

Technical — Details for programmers

[Description](#)[Remarks and examples](#)[Also see](#)

Description

Technical information for programmers who wish to extend `mi` is provided below.

Remarks and examples

stata.com

Remarks are presented under the following headings:

*Notation**Definition of styles**Style all**Style wide**Style mlong**Style flong**Style flongsep**Style flongsep_sub**Adding new commands to mi**Outline for new commands**Utility routines**u_mi_assert_set**u_mi_certify_data**u_mi_no_sys_vars and u_mi_no_wide_vars**u_mi_zap_chars**u_mi_xeq_on_tmp_flongsep**u_mi_get_flongsep_tmpname**mata: u_mi_flongsep_erase()**u_mi_sortback**u_mi_save and u_mi_use**mata: u_mi_wide_swapvars()**u_mi_fixchars**mata: u_mi_cpchars_get() and mata: u_mi_cpchars_put()**mata: u_mi_get_mata_instanced_var()**mata: u_mi_ptrace_***How to write other set commands to work with mi*

Notation

 M = # of imputations m = imputation number

0. original data with missing values

1. first imputation dataset

.

.

 M . last imputation dataset N = number of observations in $m = 0$

Definition of styles

Style describes how the `mi` data are stored. There are four styles: `wide`, `mlong`, `flong`, and `flongsep`.

Style all

Characteristics:

`_dta[_mi_marker]` “`_mi_ds_1`”

Description: `_dta[_mi_marker]` is set with all styles, including `flongsep_sub`. The definitions below apply only if “`'_dta[_mi_marker]' = "_mi_ds_1"`”.

Style wide

Characteristics:

`_dta[_mi_style]` “`wide`”
`_dta[_mi_M]` M
`_dta[_mi_ivars]` imputed variables; variable list
`_dta[_mi_pvars]` passive variables; variable list
`_dta[_mi_rvars]` regular variables; variable list
`_dta[_mi_update]` time last updated; $\%tc_value/1000$

Variables:

`_mi_miss` whether incomplete; 0 or 1
`_#_varname` $varname$ for $m = \#$, defined for each
 “`'_dta[_mi_ivars]'`” and “`'_dta[_mi_pvars]'`”

Description: $m = 0, m = 1, \dots, m = M$ are stored in one dataset with $_N = N$ observations. Each imputed and passive variable has M additional variables associated with it. If variable `bp` contains the values in $m = 0$, then values for $m = 1$ are contained in variable `_1_bp`, values for $m = 2$ in `_2_bp`, and so on. `wide` stands for *wide*.

Style mlong

Characteristics:

`_dta[_mi_style]` “`mlong`”
`_dta[_mi_M]` M
`_dta[_mi_N]` N
`_dta[_mi_n]` # of observations in marginal
`_dta[_mi_ivars]` imputed variables; variable list
`_dta[_mi_pvars]` passive variables; variable list
`_dta[_mi_rvars]` regular variables; variable list
`_dta[_mi_update]` time last updated; $\%tc_value/1000$

Variables:

`_mi_m` $m; 0, 1, \dots, M$
`_mi_id` ID; $1, \dots, N$
`_mi_miss` whether incomplete; 0 or 1 if `_mi_m = 0`, else .

Description: $m = 0, m = 1, \dots, m = M$ are stored in one dataset with $_N = N + M \times n$ observations, where n is the number of incomplete observations in $m = 0$. `mlong` stands for *marginal long*.

Style flong

Characteristics:

<code>_dta[_mi_style]</code>	“flong”
<code>_dta[_mi_M]</code>	M
<code>_dta[_mi_N]</code>	N
<code>_dta[_mi_ivars]</code>	imputed variables; variable list
<code>_dta[_mi_pvars]</code>	passive variables; variable list
<code>_dta[_mi_rvars]</code>	regular variables; variable list
<code>_dta[_mi_update]</code>	time last updated; $\%tc_value/1000$

Variables:

<code>_mi_m</code>	$m; 0, 1, \dots, M$
<code>_mi_id</code>	ID; $1, \dots, N$
<code>_mi_miss</code>	whether incomplete; 0 or 1 if <code>_mi_m = 0</code> , else .

Description: $m = 0, m = 1, \dots, m = M$ are stored in one dataset with $_N = N + M \times N$ observations, where N is the number of observations in $m = 0$. `flong` stands for *full long*.

Style flongsep

Characteristics:

<code>_dta[_mi_style]</code>	“flongsep”
<code>_dta[_mi_name]</code>	<i>name</i>
<code>_dta[_mi_M]</code>	M
<code>_dta[_mi_N]</code>	N
<code>_dta[_mi_ivars]</code>	imputed variables; variable list
<code>_dta[_mi_pvars]</code>	passive variables; variable list
<code>_dta[_mi_rvars]</code>	regular variables; variable list
<code>_dta[_mi_update]</code>	time last updated; $\%tc_value/1000$

Variables:

<code>_mi_id</code>	ID; $1, \dots, N$
<code>_mi_miss</code>	whether incomplete; 0 or 1

Description: $m = 0, m = 1, \dots, m = M$ are each separate `.dta` datasets. If $m = 0$ data are stored in `pat.dta`, then $m = 1$ data are stored in `_1_pat.dta`, $m = 2$ in `_2_pat.dta`, and so on.

The definitions above apply only to $m = 0$, the dataset named ‘`_dta[_mi_name]`’.`dta`. See *Style flongsep_sub* directly below for $m > 0$. `flongsep` stands for *full long and separate*.

Style flongsep_sub

Characteristics:

<code>_dta[_mi_style]</code>	“flongsep_sub”
<code>_dta[_mi_name]</code>	<i>name</i>
<code>_dta[_mi_m]</code>	$m; 0, 1, \dots, M$

Variables:

<code>_mi_id</code>	ID; $1, \dots, N$
---------------------	-------------------

Description: The description above applies to the `_‘_dta[_mi_m]’_‘_dta[_mi_name]’_dta` datasets. There are M such datasets recording $m = 1, \dots, M$ used by the `flongsep` style directly above.

Adding new commands to mi

New commands are written in ado. Name the new command `mi_cmd_newcmd` and store it in `mi_cmd_newcmd.ado`. When the user types `mi newcmd ...`, `mi_cmd_newcmd.ado` will be executed.

See *Writing programs for use with mi* of [P] **program properties** for details on how to write estimation commands for use with the `mi estimate` prefix.

Outline for new commands

```

program mi_cmd_newcmd, rclass                                (1)
    version 17.0
    u_mi_assert_set                                         (2)
    syntax ... [, ... noUPdate ...]                         (3)
    ...
    u_mi_certify_data, acceptable                           (4)
    ...
    if ("update"=="") {
        u_mi_certify_data, proper                           (5)
    }
    ...
end

```

Notes:

1. The command may be `rclass`; that is not required. It may be `eclass` instead if you wish.
2. `u_mi_assert_set` verifies that the data are `mi` data; see [u_mi_assert_set](#) below.
3. If you intend for your command to use `mi update` to update the data before performing its intended task, include a `noupdate` option; see [MI] **noupdate option**. Some commands instead or in addition run `mi update` to perform cleanup after performing their task. Such use does not require a `noupdate` option.
4. `u_mi_certify_data` is the internal routine that performs `mi update`. An update is divided into two parts, called `acceptable` and `proper`. All commands should verify that the data are `acceptable`; see [u_mi_certify_data](#) below.
5. `u_mi_certify_data, proper` performs the second step of `mi update`; it verifies that `acceptable` data are `proper`. Whether you verify properness is up to you, but if you do, you are supposed to include a `noupdate` option to skip running the check.

Utility routines

The only information you absolutely need to know is that already revealed. Using the utility routines described below, however, will simplify your programming task and make your code appear more professional to the end user.

As you read what follows, remember that you may review the source code for the routines by using `viewsource`; see [P] **viewsource**. If you wanted to see the source for `u_mi_assert_set`, you would type `viewsource u_mi_assert_set.ado`. If you do this, you will sometimes see that the routines allow options not documented below. Ignore those options; they may not appear in future releases.

Using `viewsource`, you may also review examples of the utility commands being used by viewing the source of the `mi` commands we have written. Each `mi` command appears in the file `mi_cmd_command.ado`. Also remember that other `mi` commands make useful utility routines. For instance, if your new command makes passive variables, use `mi register` to register them. Always call existing `mi` commands through `mi`; code `mi passive` and not `mi_cmd_passive`.

u_mi_assert_set

```
u_mi_assert_set [ desired_style ]
```

This utility verifies that data are `mi` and optionally of the desired style; it issues the appropriate error message and stops execution if not. The optional argument *desired_style* can be `wide`, `mlong`, `flong`, or `flongsep`, but is seldom specified. When not specified, any style is allowed.

u_mi_certify_data

```
u_mi_certify_data [ , acceptable proper noupdate sortok ]
```

This command performs `mi update`. `mi update` is equivalent to `u_mi_certify_data, acceptable proper sortok`.

Specify one or both of `acceptable` and `proper`. If the `noupdate` option is specified, then `proper` is specified. The `sortok` option specifies that `u_mi_certify_data` need not spend extra time to preserve and restore the original sort order of the data.

An update is divided into two parts. In the first part, called `acceptable`, $m = 0$ and the `_dta[_mi_*]` characteristics are certified. Your program will use the information recorded in those characteristics, and before that information can be trusted, the data must be certified as `acceptable`. Do not trust any `_dta[_mi_*]` characteristics until you have run `u_mi_certify_data, acceptable`.

`u_mi_certify_data, proper` verifies that data known to be `acceptable` are `proper`. In practice, this means that in addition to trusting $m = 0$, you can trust $m > 0$.

Running `u_mi_certify_data, acceptable` might actually result in the data being certified as `proper`, although you cannot depend on that. When you run `u_mi_certify_data, acceptable` and certain problems are observed in $m = 0$, they are fixed in all m , which can lead to other problems being detected, and by the time the whole process is through, the data are `proper`.

u_mi_no_sys_vars and u_mi_no_wide_vars

```
u_mi_no_sys_vars "variable_list" [ "word" ]
```

```
u_mi_no_wide_vars "variable_list" [ "word" ]
```

These routines are for use in parsing user input.

`u_mi_no_sys_vars` verifies that the specified list of variable names does not include any `mi` system variables such as `_mi_m`, `_mi_id`, `_mi_miss`, etc.

`u_mi_no_wide_vars` verifies that the specified list of variable names does not include any style wide $m > 0$ variables of the form `_#_varname`. `u_mi_no_wide_vars` may be called with any style of data but does nothing if the style is not wide.

Both functions issue appropriate error messages if problems are found. If *word* is specified, the error message will be “*word* may not include ...”. Otherwise, the error message is “may not specify ...”.

u_mi_zap_chars

```
u_mi_zap_chars
```

`u_mi_zap_chars` deletes all `_dta[_mi_*]` characteristics from the data in memory.

u_mi_xeq_on_tmp_flongsep

```
u_mi_xeq_on_tmp_flongsep [ , nopreserve ]: command
```

`u_mi_xeq_on_tmp_flongsep` executes *command* on the data in memory, said data converted to style `flongsep`, and then converts the `flongsep` result back to the original style. If the data already are `flongsep`, a temporary copy is made and, at the end, posted back to the original. Either way, *command* is run on a temporary copy of the data. If anything goes wrong, the user's original data are restored; that is, they are restored unless `nopreserve` is specified. If *command* completes without error, the `flongsep` data in memory are converted back to the original style and the original data are discarded.

It is not uncommon to write commands that can deal only with `flongsep` data, and yet these seem to users as if they work with all styles. That is because the routines use `u_mi_xeq_on_tmp_flongsep`. They start by allowing any style, but the guts of the routine are written assuming `flongsep`. `mi_stjoin` is implemented in this way. There are two parts to `mi_stjoin`: `mi_cmd_stjoin.ado` and `mi_sub_stjoin_flongsep.ado`. `mi_cmd_stjoin.ado` ends with

```
u_mi_xeq_on_tmp_flongsep: mi_sub_stjoin_flongsep 'if', 'options'
```

`mi_sub_stjoin_flongsep` does all the work, while `u_mi_xeq_on_tmp_flongsep` handles the issue of converting to `flongsep` and back again. The `mi_sub_stjoin_flongsep` subroutine must appear in its own ado-file because `u_mi_xeq_on_tmp_flongsep` is itself implemented as an ado-file. `u_mi_xeq_on_tmp_flongsep` would be unable to find the subroutine otherwise.

u_mi_get_flongsep_tmpname

```
u_mi_get_flongsep_tmpname macname : basename
```

`u_mi_get_flongsep_tmpname` creates a temporary `flongsep` name based on *basename* and stores it in the local macro *macname*. `u_mi_xeq_on_tmp_flongsep`, for your information, obtains the temporary name it uses from this routine.

`u_mi_get_flongsep_tmpname` is seldom used directly because `u_mi_xeq_on_tmp_flongsep` works well for shifting temporarily into `flongsep` mode, and `u_mi_xeq_on_tmp_flongsep` does a lot more than just getting a name under which the data should be temporarily stored. There are instances, however, when one needs to be more involved in the conversion. For examples, see the source `mi_cmd_append.ado` and `mi_cmd_merge.ado`. The issue these two routines face is that they need to shift two input datasets to `flongsep`, then they create a third from them, and that is the only one that needs to be shifted back to the original style. So these two commands handle the conversions themselves using `u_mi_get_flongsep_tmpname` and `mi convert` (see [MI] [mi convert](#)).

For instance, they start with something like

```
u_mi_get_flongsep_tmpname master : __mimaster
```

That creates a temporary name suitable for use with `mi convert` and stores it in 'master'. The suggested name is `__mimaster`, but if that name is in use, then `u_mi_get_flongsep_tmpname` will form from it `__mimaster1`, or `__mimaster2`, etc. We recommend that you specify a *basename* that begins with `__mi`, which is to say, two underscores followed by `mi`.

Next you must appreciate that it is your responsibility to eliminate the temporary files. You do that by coding something like

```

...
local origstyle "'_dta[_mi_style]'"
if ("'origstyle'"=="flongsep") {
    local origstyle "'origstyle' '_dta[_mi_name]'"
}
u_mi_get_flongsep_tmpname master : __mimaster
capture {
    quietly mi convert flongsep 'master'
    ...
    quietly mi convert 'origstyle', clear replace
}
nobreak {
    local rc = _rc
    mata: u_mi_flongsep_erase("'master'", 0, 0)
    if ('rc') {
        exit 'rc'
    }
}

```

The other thing to note above is our use of `mi convert 'master'` to convert our data to flongsep under the name `'master'`. What, you might wonder, happens if our data already is flongsep? A nice feature of `mi convert` is that when run on data that are already flongsep, it performs an `mi copy`; see [MI] [mi copy](#).

mata: u_mi_flongsep_erase()

```
mata: u_mi_flongsep_erase("name", from [, output])
```

where

<i>name</i>	<i>string</i> ; flongsep name
<i>from</i>	<i>#</i> ; where to begin erasing
<i>output</i>	0 1; whether to produce output

`mata: u_mi_flongsep_erase()` is the internal version of `mi erase` (see [MI] [mi erase](#)); use whichever is more convenient.

Input *from* is usually specified as 0 and then `mata: u_mi_flongsep_erase()` erases *name*.*dta*, *_1_name*.*dta*, *_2_name*.*dta*, and so on. *from* may be specified as a number greater than zero, however, and then erased are *_*from*_name*.*dta*, *_*from*+1>_name*.*dta*, *_*from*+2>_name*.*dta*,

If *output* is 0, no output is produced; otherwise, the erased files are also listed. If *output* is not specified, files are listed.

See `viewsource u_mi.mata` for the source code for this routine.

u_mi_sortback

```
u_mi_sortback varlist
```

`u_mi_sortback` removes dropped variables from *varlist* and sorts the data on the remaining variables. The routine is for dealing with sort-preserve problems when `program name`, `sortpreserve` is not adequate, such as when the data might be subjected to substantial editing between the preserving of the sort order and the restoring of it. To use `u_mi_sortback`, first record the order of the data:

```

local sortedby : sortedby
tempvar recnum
gen long `recnum' = _n
quietly compress `recnum'

```

Later, when you want to restore the sort order, you code

```
u_mi_sortback `sortedby' `recnum'
```

u_mi_save and u_mi_use

```
u_mi_save macname : filename [, save_options]
```

```
u_mi_use "'macname'" filename [, clear nolabel]
```

save_options are as described in [D] [save](#). `clear` and `no`label are as described in [D] [use](#). In both commands, *filename* must be specified in quotes if it contains any special characters or blanks.

It is sometimes necessary to save data in a temporary file and reload them later. In such cases, when the data are reloaded, you would like to have the original `c(filename)`, `c(filedate)`, and `c(changed)` restored. `u_mi_save` saves that information in *macname*. `u_mi_use` restores the information from the information saved in *macname*. Note the use of compound quotes around '*macname*' in `u_mi_use`; they are not optional.

mata: u_mi_wide_swapvars()

```
mata: u_mi_wide_swapvars(m, tmpvarname)
```

where

<i>m</i>	#; $1 \leq \# \leq M$
<i>tmpvarname</i>	string; name from <code>tempvar</code>

This utility is for use with wide data only. For each variable name contained in `_dta[_mi_ivars]` and `_dta[_mi_pvars]`, `mata: u_mi_wide_swapvars()` swaps the contents of *varname* with *m_varname*. Argument *tmpvarname* must be the name of a temporary variable obtained from command `tempvar`, and the variable must not exist. `mata: u_mi_wide_swapvars()` will use this variable while swapping. See [P] [macro](#) for more information on `tempvar`.

This function is its own inverse, assuming `_dta[_mi_ivars]` and `_dta[_mi_pvars]` have not changed.

See `viewsource u_mi.mata` for the source code for this routine.

u_mi_fixchars

```
u_mi_fixchars [, acceptable proper]
```

`u_mi_fixchars` makes the data and variable characteristics the same in $m = 1, m = 2, \dots, m = M$ as they are in $m = 0$. The options specify what is already known to be true about the data, that the data are known to be acceptable or known to be proper. If neither is specified, you are stating that you do not know whether the data are even acceptable. That is okay. `u_mi_fixchars` handles performing whatever certification is required. Specifying the options makes `u_mi_fixchars` run faster.

This stabilizing of the characteristics is not about `mi`'s characteristics; that is handled by `u_mi_certify_data`. Other commands of Stata set and use characteristics, while `u_mi_fixchars` ensures that those characteristics are the same across all `m`.

mata: u_mi_cpchars_get() and **mata: u_mi_cpchars_put()**

```
mata: u_mi_cpchars_get(matavar)
mata: u_mi_cpchars_put(matavar, {0|1|2})
```

where *matavar* is a Mata [transmorphic](#) variable. Obtain *matavar* from `u_mi_get_mata_instanced_var()` when using these functions from Stata.

These routines replace the characteristics in one dataset with those of another. They are used to implement `u_mi_fixchars`.

`mata: u_mi_cpchars_get(matavar)` stores in *matavar* the characteristics of the data in memory. The data in memory remain unchanged.

`mata: u_mi_cpchars_put(matavar, #)` replaces the characteristics of the data in memory with those previously recorded in *matavar*. The second argument specifies the treatment of `_dta[_mi_*]` characteristics:

- | | |
|---|--|
| 0 | delete them in the destination data |
| 1 | copy them from the source just like any other characteristic |
| 2 | retain them as-is from the destination data. |

mata: u_mi_get_mata_instanced_var()

```
mata: u_mi_get_mata_instanced_var("macname", "basename" [, i_value])
```

where

<i>macname</i>	name of local macro
<i>basename</i>	suggested name for instanced variable
<i>i_value</i>	initial value for instanced variable

`mata: u_mi_get_mata_instanced_var()` creates a new Mata global variable, initializes it with *i_value* or as a 0×0 real, and places its name in local macro *macname*. Typical usage is

```
local var
capture noisily {
    mata: u_mi_get_mata_instanced_var("var", "myvar")
    ...
    ... use 'var' however you wish ...
    ...
}
nobreak {
    local rc = _rc
    capture mata: mata drop 'var'
    if ('rc') {
        exit 'rc'
    }
}
```

mata: u_mi_ptrace_*()

```
h = u_mi_ptrace_open("filename", {"r"|"w"} [, {0|1}])
```

```
u_mi_ptrace_write_stripes(h, id, ynames, xnames)
```

```
u_mi_ptrace_write_iter(h, m, iter, B, V)
```

```
u_mi_ptrace_close(h)
```

```
u_mi_ptrace_safeclose(h)
```

The above are Mata functions, where

h, if it is declared, should be declared transmorphic

id is a string scalar

ynames and *xnames* are string scalars

m and *iter* are real scalars

B and *V* are real matrices; *V* must be symmetric

These routines write parameter-trace files; see [MI] **mi ptrace**. The procedure is 1) open the file; 2) write the stripes; 3) repeatedly write iteration information; and 4) close the file.

1. Open the file: *filename* may be specified with or without a file suffix. Specify the second argument as "w". The third argument should be 1 if the file may be replaced when it exists, and 0 otherwise.
2. Write the stripes: Specify *id* as the name of your routine or as ""; `mi ptrace describe` will show this string as the creator of the file if the string is not "". *ynames* and *xnames* are both string scalars containing space-separated names or, possibly, *op.names*.
3. Repeatedly write iteration information: Written are *m*, the imputation number; *iter*, the iteration number; *B*, the matrix of coefficients; and *V*, the variance matrix. *B* must be $ny \times nx$ and *V* must be $ny \times ny$ and symmetric, where $nx = \text{length}(\text{tokens}(xnames))$ and $ny = \text{length}(\text{tokens}(ynames))$.
4. Close the file: In Mata, use `u_mi_ptrace_close(h)`. It is highly recommended that, before step 1, *h* be obtained from inside Stata (not Mata) using `mata: u_mi_get_mata_instanced_var("h", "myvar")`. If you follow this advice, include a `mata: u_mi_ptrace_safeclose('h')` in the ado-file cleanup code. This will ensure that open files are closed if the user presses *Break* or something else causes your routine to exit before the file is closed. A correctly written program will have two closes, one in Mata and another in the ado-file, although you could omit the one in Mata. See [mata: u_mi_get_mata_instanced_var\(\)](#) directly above.

Also included in `u_mi_ptrace_*`() are routines to read parameter-trace files. You should not need these routines because users will use Stata command `mi ptrace use` to load the file you have written. If you are interested, however, then type `viewsource u_mi_ptrace.mata`.

How to write other set commands to work with mi

This section concerns the writing of other set commands such as [ST] `stset` or [XT] `xtset`—set commands having nothing to do with `mi`—so that they properly work with `mi`.

The definition of a set command is any command that creates characteristics in the data, and possibly creates variables in the data, that other commands in the suite will subsequently access. Making such set commands work with `mi` is mostly `mi`'s responsibility, but there is a little you need to do to assist `mi`. Before dealing with that, however, write and debug your set command ignoring `mi`. Once that is done, go back and add a few lines to your code. We will pretend your set command is named `mynewset` and your original code looks something like this:

```
program mynewset
  ...
  syntax ... [, ... ]
  ...
end
```

Our goal is to make it so that `mynewset` will not run on `mi` data while simultaneously making it so that `mi` can call it (the user types `mi mynewset`). When the user types `mi mynewset`, `mi` will 1) give `mynewset` a clean, $m = 0$ dataset on which it can run and 2) duplicate whatever `mynewset` does to $m = 0$ on $m = 1, m = 2, \dots, m = M$.

To achieve this, modify your code to look like this:

```
program mynewset
  ...
  syntax ... [, ... MI] (1)
  if ("mi'"=="") { (2)
    u_mi_not_mi_set "mynewset"
    local checkvars "*" (3)
  }
  else {
    local checkvars "u_mi_check_setvars settime" (3)
  }
  ...
  'checkvars' 'varlist' (4)
  ...
end
```

That is,

1. Add the `mi` option to any options you already have.
2. If the `mi` option is not specified, execute `u_mi_not_mi_set`, passing to it the name of your set command. If the data are not `mi`, then `u_mi_not_mi_set` will do nothing. If the data are `mi`, then `u_mi_not_mi_set` will issue an error telling the user to run `mi mynewset`.
3. Set new local macro `checkvars` to `*` if the `mi` option is not specified, and otherwise to `u_mi_check_setvars`. We should mention that the `mi` option will be specified when `mi mynewset` calls `mynewset`.
4. Run `'checkvars'` on any input variables `mynewset` uses that must not vary across m . `mi` does not care about other variables or even about new variables `mynewset` might create; it cares only about existing variables that should not vary across m .

Let's understand what `'checkvars' varlist` does. If the `mi` option was not specified, the line expands to `"* varlist"`, which is a comment, and does nothing. If the `mi` option was specified, the line expands to `"u_mi_check_setvars settime varlist"`. We are calling `mi` routine `u_mi_check_setvars`, telling it that we are calling at set time, and passing along `varlist`. `u_mi_check_setvars` will verify that `varlist` does not contain `mi` system variables

or variables that vary across m . Within `mynewset`, you may call ‘`checkvars`’ repeatedly if that is convenient.

You have completed the changes to `mynewset`. You finally need to write one short program that reads

```
program mi_cmd_mynewset
  version 17.0
  mi_cmd_genericset "mynewset '0'" "_mynewset_x _mynewset_y"
end
```

In the above, we assume that `mynewset` might add one or two variables to the data named `_mynewset_x` and `_mynewset_y`. List in the second argument all variables `mynewset` might create. If `mynewset` never creates new variables, then the program should read

```
program mi_cmd_mynewset
  version 17.0
  mi_cmd_genericset "mynewset '0'"
end
```

You are done.

Also see

[MI] [Intro](#) — Introduction to mi