Title

stata.com

setbreakintr( ) — Break-key processing

Description

`setbreakintr(`*val*`)` turns the break-key interrupt off (*val*`==0`) or on (*val*`!=0`) and returns the value of the previous break-key mode, 1, it was on, or 0, it was off.

`querybreakintr()` returns 1 if the break-key interrupt is on and 0 otherwise.

`breakkey()` (for use in `setbreakintr(0)` mode) returns 1 if the break key has been pressed since it was last reset.

`breakkeyreset()` (for use in `setbreakintr(0)` mode) resets the break key.

Syntax

real scalar `setbreakintr(`*real scalar val*`)`

real scalar `querybreakintr()`

real scalar `breakkey()`

void `breakkeyreset()`

Remarks and examples

stata.com

Remarks are presented under the following headings:

Default break-key processing

By default, if the user presses *Break*, Mata stops execution and returns control to the console, setting the return code to 1.

To obtain this behavior, there is nothing you need do. You do not need to use these functions.

Suspending the break-key interrupt

The default behavior is known as interrupt-on-break and is also known as `setbreakintr(1)` mode.

The alternative is break-key suspension, also known as `setbreakintr(0)` mode.

For instance, you have several steps that must be performed in their entirety or not at all. The way to do this is

```
val = setbreakintr(0)
...
... (critical code) ...
...
(void) setbreakintr(val)
```

The first line stores in *val* the current break-key processing mode and then sets the mode to break-key suspension. The critical code then runs. If the user presses *Break* during the execution of the critical code, that will be ignored. Finally, the code restores the previous break-key processing mode.

Break-key polling

In coding large, interactive systems, you may wish to adopt the break-key polling style of coding rather than interrupt-on-break. In this alternative style of coding, you turn off interrupt-on-break:

```
val = setbreakintr(0)
```

and, from then on in your code, wherever you are willing to interrupt your code, you ask (poll whether) the break key has been pressed:

```
...
if (breakkey()) {
      ...
}
...
```

In this style of coding, you must decide where and when you are going to reset the break key, because once the break key has been pressed, `breakkey()` will continue to return 1 every time it is called. To reset the break key, code,

```
breakkeyreset()
```

You can also adopt a mixed style of coding, using interrupt-on-break in some places and polling in others. Function `querybreakintr()` can then be used to determine the current mode.

Conformability

`setbreakintr(`*val*`)`:
val: 1×1
result: 1×1

`querybreakintr()`, `breakkey()`:
result: 1×1

`breakkeyreset()`:
result: *void*

Diagnostics

`setbreakintr(1)` aborts with break if the break key has been pressed since the last `setbreakintr(0)` or `breakkeyreset()`. Code `breakkeyreset()` before `setbreakintr(1)` if you do not want this behavior.

After coding `setbreakintr(1)`, remember to restore `setbreakintr(0)` mode. It is not, however, necessary, to restore the original mode if `exit()` or `_error()` is about to be executed.

`breakkey()`, once the break key has been pressed, continues to return 1 until `breakkeyreset()` is executed.

There is absolutely no reason to use `breakkey()` in `setbreakintr(0)` mode, because the only value it could return is 0.

Also see

[M-5] **error( )** — Issue error message

[M-4] **Programming** — Programming functions