

For those who care, the format of type 1 records is a binary-zero terminated variable name in bytes 0-32, a binary-zero terminated characteristic name in bytes 33-65, and a binary-zero terminated string defining the contents in bytes 66 through the end of the record.

5.5 Data

The Data are recorded as

Contents	Length	Format
obs 1, var 1	per **typlist**	per **typlist**
obs 1, var 2	per **typlist**	per **typlist**
...		
obs 1, var **nvar**	per **typlist**	per **typlist**
obs 2, var 1	per **typlist**	per **typlist**
obs 2, var 2	per **typlist**	per **typlist**
...		
obs 2, var **nvar**	per **typlist**	per **typlist**
.		
.		
obs **nobs**, var 1	per **typlist**	per **typlist**
obs **nobs**, var 2	per **typlist**	per **typlist**
...		
obs **nobs**, var **nvar**	per **typlist**	per **typlist**

The data are written as all the variables on the first observation, followed by all the data on the second observation, and so on. Each variable is written in its own internal format, as given in **typlist**. All values are written per **byteorder**. Strings are null terminated if they are shorter than the allowed space, but they are not terminated if they occupy the full width.

End-of-file may occur at this point. If it does, there are no value labels to be read. End-of-file may similarly occur between value labels. On end-of-file, all data have been processed.

5.6 Value labels

If there are no value labels, end-of-file will have occurred while reading the data. If there are value labels, each value label is written as

Contents	len	format	comment
len	4	int	length of **value_label_table**
labname	33	char	\0 terminated
padding	3		
value_label_table	**len**		see next table

and this is repeated for each value label included in the file. The format of the **value_label_table** is

Contents	len	format	comment

```
    ------------------------------------------------------------
    n                        4   int        number of entries
    txtlen                   4   int        length of txt[]
    off[]                  4*n   int array  txt[] offset table
    val[]                  4*n   int array  sorted value table
    txt[]               txtlen   char       text table
    ------------------------------------------------------------
```

len, **n**, **txtlen**, **off[]**, and **val[]** are encoded per **byteorder**. The maximum length of **txt[]** for a label is 32,000 characters. Stata is robust to datasets which might contain labels longer than this; labels which exceed the limit, if any, will be dropped during a **use**.

For example, the **value_label_table** for 1<->yes and 2<->no, shown in HILO format, would be

```
    byte position:  00 01 02 03   04 05 06 07   08 09 10 11   12 13 14 15
         contents:  00 00 00 02   00 00 00 07   00 00 00 00   00 00 00 04
          meaning:        n = 2    txtlen = 7    off[0] = 0    off[1] = 4

    byte position:  16 17 18 19   20 21 22 23   24 25 26 27 28 29 30
         contents:  00 00 00 01   00 00 00 02    y  e  s 00  n  o 00
          meaning:   val[0] = 1    val[1] = 2    txt --->
```

The interpretation is that there are **n**=2 values being mapped. The values being mapped are **val[0]**=1 and **val[1]**=2. The corresponding text for **val[0]** would be at **off[0]**=0 (and so be "**yes**") and for **val[1]** would be at **off[1]**=4 (and so be "**no**").

Interpreting this table in C is not as daunting as it appears. Let **(char *) p** refer to the memory area into which **value_label_table** is read. Assume your compiler uses 4-byte **int**s. The following manifests make interpreting the table easier:

```
    #define SZInt               4
    #define Off_n               0
    #define Off_nxtoff          SZInt
    #define Off_off             (SZInt+SZInt)
    #define Off_val(n)          (SZInt+SZInt+n*SZInt)
    #define Off_txt(n)          (Off_val(n) + n*SZInt)
    #define Len_table(n,nxtoff) (Off_txt(n) + nxtoff)

    #define Ptr_n(p)            ( (int *) ( ((char *) p) + Off_n ) )
    #define Ptr_nxtoff(p)       ( (int *) ( ((char *) p) + Off_nxtoff ) )
    #define Ptr_off(p)          ( (int *) ( ((char *) p) + Off_off ) )
    #define Ptr_val(p,n)        ( (int *) ( ((char *) p) + Off_val(n) ) )
    #define Ptr_txt(p,n)        ( (char *) ( ((char *) p) + Off_txt(n) ) )
```

It is now the case that **for(i=0; i < *Ptr_n(p); i++)**, the value ***Ptr_val(p,i)** is mapped to the character string **Ptr_txt(p,i)**.

Remember in allocating memory for ***p** that the table can be big. The limits are **n**=65,536 mapped values with each value being up to 81 characters long (including the null terminating byte). Such a table would be 5,823,712 bytes long. No user is likely to approach that limit and, in any case, after reading the 8 bytes preceding the table (**n** and **txtlen**), you can calculate the remaining length as 2*4*n+**txtlen** and **malloc()** the exact amount.

Constructing the table is more difficult. The easiest approach is to set arbitrary limits equal to or smaller than Stata's as to the maximum number of entries and total text length you will allow and simply declare the three pieces **off[]**, **val[]**, and **txt[]** according to those limits:

```
    int off[MaxValueForN] ;
    int val[MaxValueForN] ;
```

```
    char txt[MaxValueForTxtlen] ;
```

Stata's internal code follows a more complicated strategy of always keeping the table in compressed form and having a routine that will "add one position" in the table. This is slower but keeps memory requirements to be no more than the actual size of the table.

In any case, when adding new entries to the table, remember that **val[]** must be in ascending order: **val[0]** < **val[1]** < ... < **val[n]**.

It is not required that **off[]** or **txt[]** be kept in ascending order. We previously offered the example of the table that mapped 1<->yes and 2<->no:

```
    byte position:  00 01 02 03   04 05 06 07   08 09 10 11   12 13 14 15
         contents:  00 00 00 02   00 00 00 07   00 00 00 00   00 00 00 04
          meaning:        n = 2    txtlen = 7    off[0] = 0    off[1] = 4

    byte position:  16 17 18 19   20 21 22 23   24 25 26 27 28 29 30
         contents:  00 00 00 01   00 00 00 02    y  e  s 00  n  o 00
          meaning:   val[0] = 1    val[1] = 2    txt --->
```

This table could just as well be recorded as

```
    byte position:  00 01 02 03   04 05 06 07   08 09 10 11   12 13 14 15
         contents:  00 00 00 02   00 00 00 07   00 00 00 03   00 00 00 00
          meaning:        n = 2    txtlen = 7    off[0] = 3    off[1] = 0

    byte position:  16 17 18 19   20 21 22 23   24 25 26 27 28 29 30
         contents:  00 00 00 01   00 00 00 02    n  o 00  y  e  s 00
          meaning:   val[0] = 1    val[1] = 2    txt --->
```

but it could not be recorded as

```
    byte position:  00 01 02 03   04 05 06 07   08 09 10 11   12 13 14 15
         contents:  00 00 00 02   00 00 00 07   00 00 00 04   00 00 00 00
          meaning:        n = 2    txtlen = 7    off[0] = 4    off[1] = 0

    byte position:  16 17 18 19   20 21 22 23   24 25 26 27 28 29 30
         contents:  00 00 00 02   00 00 00 01    y  e  s 00  n  o 00
          meaning:   val[0] = 2    val[1] = 1    txt --->
```

It is not the out-of-order values of **off[]** that cause problems; it is out-of-order values of **val[]**. In terms of table construction, we find it easier to keep the table sorted as it grows. This way one can use a binary search routine to find the appropriate position in **val[]** quickly.

The following routine will find the appropriate slot. It uses the manifests we previously defined and thus it assumes the table is in compressed form, but that is not important. Changing the definitions of the manifests to point to separate areas would be easy enough.

```
    /*
        slot = vlfindval(char *baseptr, int val)

        Looks for value val in label at baseptr.
            If found:
                    returns slot number:  0, 1, 2, ...
            If not found:
                    returns k<0 such that val would go in slot -(k+1)
                            k== -1        would go in slot 0.
                            k== -2        would go in slot 1.
                            k== -3        would go in slot 2.
    */

    int vlfindval(char *baseptr, int myval)
```

```
        {
                int     n ;
                int     lb, ub, try ;
                int     *val ;
                char    *txt ;
                int     *off ;
                int     curval ;

                n   = *Ptr_n(baseptr) ;
                val =  Ptr_val(baseptr, n) ;

                if (n==0) return(-1) ;  /* not found, insert into 0 */

                                        /* in what follows,                */
                                        /* we know result between [lb,ub   */
                                        /* or it is not in the table       */
                lb = 0 ;
                ub = n - 1 ;
                while (1) {
                        try = (lb + ub) / 2 ;
                        curval = val[try] ;
                        if (myval == curval) return(try) ;
                        if (myval<curval) {
                                ub = try - 1 ;
                                if (ub<lb) return(-(try+1)) ;
                                /* because want to insert before try, ergo,
                                want to return try, and transform is -(W+1). */
                        }
                        else /* myval>curval */ {
                                lb = try + 1 ;
                                if (ub<lb) return(-(lb+1)) ;
                                /* because want to insert after try, ergo,
                                want to return try+1 and transform is -(W+1) */
                        }
                }
                /*NOTREACHED*/
        }
```

For earlier documentation, see dta_113.

Also see

Manual: **[P] file formats .dta**

Online: [D] save, **[D] use**; **[D] sysuse**, **[D] webuse**; **[U] 1.2.1 Sample datasets**