Title

stata.com

dialog programming — Dialog programming

Description Remarks and examples Also see

Description

Dialog-box programs—also called dialog resource files—allow you to define the appearance of a dialog box, specify how its controls work when the user fills it in (such as hiding or disabling specific controls), and specify the ultimate action to be taken (such as running a Stata command) when the user clicks on **OK** or **Submit**.

Remarks and examples

stata.com

Remarks are presented under the following headings:

1. Introduction

At a programming level, the purpose of a dialog box is to produce a Stata command to be executed. Along the way, it hopefully provides the user with an intuitive and consistent experience—that is your job as a dialog-box programmer—but the ultimate output will be

```
list mpg weight or
regress mpg weight if foreign or
append using myfile
```

or whatever other Stata command is appropriate. Dialog boxes are limited to executing one Stata command, but that does not limit what you can do with them because that Stata command can be an ado-file. (Actually, there is another way around the one-command limit, which we will discuss in *5.1.3 rstrings: cmdstring and optstring*.)

This ultimate result is called the dialog box's u-action.

The u-action of the dialog box is determined by the code you write, called dialog code, which you store in a `.dlg` file. The name of the `.dlg` file is important because it determines the name of the dialog box. When a user types

```
. db regress
```

`regress.dlg` is executed. Stata finds the file the same way it finds ado-files—by looking along the ado-path; see [P] **sysdir**. `regress.dlg` runs regress commands because of the dialog code that appears inside the `regress.dlg` file. `regress.dlg` could just as well execute `probit` commands or even `merge` commands if the code were written differently.

`.dlg` files describe

1. how the dialogs look;
2. how the input controls of the dialogs interact with each other; and
3. how the u-action is constructed from the user's input.

Items 1 and 2 determine how intuitive and consistent the user finds the dialog. Item 3 determines what the dialog box does. Item 2 determines whether some fields are disabled or hidden so that they cannot be mistakenly filled in until the user clicks on something, checks something, or fills in a certain result.

2. Concepts

A dialog box is composed of many elements called controls, including static text, edit fields, and checkboxes. Input controls are those that the user fills in, such as checkboxes and text-entry fields. Static controls are fixed text and lines that appear on the dialog box but that the user cannot change. See *Appendix A* below for definitions of the various types of controls as well as other related jargon.

In the jargon we use, a dialog box is composed of dialogs, and dialogs are composed of controls. When a dialog box contains multiple dialogs, only one dialog is shown at a time. Here access to the dialogs is made possible through small tabs. Clicking on the tab associated with a dialog makes that dialog active.

The dialog box may contain the helper buttons **Help** (shown as a small button with a question mark on it) and **Reset** (shown as a small button with an `R` on it). These buttons appear in the dialog box—not the individual dialogs—so in a multiple-dialog dialog box, they appear regardless of the dialog (tab) selected.

The **Help** helper button displays a help file associated with the dialog box.

The **Reset** helper button resets the dialog box to its initial state. Each time a user invokes a particular dialog box, it will remember the values last set for its controls. The reset button allows the user to restore the default values for all controls in the dialog box.

The dialog box may also include the u-action buttons **OK**, **Submit**, **Copy**, and **Cancel**. Like the helper buttons, u-action buttons appear in the dialog box—not the individual dialogs—so in a multiple-dialog dialog box, they appear regardless of the dialog (tab) selected.

The **OK** u-action button constructs the u-action, sends it to Stata for execution, and closes the dialog box.

The **Submit** u-action button constructs the u-action, sends it to Stata for execution, and leaves the dialog box open.

The **Copy** u-action button constructs the u-action, sends it to the clipboard, and leaves the dialog box open.

The **Cancel** u-action button closes the dialog box without constructing the u-action.

A dialog box does not have to include all of these u-action buttons, but it needs at least one.

Thus the nesting is

Dialog box, which contains
  Dialog 1, which contains
    input controls and static controls
  Dialog 2, which is optional and which, if defined, contains
    input controls and static controls
  [. . .]
  Helper buttons, which are optional and which, if defined, contain
    [**Help** button]
    [**Reset** button]
  U-action buttons, which contain
    [**OK** button]
    [**Submit** button]
    [**Copy** button]
    [**Cancel** button]

Said differently,

1. a dialog box must have at least one dialog, must have one set of u-action buttons, and may have helper buttons;
2. a dialog must have at least one control and may have many controls; and
3. the u-action buttons may include any of **OK**, **Submit**, **Copy**, and **Cancel** and must include at least one of them.

Here is a simple `.dlg` file that will execute the `kappa` command, although it does not allow `if` *exp* and `in` *range*:

```
———————— BEGIN ———————— mykappa.dlg ————————————————————————————
// ----------------- set version number and define size of box ---------
VERSION 13
POSITION . . 290 200
// ------------------------------------------- define a dialog ---------
DIALOG main, label("kappa - Interrater agreement")
BEGIN
        TEXT    tx_var 10  10 270 ., label("frequency variables:")
        VARLIST vl_var  @ +20   @ ., label("frequencies")
END
// -------------------- define the u-action and helper buttons ---------
OK     ok1, label("OK")
CANCEL can1, label("Cancel")
SUBMIT sub1, label("Submit")
COPY   copy1,
HELP   hlp1, view("help kappa")
RESET  res1
// --------------------------- define how to assemble u-action ---------
PROGRAM command
BEGIN
        put "kappa "
        varlist main.vl_var
END
———————— END ———————— mykappa.dlg ————————————————————————————
```

2.1 Organization of the .dlg file

A .dlg file consists of seven parts, some of which are optional:

```
———————— BEGIN ———————— dialogboxname.dlg————————————————————————
VERSION 13                  Part 1:  version number
POSITION . . .              Part 2:  set size of dialog box
DEFINE . . .                Part 3, optional:  common definitions
LIST . . .
DIALOG . . .                Part 4: dialog definitions
    BEGIN
        FILE . . .              . . . which contain input controls
        BUTTON . . .
        CHECKBOX . . .
        COMBOBOX . . .
        EDIT . . .
        LISTBOX . . .
        RADIO . . .
        SPINNER . . .
        VARLIST . . .
        VARNAME . . .

        FRAME . . .             . . . and static controls
        GROUPBOX . . .
        TEXT . . .
    END
        repeat DIALOG. . . BEGIN. . . END as necessary
SCRIPT . . .                Part 5, optional:  i-action definitions
    BEGIN                       . . . usually done as scripts
        . . .
    END
PROGRAM . . .                   . . . but sometimes as programs
    BEGIN
        . . .
    END

OK . . .                    Part 6:  u-action and helper button definitions
CANCEL . . .
SUBMIT . . .
HELP . . .
RESET . . .

PROGRAM command             Part 7:  u-action definition
    BEGIN
        . . .
    END
———————— END ———————— dialogboxname.dlg————————————————————————
```

The VERSION statement must appear at the top; the other parts may appear in any order.

I-actions, mentioned in Part 5, are intermediate actions, such as hiding or showing, disabling or enabling a control, or opening the Viewer to display something, etc., while leaving the dialog up and waiting for the user to fill in more or press a u-action button.

2.2 Positions, sizes, and the DEFINE command

Part of specifying how a dialog appears is defining where things go and how big they are.

Positions are indicated by a pair of numbers, x and y. They are measured in pixels and are interpreted as being measured from the top-left corner: x is how far to the right, and y is how far down.

Sizes are similarly indicated by a pair of numbers, *xsize* and *ysize*. They, too, are measured in pixels and indicate the size starting at the top-left corner of the object.

Any command that needs a position or a size always takes all four numbers—position and size—and you must specify all four. In addition to each element being allowed to be a number, some extra codes are allowed. A position or size element is defined as

`#` any unsigned integer number, such as 0, 1, 10, 200, ...

`.` (period) meaning the context-specific default value for this position or size element. `.` is allowed only with heights of controls (heights are measured from the top down) and for the initial position of a dialog box.

`@` means the previous value for this position or size element. If `@` is used for an *x* or a *y*, then the *x* or *y* from the preceding command will be used. If `@` is used for an *xsize* or a *ysize*, then the previous *xsize* or *ysize* will be used.

`+#` means a positive offset from the last value (meaning to the right or down or bigger). If `+10` is used for *x*, the result will be 10 pixels to the right of the previous position. If `+10` is used for a *ysize*, it means 10 pixels taller.

`-#` means a negative offset from the last value (meaning to the left or up or smaller). If `-10` is used for *y*, the result will be 10 pixels above the previous position. If `-10` is used for a *xsize*, it means 10 pixels narrower.

name means the value last recorded for *name* by the `DEFINE` command.

The `DEFINE` command has the syntax

`DEFINE` *name* { `.` | `#` | `+#` | `-#` | `@x` | `@y` | `@xsize` | `@ysize` }

and may appear anywhere in your dialog code, even inside the `BEGIN/END` of `DIALOG`. Anywhere you need to specify a position or size element, you can use a *name* defined by `DEFINE`.

The first four possibilities for defining *name* have the obvious meaning: `.` means the default, `#` means the number specified, `+#` means a positive offset, and `-#` means a negative offset. The other four possibilities—`@x`, `@y`, `@xsize`, and `@ysize`—refer to the previous *x*, *y*, *xsize*, and *ysize* values, with "previous" meaning previous to the time the `DEFINE` command was issued.

2.3 Default values

You can also load input controls with initial, or default, values. For instance, perhaps, as a default, you want one checkbox checked and another unchecked, and you want an edit field filled in with "Default title".

The syntax of the `CHECKBOX` command, which creates checkboxes, is

`CHECKBOX ...` [`, ... default(`*defnumval*`) ...`]

In checkboxes, the `default()` option specifies how the box is to be filled in initially, and 1 corresponds to checked and 0 to unchecked.

The syntax of `EDIT`, which creates edit fields, is

`EDIT ...` [`, ... default(`*defstrval*`) ...`]

In edit fields, `default()` specifies what the box will contain initially.

Wherever *defnumval* appears in a syntax diagram, you may type

defnumval	Definition
#	meaning the number specified
`literal` #	same as #
`c(`*name*`)`	value of `c(`*name*`)`; see [P] **creturn**
`r(`*name*`)`	value of `r(`*name*`)`; see [P] **return**
`e(`*name*`)`	value of `e(`*name*`)`; see [P] **ereturn**
`s(`*name*`)`	value of `s(`*name*`)`; see [P] **return**
`global` *name*	value of global macro $*name*

Wherever *defstrval* appears in a syntax diagram, you may type

defstrval	Definition
string	meaning the string specified
`literal` *string*	same as *string*
`c(`*name*`)`	contents of `c(`*name*`)`; see [P] **creturn**
`r(`*name*`)`	contents of `r(`*name*`)`; see [P] **return**
`e(`*name*`)`	contents of `e(`*name*`)`; see [P] **ereturn**
`s(`*name*`)`	contents of `s(`*name*`)`; see [P] **return**
`char` *varname*`[`*charname*`]`	value of characteristic; see [P] **char**
`global` *name*	contents of global macro $*name*

Note: If *string* is enclosed in double quotes (simple or compound), the first set of quotes is stripped.

List and combo boxes present the user with a list of items from which to choose. In dialog-box jargon, rather than having initial or default values, the boxes are said to be populated. The syntax for creating a list-box input control is

`LISTBOX ... [ , ... contents(`*conspec*`) ... ]`

Wherever a *conspec* appears in a syntax diagram, you may type

`list` *listname*
populates the box with the specified list, which you create separately by using the `LIST` command. `LIST` has the following syntax:

```
LIST
    BEGIN
        item to appear
        item to appear
        ...
    END
```

`matrix`
populates the box with the names of all matrices currently defined in Stata.

`vector`
populates the box with the names of all $1 \times k$ and $k \times 1$ matrices currently defined in Stata.

`row`
populates the box with the names of all $1 \times k$ matrices currently defined in Stata.

column
populates the box with the names of all $k \times 1$ matrices currently defined in Stata.

square
populates the box with the names of all $k \times k$ matrices currently defined in Stata.

scalar
populates the box with the names of all scalars currently defined in Stata.

constraint
populates the box with the names of all constraints currently defined in Stata.

estimates
populates the box with the names of all saved estimates currently defined in Stata.

char *varname*[*charname*]
populates the box with the elements of the characteristic *varname*[*charname*], parsed on spaces.

e(*name*)
populates the box with the elements of e(name), parsed on spaces.

global
populates the box with the names of all global macros currently defined in Stata.

valuelabels
populates the box with the names of all values labels currently defined in Stata.

Predefined lists for use with Stata graphics:

Predefined lists	Definition
symbols	list of marker symbols
symbolsizes	list of marker symbol sizes
colors	list of colors
intensity	list of fill intensities
clockpos	list of clock positions
linepatterns	list of line patterns
linewidths	list of line widths
connecttypes	list of line connecting types
textsizes	list of text sizes
justification	list of horizontal text justifications
alignment	list of vertical text alignments
margin	list of margins
tickpos	list of axis-tick positions
angles	list of angles; usually used for axis labels
compass	list of compass directions
yesno	list containing Default, Yes, and No; usually accompanied by a user-defined *values* list

2.4 Memory (recollection)

All input control commands have a `default()` or `contents()` option that specifies how the control is to be filled in, for example,

`CHECKBOX ... [ , ... default(`*defnumval*`) ... ]`

In this command, if *defnumval* evaluates to 0, the checkbox is initially unchecked; otherwise, it is checked. If `default()` is not specified, the box is initially unchecked.

Dialogs remember how they were last filled in during a session, so the next time the user invokes the dialog box that contains this `CHECKBOX` command, the `default()` option will be ignored and the checkbox will be as the user last left it. That is, the setting will be remembered unless you specify the input control's `nomemory` option.

`CHECKBOX ... [ , ... default(`*defnumval*`) nomemory ... ]`

`nomemory` specifies that the dialog-box manager not remember between invocations how the control is filled in; it will always reset it to the default, whether that default is explicitly specified or implied.

Whether or not you specify `nomemory`, explicit or implicit defaults are also restored when the user presses the **Reset** helper button.

The contents of dialog boxes are only remembered during a session, not between them. Within a session, the `discard` command causes Stata to forget the contents of all dialog boxes.

The issues of initialization and memory are in fact more complicated than they first appear. Consider a list box. A list box might be populated with the currently saved estimates. If the dialog box containing this list box is closed and reopened, the available estimates may have changed. So list boxes are always repopulated according to the instructions given. Even so, list boxes remember the choice that was made. If that choice is still among the possibilities, that choice will be the one selected unless `nomemory` is specified; otherwise, the choice goes back to being the default—the first choice in the list of alternatives.

The same issues arise with combo boxes, and that is why some controls have the `default()` option and others have `contents()`. `default()` is used once, and after that, memory is substituted (unless `nomemory` is specified). `contents()` is always used—`nomemory` or not—but the choice made is remembered (unless `nomemory` is specified).

2.5 I-actions and member functions

I-actions—intermediate actions—refer to all actions taken in producing the u-action. An i-action might disable or hide controls when another control is checked or unchecked, although there are many other possibilities. I-actions are always optional.

I-actions are invoked by `on*()` options—those that begin with the letters "`on`". For instance, the syntax for the `CHECKBOX` command—the command for defining a checkbox control—is

`CHECKBOX` *controlname* `... [ , ... onclickon(`*iaction*`) onclickoff(`*iaction*`) ... ]`

`onclickon()` is the i-action to be taken when the checkbox is checked, and `onclickoff()` is the i-action for when the checkbox is unchecked. You do not have to fill in the `onclickon()` and `onclickoff()` options—the checkbox will work fine taking no i-actions—but you may fill them in if you want, say, to disable or to enable other controls when this control is checked. For instance, you might code

```
CHECKBOX sw2 ..., onclickon(d2.sw3.show) onclickoff(d2.sw3.hide) ...
```

`d2.sw3` refers to the control named `sw3` in the dialog `d2` (for instance, the control we just defined is named `sw2`). `hide` and `show` are called member functions. `hide` is the member function that hides a control, and `show` is its inverse. Controls have other member functions as well; what member functions are available is documented with the command that creates the specific control.

Many commands have `on*()` options that allow you to specify i-actions. When *iaction* appears in a syntax diagram, you can specify

`.` (period)
Do nothing; take no action. This is the default if you do not specify the `on*()` option.

`gaction` *dialogname*`.`*controlname*`.`*memberfunction* [*arguments*]
Execute the specified *memberfunction* on the specified control, where *memberfunction* may be

{ `hide` | `show` | `disable` | `enable` | `setposition` | *something_else* [*arguments*] }

All controls provide the *memberfunctions* `hide`, `show`, `disable`, `enable`, and `setposition`, and some controls make other, special *memberfunctions* available.

`hide` specifies that the control disappear from view (if it has not already done so). `show` specifies that it reappear (if it is not already visible).

`disable` specifies that the control be disabled (if it is not already). `enable` specifies that it be enabled (if it is not already).

`setposition` specifies the new position and size of a control. `setposition` requires *arguments* in the form of *x y xsize ysize*. A dot can be used with any of the four *arguments* to mean the current value.

Sometimes *arguments* may require quotes. For instance, `CHECKBOX` provides a special *memberfunction*

`setlabel` *string*

which sets the text shown next to the checkbox, so you might specify `onclickon('"gaction main.robust.setlabel "Robust VCE""')`. Anytime a *string* is required, you must place quotes around it if that *string* contains a space. When you specify an *iaction* inside the parentheses of an option, it is easier to leave the quotes off unless they are required. If quotes are required, you must enclose the entire contents of the option in compound double quotes as in the example above.

dialogname`.`*controlname*`.`*memberfunction* [*arguments*]
Same as `gaction`; the `gaction` is optional.

`action` *memberfunction* [*arguments*]
Same as `gaction` *currentdialog*`.`*currentcontrol*`.`*memberfunction*; executes the specified *memberfunction* on the current control.

`view` *topic*
Display *topic* in viewer; see [R] **view**.

`script` *scriptname*
Execute the specified script. A script is a set of lines, each specifying an *iaction*. So if you wanted to disable three things, `gaction` would be insufficient. You would instead define a script containing the three `gaction` lines.

`program` *programname*
Execute the specified dialog-box program. Programs can do more than scripts because they provide if-statement flow of control (among other things), but they are more difficult to write; typically, the extra capabilities are not needed when specifying i-actions.

`create STRING | DOUBLE | BOOLEAN` *propertyname*
Creates a new instance of a dialog property. See *6. Properties* for details.

`create PSTRING | PDOUBLE | PBOOLEAN` *propertyname*
Creates a new instance of a persistent dialog property. See *6. Properties* for details.

`create CHILD` *dialogname* [`AS` *referencename*] [`, nomodal allowsubmit allowcopy`]
Creates a new instance of a child dialog. By default, the reference name will be the name of the dialog unless otherwise specified. See *7. Child dialogs* for details.

2.6 U-actions and communication options

Remember that the ultimate goal of a dialog box is to construct a u-action—a Stata command to be executed. What that command is depends on how the user fills in the dialog box.

You construct the command by writing a dialog-box program, also known as a `PROGRAM`. You arrange that the program be invoked by specifying the `uaction()` option allowed with the `OK`, `SUBMIT`, `CANCEL`, and `COPY` u-action buttons. For instance, the syntax of `OK` is

`OK ... [ , ... uaction(`*pgmname*`) target(`*target*`) ... ]`

pgmname is the name of the dialog program you write, and `target()` specifies how the command constructed by *pgmname* is to be executed. Usually, you will simply want Stata to execute the command, which could be coded `target(stata)`, but because that is the default, most programmers omit the `target()` option altogether.

The dialog-box program you write accesses the information the user has filled in and outputs the Stata command to be executed. Without going into details, the program might say to construct the command by outputting the word `regress`, followed by the *varlist* the user specified in the varlist field of the first dialog, and followed by `if` *exp*, getting the expression from what the user filled in an edit field of the second dialog.

Dialogs and input controls are named, and in your dialog-box program, when you want to refer to what a user has filled in, you refer to *dialogname*`.`*inputcontrolname*. *dialogname* was determined when you coded the `DIALOG` command to create the dialog

`DIALOG` *dialogname* ...

and *inputcontrolname* was determined when you coded the input-control command to create the input control, for instance,

`CHECKBOX` *inputcontrolname* ...

The details are discussed in *5. PROGRAM*, but do not get lost in the details. Think first about coding how the dialogs look and second about how to translate what the user specifies into the u-action.

On the various commands that specify how dialogs look, you can specify an option that will make writing the u-action program easier: the communication option `option()`, which communicates something about the control to the u-action program, is allowed with every control. For instance, on the `CHECKBOX` command, you could code

`CHECKBOX ..., ... option(robust) ...`

When you wrote your dialog-box `PROGRAM`, you would find it easier to associate the `robust` option in the command you are constructing with this checkbox. Communication options never alter how a control looks or works: they just make extra information available to the `PROGRAM` and make writing the u-action routine easier.

Do not worry much about communication options when writing your dialog. Wait until you are writing the corresponding u-action program. Then it will be obvious what communication options you should have specified, and you can go back and specify them.

2.7 The distinction between i-actions and u-actions

In this documentation, we distinguish between i-actions and u-actions, but if you read carefully, you will realize that the distinction is more syntactical than real. One way we have distinguished i-actions from u-actions is to note that only u-actions can run Stata commands. In fact, i-actions can also run Stata commands; you just code them differently. In the vast majority of dialog boxes, you will not do this.

Nevertheless, if you were writing a dialog box to edit a Stata graph, you might construct your dialog box so that it contained no u-actions and only i-actions. Some of those i-actions might invoke Stata commands.

As you already know, i-actions can invoke `PROGRAM`s, and `PROGRAM`s serve two purposes: coding of i-actions and coding of u-actions. `PROGRAM`s themselves, however, have the ability to submit commands to Stata, and therein lies the key. I-actions can invoke `PROGRAM`s, and `PROGRAM`s can invoke Stata commands. How this is done is discussed in *5.1.3 rstrings: cmdstring and optstring* and *5.5 Command-execution commands*.

We recommend that you not program i-actions and u-actions that are virtually indistinguishable except in rare, special circumstances. Users expect to fill in a dialog box and to be given the opportunity to click on `OK` or `Submit` before anything too severe happens.

2.8 Error and consistency checking

In filling in the dialogs you construct, the user might make errors. One alternative is simply to ignore that possibility and let Stata complain when it executes the u-action command you construct. Even in well-written dialog boxes, most errors should be handled this way because discovering all the problems would require rewriting the entire logic of the Stata command.

Nevertheless, you will want to catch easy-to-detect errors while the dialog is still open and the user can easily fix them. Errors come in two forms: An outright error would be typing a number in an edit field that is supposed to contain a variable name. A consistency error would be checking two checkboxes that are, logically speaking, mutually exclusive.

You will want to handle most consistency errors at the dialog level, either by design (if two checkboxes are mutually exclusive, perhaps the information should be collected as radio buttons) or by i-actions (disabling or even hiding some fields depending on what has been filled in). The latter was discussed in *2.5 I-actions and member functions*.

Outright errors can be detected and handled in dialog-box programs and are usually detected and handled in the u-action program. For instance, in your dialog-box program, you can assert that *dialogname*`.`*inputcontrolname* must be filled in and pop up a custom error message if it is not, or the program code can be written so that an automatically generated error message is presented. You will find that all input-control commands have an `error()` option; for example,

`VARLIST ... [ , ... error(`*string*`) ... ]`

The `error()` string provides the text to describe the control when the dialog-box manager presents an error. For instance, if we specified

`VARLIST ... [ , ... error(`*dependent variable*`) ... ]`

the dialog-box manager might use that information later to construct the error message "dependent variable must be specified".

If you do not specify the `error()` option, the dialog-box manager will use what was specified in the `label()`; otherwise, `""` is used. The `label()` option specifies the text that usually appears near the control describing it to the user, but `label()` will do double duty so that you only need to specify `error()` when the two strings need to differ.

3. Commands

3.1 VERSION

Syntax

`VERSION` #[`.##`] [*valid_operating_systems*]

Description

`VERSION` specifies how the commands that follow are to be interpreted.

Remarks

`VERSION` must appear first in the `.dlg` file (it may be preceded by comments). In the current version of Stata, it should read `VERSION 13` or `VERSION 13.0`. It makes no difference; both mean the same thing.

Optionally, `VERSION` can specify one or more valid operating systems. Accepted values are `WINDOWS`, `MACINTOSH`, and `UNIX`. If none of these are specified, all are assumed.

Including `VERSION` at the top is of vital importance. Stata is under continual development, so syntax and features can change. Including `VERSION` is how you ensure that your dialog box will continue to work as you intended.

3.2 INCLUDE

Syntax

`INCLUDE` *includefilename*

where *includefilename* refers to *includefilename*`.idlg` and must be specified without the suffix and without a path.

Description

`INCLUDE` reads and processes the lines from *includefilename*`.idlg` just as if they were part of the current file being read. `INCLUDE` may appear in both `.dlg` and `.idlg` files.

Remarks

The name of the file is specified without a file suffix and without a path. `.idlg` files are searched for along the ado-path, as are `.dlg` files.

`INCLUDE` may appear anywhere in the dialog code and may appear in both `.dlg` and `.idlg` files; include files may `INCLUDE` other include files. Files may contain multiple `INCLUDE`s. The maximum nesting depth is 10.

3.3 DEFINE

Syntax

DEFINE *name* { . | # | +# | -# | @x | @y | @xsize | ,@ysize }

Description

DEFINE creates *name*, which may be used in other commands wherever a position or size element is required.

Remarks

The first four possibilities for defining *name*—., #, +#, and -#—specify default, number specified, positive offset, and negative offset.

The other four possibilities—@x, @y, @xsize, and @ysize—refer to the previous *x*, *y*, *xsize*, and *ysize* values, with "previous" meaning previous to the time the DEFINE command is issued, not at the time *name* is used.

3.4 POSITION

Syntax

POSITION *x y xsize ysize*

Description

POSITION is used to set the location and size of the dialog box. *x* and *y* refer to the upper-left-hand corner of the dialog box. *xsize* and *ysize* refer to the width and height of the dialog box.

Remarks

The positions *x* and *y* may each be specified as ., and Stata will determine where the dialog box will be displayed; this is recommended.

xsize and *ysize* may not be specified as . because they specify the overall size of the dialog box. You can discover the size by experimentation. If you specify a size that is too small, some elements will flow off the dialog box. If you specify a size that is too large, there will be large amounts of white space on the right and bottom of the dialog box. Good initial values for *xsize* and *ysize* are 400 and 300.

POSITION may be specified anywhere in the dialog code outside BEGIN ... END blocks. It does not matter where it is specified because the entire .dlg file is processed before the dialog box is displayed.

3.5 LIST

Syntax

```
LIST  newlistname
     BEGIN
          item
          item
          ...
     END
```

Description

LIST creates a named list for populating list and combo boxes.

Example

```
LIST choices
    BEGIN
        Statistics
        Graphics
        Data management
    END
...
DIALOG ...
    BEGIN
        ...
        LISTBOX ... , ... contents(choices) ...
        ...
    END
```

3.6 DIALOG

Syntax

```
DIALOG  newdialogname  [ , title(" string") tabtitle(" string") ]
     BEGIN
          { control definition statements | INCLUDE | DEFINE }
                    ...
     END
```

Description

DIALOG defines a dialog. Every .dlg file should define at least one dialog. Only control definition statements, INCLUDE, and DEFINE are allowed between BEGIN and END.

Options

title("*string*") defines the text to be displayed in the dialog's title bar.

tabtitle("*string*") defines the text to be displayed on the dialog's tab. Dialogs are tabbed if more than one dialog is defined. When a user clicks on the tab, the dialog becomes visible and active. If only one dialog is specified, the contents of tabtitle() are irrelevant.

Member functions

`settabtitle` *string*	sets tab title to *string*
`settitle` *string*	sets overall dialog box title to *string*

`settitle` may be called as a member function of any dialog tab, but it is more appropriate to call it as a member function of the dialog box. This is accomplished by calling it in the local scope of the dialog.

Example:

```
settitle "sort - Sort data"
```

3.6.1 CHECKBOX on/off input control

Syntax

`CHECKBOX` *newcontrolname x y xsize ysize* [`,` `label("`*string*`")` `error("`*string*`")`
`default(`*defnumval*`)` `nomemory` `groupbox` `onclickon(`*iaction*`)` `onclickoff(`*iaction*`)`
`option(`*optionname*`)` `tooltip("`*string*`")`]

Member functions

`setlabel` *string*	sets text to *string*
`setoff`	unchecks checkbox
`seton`	checks checkbox
`setoption` *optionname*	associates *optionname* with the value of the checkbox
`setdefault` *value*	sets the default value for the checkbox; this does not change the selected state
`settooltip` *string*	sets the tooltip text to *string*

The standard member functions `hide`, `show`, `disable`, `enable`, and `setposition` are also provided.

Returned values for use in PROGRAM

Returns numeric, 0 or 1, depending on whether the box is checked.

Description

`CHECKBOX` defines a checkbox control, which indicates an option that is either on or off.

Options

`label("`*string*`")` specifies the text to be displayed next to the control. You should specify text that clearly implies two opposite states so that it is obvious what happens when the checkbox is checked or unchecked.

`error("`*string*`")` specifies the text to be displayed describing this field to the user in automatically generated error boxes.

`default(`*defnumval*`)` specifies whether the box is checked or unchecked initially; it will be unchecked if *defnumval* evaluates to 0, and it will be checked otherwise. If `default()` is not specified, `default(0)` is assumed.

`nomemory` specifies that the checkbox not remember how it was filled in between invocations.

`groupbox` makes this checkbox control also a group box into which other controls can be placed to emphasize that they are related. The group box is just an outline; it does not cause the controls "inside" to be disabled or hidden or in any other way act differently than they would if they were outside the group box. On some platforms, radio buttons have precedence over checkbox group boxes. You may place radio buttons within a checkbox group box, but do not place a checkbox group box within a group of radio buttons. If you do, you may not be able to click on the checkbox control on some platforms.

`onclickon(`*iaction*`)` and `onclickoff(`*iaction*`)` specify the i-actions to be invoked when the checkbox is clicked on or off. This could be used, for instance, to hide, show, disable, or enable other input controls. The default i-action is to do nothing. The `onclickon()` or `onclickoff()` i-action will be invoked the first time the checkbox is displayed.

`option(`*optionname*`)` is a communication option that associates *optionname* with the value of the checkbox.

`tooltip("`*string*`")` specifies the text to be displayed as a tip or hint when the user hovers over the control with the mouse.

Example

```
CHECKBOX robust 10 10 100 ., label(Robust VCE)
```

3.6.2 RADIO on/off input control

Syntax

`RADIO` *newcontrolname x y xsize ysize* [`,` [`first` | `middle` | `last`] `label("`*string*`")`
`error("`*string*`")` `default(`*defnumval*`)` `nomemory` `onclickon(`*iaction*`)`
`onclickoff(`*iaction*`)` `option(`*optionname*`)` `tooltip("`*string*`")`]

Member functions

`setlabel` *string*	sets text to *string*
`seton`	checks the radio button and unchecks any other buttons in the group
`setoption` *optionname*	associates *optionname* with the value of the radio
`setdefault` *value*	sets the default value for the radio; this does not change the selected state
`settooltip` *string*	sets the tooltip text to *string*

The standard member functions `hide`, `show`, `disable`, `enable`, and `setposition` are also provided.

Returned values for use in PROGRAM

Returns numeric, 0 or 1, depending on whether the button is checked.

Description

RADIO defines a radio button control in a radio-button group. Radio buttons are used in groups of two or more to select mutually exclusive, but related, choices when the number of choices is small. Selecting one radio button automatically unselects the others in its group.

Options

first, middle, and last specify whether this radio button is the first, a middle, or the last member of a group. There must be one first and one last. There can be zero or more middle members. middle is the default if no option is specified.

label("*string*") specifies the text to be displayed next to the control.

error("*string*") specifies the text to be displayed describing this field to the user in automatically generated error boxes.

default(*defnumval*) specifies whether the radio button is to start as selected or unselected; it will be unselected if *defnumval* evaluates to 0 and will be selected otherwise. If default() is not specified, default(0) is assumed unless first is also specified, in which case default(1) is assumed. It is considered bad style to use anything other than the first button as the default, so this option is rarely specified.

nomemory specifies that the radio button not remember how it was filled in between invocations.

onclickon(*iaction*) and onclickoff(*iaction*) specify that i-action be invoked when the radio button is clicked on or clicked off. This could be used, for instance, to hide, show, disable, or enable other input controls. The default i-action is to do nothing. The onclickon() i-action will be invoked the first time the radio button is displayed if it is selected.

option(*optionname*) is a communication option that associates *optionname* with the value of the radio button.

tooltip("*string*") specifies the text to be displayed as a tip or hint when the user hovers over the control with the mouse.

Example

```
RADIO r1 10  10 100 ., first  label("First choice")
RADIO r2  @ +20   @ ., middle label("Second choice")
RADIO r3  @ +20   @ ., middle label("Third choice")
RADIO r4  @ +20   @ ., last   label("Last choice")
```

3.6.3 SPINNER numeric input control

Syntax

SPINNER *newcontrolname x y xsize ysize* [, label("*string*") error("*string*")
default(*defnumval*) nomemory min(*defnumval*) max(*defnumval*) onchange(*iaction*)
option(*optionname*) tooltip("*string*")]

Member functions

`setvalue` *value*	sets the actual value of the spinner to *value*
`setrange` *min# max#*	sets the range of the spinner to *min# max#*
`setoption` *optionname*	associates *optionname* with the value of the spinner
`setdefault` #	sets the default of the spinner to #; this does not change the value shown in the spinner control.
`settooltip` *string*	sets the tooltip text to *string*

The standard member functions `hide`, `show`, `disable`, `enable`, and `setposition` are also provided.

Returned values for use in PROGRAM

Returns numeric, the value of the spinner.

Description

`SPINNER` defines a spinner, which displays an edit field that accepts an integer number, which the user may either increase or decrease by clicking on an up or down arrow.

Options

`label("`*string*`")` specifies a description for the control, but it does not display the label next to the spinner. If you want to label the spinner, you must use a `TEXT` static control.

`error("`*string*`")` specifies the text to be displayed in describing this field to the user in automatically generated error boxes.

`default(`*defnumval*`)` specifies the initial integer value of the spinner. If not specified, `min()` is assumed, and if that is not specified, 0 is assumed.

`nomemory` specifies that the spinner not remember how it was filled in between invocations.

`min(`*defnumval*`)` and `max(`*defnumval*`)` set the minimum and maximum integer values of the spinner. `min(0)` and `max(100)` are the defaults.

`onchange(`*iaction*`)` specifies the i-action to be invoked when the spinner is changed. The default i-action is to do nothing. The `onchange()` i-action will be invoked the first time the spinner is displayed.

`option(`*optionname*`)` is a communication option that associates *optionname* with the value of the spinner.

`tooltip("`*string*`")` specifies the text to be displayed as a tip or hint when the user hovers over the control with the mouse.

Example

```
SPINNER level 10 10 60 ., label(Sig. level) min(5) max(100) ///
          default(c(level)) option(level)
```

3.6.4 EDIT string input control

Syntax

```
EDIT newcontrolname x y xsize ysize [ , label("string") error("string")
    default(defstrval) nomemory max(#) numonly password onchange(iaction)
    option(optionname) tooltip("string") ]
```

Member functions

`setlabel` *string*	sets the label for the edit field
`setvalue` *strvalue*	sets the value shown in the edit field
`append` *string*	appends *string* to the value in the edit field
`prepend` *string*	prepends *string* to the value of the edit field
`insert` *string*	inserts *string* at the current cursor position of the edit field
`smartinsert` *string*	inserts *string* at the current cursor position in the edit field with leading and trailing spaces around it
`setfocus`	causes the control to obtain keyboard focus
`setoption` *optionname*	associates *optionname* with the contents of the edit field
`setdefault` *string*	sets the default value for the edit field; this does not change what is displayed
`settooltip` *string*	sets the tooltip text to *string*

The standard member functions `hide`, `show`, `disable`, `enable`, and `setposition` are also provided.

Returned values for use in PROGRAM

Returns *string*, the contents of the edit field.

Description

`EDIT` defines an edit field. An edit field is a box into which the user may enter text or in which the user may edit text; the width of the box does not limit how much text can be entered.

Options

`label("`*string*`")` specifies a description for the control, but it does not display the label next to the edit field. If you want to label the edit field, you must use a `TEXT` static control.

`error("`*string*`")` specifies the text to be displayed describing this field to the user in automatically generated error boxes.

`default(`*defstrval*`)` specifies the default contents of the edit field. If not specified, `default("")` is assumed.

`nomemory` specifies that the edit field is not to remember how it was filled in between invocations.

`max(#)` specifies the maximum number of characters that may be entered into the edit field.

`numonly` specifies that the edit field be able to contain only a period, numeric characters 0 through 9, and `-` (minus).

`password` specifies that the characters entered into the edit field be shown on the screen as asterisks or bullets, depending on the operating system.

`onchange(`*iaction*`)` specifies the i-action to be invoked when the contents of the edit field are changed. The default i-action is to do nothing. Note that the `onchange()` i-action will be invoked the first time the edit field is displayed.

`option(`*optionname*`)` is a communication option that associates *optionname* with the contents of the edit field.

`tooltip("`*string*`")` specifies the text to be displayed as a tip or hint when the user hovers over the control with the mouse.

Example

```
TEXT tlab    10  10 200   ., label("Title")
EDIT title    @ +20   @    ., label("title")
```

3.6.5 VARLIST and VARNAME string input controls

Syntax

{ `VARLIST` | `VARNAME` } *newcontrolname x y xsize ysize* [`, label("`*string*`")`
`error("`*string*`") default(`*defstrval*`) nomemory fv ts option(`*optionname*`)`
`tooltip("`*string*`")`]

Member functions

`setlabel` *string*	sets the label for the varlist edit field
`setvalue` *strvalue*	sets the value shown in the varlist edit field
`append` *string*	appends *string* to the value in the varlist edit field
`prepend` *string*	prepends *string* to the value of the varlist edit field
`insert` *string*	inserts *string* at the current cursor position of the varlist edit field
`smartinsert` *string*	inserts *string* at the current cursor position in the varlist edit field with leading and trailing spaces around it
`setfocus`	causes the control to obtain keyboard focus
`setoption` *optionname*	associates *optionname* with the contents of the edit field
`setdefault` *string*	sets the default value for the edit field; this does not change what is displayed
`settooltip` *string*	sets the tooltip text to *string*

The standard member functions `hide`, `show`, `disable`, `enable`, and `setposition` are also provided.

Returned values for use in PROGRAM

Returns *string*, the contents of the varlist edit field.

Description

`VARLIST` and `VARNAME` are special cases of an edit field. `VARLIST` provides an edit field into which one or more Stata variable names may be entered (along with standard Stata varlist abbreviations), and `VARNAME` provides an edit field into which one Stata variable name may be entered (with standard Stata varname abbreviations allowed).

Options

`label("`*string*`")` specifies a description for the control, but does not display the label next to the varlist edit field. If you want to label the control, you must use a `TEXT` static control.

`error("`*string*`")` specifies the text to be displayed describing this field to the user in automatically generated error boxes.

`default(`*defstrval*`)` specifies the default contents of the edit field. If not specified, `default("")` is assumed.

`nomemory` specifies that the edit field not remember how it was filled in between invocations.

`fv` specifies that the control add a factor-variable dialog button.

`ts` specifies that the control add a time-series-operated variable dialog button.

`option(`*optionname*`)` is a communication option that associates *optionname* with the contents of the edit field.

`tooltip("`*string*`")` specifies the text to be displayed as a tip or hint when the user hovers over the control with the mouse.

Example

```
TEXT     dvlab     10  10 200   ., label("Dependent variable")
VARNAME  depvar     @ +20   @   ., label("dep. var")
TEXT     ivlab      @ +30   @   ., label("Independent variables")
VARLIST  idepvars   @ +20   @   ., label("ind. vars.")
```

3.6.6 FILE string input control

Syntax

`FILE` *newcontrolname x y xsize ysize* [`,` `label("`*string*`")` `error("`*string*`")`
`default(`*defstrval*`)` `nomemory` `buttonwidth(`*#*`)` `dialogtitle(`*string*`)` `save`
`multiselect` `directory` `filter(`*string*`)` `onchange(`*iaction*`)` `option(`*optionname*`)`
`tooltip("`*string*`")`]

Member functions

`setlabel` *string*	sets the label shown on the edit button
`setvalue` *strvalue*	sets the value shown in the edit field
`append` *string*	appends *string* to the value in the edit field
`prepend` *string*	prepends *string* to the value of the edit field
`insert` *string*	inserts *string* at the current cursor position of the edit field
`smartinsert` *string*	inserts *string* at the current cursor position in the edit field with leading and trailing spaces around it
`setoption` *optionname*	associates *optionname* with the contents of the edit field
`setdefault` *string*	sets the default value for the edit field; this does not change what is displayed
`settooltip` *string*	sets the tooltip text to *string*

The standard member functions `hide`, `show`, `disable`, `enable`, and `setposition` are also provided.

Returned values for use in PROGRAM

Returns *string*, the contents of the edit field (the file chosen).

Description

`FILE` is a special edit field with a button on the right for selecting a filename. When the user clicks on the button, a file dialog is displayed. If the user selects a filename and clicks on `OK`, that filename is put into the edit field. The user may alternatively type a filename into the edit field.

Options

`label("`*string*`")` specifies the text to appear on the button. The default is `("Browse ...")`.

`error("`*string*`")` specifies the text to be displayed describing this field to the user in automatically generated error boxes.

`default(`*defstrval*`)` specifies the default contents of the edit field. If not specified, `default("")` is assumed.

`nomemory` specifies that the edit field not remember how it was filled in between invocations.

`buttonwidth(#)` specifies the width in pixels of the button. The default is `buttonwidth(80)`. The overall size specified in *xsize* includes the button.

`dialogtitle(`*string*`)` is the title to show on the file dialog when you click on the file button.

`save` specifies that the file dialog allow the user to choose a filename for saving rather than one for opening.

`multiselect` specifies that the file dialog allow the user to select multiple filenames rather than only one filename.

`directory` specifies that the file dialog select a directory rather than a filename. If specified, any nonrelevant options will be ignored.

`filter(`*string*`)` consists of pairs of descriptions and wildcard file selection strings separated by "`|`", such as

```
filter("Stata Graphs|*.gph|All Files|*.*")
```

`onchange(`*iaction*`)` specifies an i-action to be invoked when the user changes the chosen file. The default i-action is to do nothing. The `onchange()` i-action will be invoked the first time the file chooser is displayed.

`option(`*optionname*`)` is a communication option that associates *optionname* with the contents of the edit field.

`tooltip("`*string*`")` specifies the text to be displayed as a tip or hint when the user hovers over the control with the mouse.

Example

```
FILE fname 10 10 300 ., error("Filename to open") label("Browse ...")
```

3.6.7 LISTBOX list input control

Syntax

```
LISTBOX newcontrolname x y xsize ysize [ , label("string") error("string")
    nomemory contents(conspec) values(listname) default(defstrval)
    ondblclick(iaction) [ onselchange(iaction) | onselchangelist(listname) ]
    option(optionname) tooltip("string") ]
```

Member functions

`setlabel` *string*	sets the label for the list box
`setvalue` *strvalue*	sets the currently selected item
`setfocus`	causes the control to obtain keyboard focus
`setoption` *optionname*	associates *optionname* with the element chosen from the list
`setdefault` *string*	sets the default value for the list box; this does not change what is displayed
`repopulate`	causes the associated contents list to rebuild itself and then updates the control with the new values from that list
`forceselchange`	forces an `onselchange` event to occur
`settooltip` *string*	sets the tooltip text to *string*

The standard member functions `hide`, `show`, `disable`, `enable`, and `setposition` are also provided.

Returned values for use in PROGRAM

Returns *string*, the text of the item chosen, or, if `values(`*listname*`)` is specified, the text from the corresponding element of *listname*.

Description

`LISTBOX` defines a list box control. Like radio buttons, a list box allows the user to make a selection from a number of mutually exclusive, but related, choices. A list box control is more appropriate when the number of choices is large.

Options

`label("`*string*`")` specifies a description for the control but does not display the label next to the control. If you want to label the list box, you must use a `TEXT` static control.

`error("`*string*`")` specifies the text to be displayed describing this field to the user in automatically generated error boxes.

`nomemory` specifies that the list box not remember the item selected between invocations.

`contents(`*conspec*`)` specifies the items to be shown in the list box. If `contents()` is not specified, the list box will be empty.

`values(`*listname*`)` specifies the list (see *3.5 LIST*) for which the values of `contents()` should match one to one. When the user chooses the *k*th element from `contents()`, the *k*th element of *listname* will be returned. If the lists do not match one to one, extra elements of *listname* are ignored, and extra elements of `contents()` return themselves.

`default(`*defstrval*`)` specifies the default selection. If not specified, or if *defstrval* does not exist, the first item is the default.

`ondblclick(`*iaction*`)` specifies the i-action to be invoked when an item in the list is double clicked. The double-clicked item is selected before the *iaction* is invoked.

`onselchange(`*iaction*`)` and `onselchangelist(`*listname*`)` are alternatives. They specify the i-action to be invoked when a selection in the list changes.

`onselchange(`*iaction*`)` performs the same i-action, regardless of which element of the list was chosen.

`onselchangelist(`*listname*`)` specifies a vector of *iactions* that should match one to one with `contents()`. If the user selects the *k*th element of `contents()`, the *k*th i-action from *listname* is invoked. See *3.5 LIST* for information on creating *listname*. If the elements of *listname* do not match one to one with the elements of `contents()`, extra elements are ignored, and if there are too few elements, the last element will be invoked for the extra elements of `contents()`.

`option(`*optionname*`)` is a communication option that associates *optionname* with the element chosen from the list.

`tooltip("`*string*`")` specifies the text to be displayed as a tip or hint when the user hovers over the control with the mouse.

Example

```
LIST ourlist
    BEGIN
        Good
        Common or average
        Poor
    END
. . .
DIALOG . . .
    BEGIN
        . . .
        TEXT ourlab    10  10 200   ., label("Pick a rating")
        LISTBOX rating  @ +20 150 200, contents(ourlist)
        . . .
    END
```

3.6.8 COMBOBOX list input control

Syntax

`COMBOBOX` *newcontrolname x y xsize ysize* [`,` `label("`*string*`")` `error("`*string*`")`
[`regular` | `dropdown` | `dropdownlist`] `default(`*defstrval*`)` `nomemory`
`contents(`*conspec*`)` `values(`*listname*`)` `append`
[`onselchange(`*iaction*`)` | `onselchangelist(`*listname*`)`] `option(`*optionname*`)`
`tooltip("`*string*`")`]

Member functions

`setlabel` *string*	sets the label for the combo box
`setvalue` *strvalue*	in the case of regular and drop-down combo boxes, sets the value of the edit field; in the case of a `dropdownlist`, sets the currently selected item
`setfocus`	causes the control to obtain keyboard focus
`setoption` *optionname*	associates *optionname* with the element chosen from the list
`setdefault` *string*	sets the default value for the combo box; this does not change what is displayed or selected
`repopulate`	causes the associated contents list to rebuild itself and then updates the control with the new values from that list
`forceselchange`	forces an `onselchange` event to occur
`settooltip` *string*	sets the tooltip text to *string*

Also, except for drop-down lists (option `dropdownlist` specified), the following member functions are also available:

`append` *string*	appends *string* to the value in the edit field
`prepend` *string*	prepends *string* to the value of the edit field
`insert` *string*	inserts *string* at the current cursor position of the edit field
`smartinsert` *string*	inserts *string* at the current cursor position in the edit field with leading and trailing spaces around it

The standard member functions `hide`, `show`, `disable`, `enable`, and `setposition` are also always provided.

Returned values for use in PROGRAM

Returns *string*, the contents of the edit field.

Description

`COMBOBOX` defines regular combo boxes, drop-down combo boxes, and drop-down-list combo boxes. By default, `COMBOBOX` creates a regular combo box; it creates a drop-down combo box if the `dropdown` option is specified, and it creates a drop-down-list combo box if the `dropdownlist` option is specified.

A regular combo box contains an edit field and a visible list box. The user may make a selection from the list box, which is entered into the edit field, or type in the edit field. Multiple selections are allowed using the `append` option. Regular combo boxes are useful for allowing multiple selections from the list as well as for allowing the user to type in an item not in the list.

A drop-down combo box contains an edit field and a list box that appears when the control is clicked on. The user may make a selection from the list box, which is entered into the edit field, or type in the edit field. The control has the same functionality and options as a regular combo box but requires less space. Multiple selections are allowed using the `append` option. Drop-down combo boxes may be cumbersome to use if the number of choices is large, so use them only when the number of choices is small or when space is limited.

A drop-down-list combo box contains a list box that displays only the current selection. Clicking on the control displays the entire list box, allowing the user to make a selection without typing in the edit field; the user chooses among the given alternatives. Drop-down-list combo boxes should be used only when the number of choices is small or when space is limited.

Options

`label("`*string*`")` specifies a description for the control but does not display the label next to the combo box. If you want to label a combo box, you must use a `TEXT` static control.

`error("`*string*`")` specifies the text to be displayed describing this field to the user in automatically generated error boxes.

`regular`, `dropdown`, and `dropdownlist` specify the type of combo box to be created.

If `regular` is specified, a regular combo box is created. `regular` is the default.

If `dropdown` is specified, a drop-down combo box is created.

If `dropdownlist` is specified, a drop-down-list combo box is created.

`default(`*defstrval*`)` specifies the default contents of the edit field. If not specified, `default("")` is assumed. If `dropdownlist` is specified, the first item is the default.

`nomemory` specifies that the combo box not remember the item selected between invocations. Even for drop-down lists—where there is no `default()`—combo boxes remember previous selections by default.

`contents(`*conspec*`)` specifies the items to be shown in the list box from which the user may choose. If `contents()` is not specified, the list box will be empty.

`values(`*listname*`)` specifies the list (see *3.5 LIST*) for which the values of `contents()` should match one to one. When the user chooses the kth element from `contents()`, the kth element of *listname* is copied into the edit field. If the lists do not match one to one, extra elements of *listname* are ignored, and extra elements of `contents()` return themselves.

`append` specifies that selections made from the combo box's list box be appended to the contents of the combo box's edit field. By default, selections replace the contents of the edit field. `append` is not allowed if `dropdownlist` is also specified.

`onselchange(`*iaction*`)` and `onselchangelist(`*listname*`)` are alternatives that specify the i-action to be invoked when a selection in the list changes.

`onselchange(`*iaction*`)` performs the same i-action, regardless of the element of the list that was chosen.

`onselchangelist(`*listname*`)` specifies a vector of *iactions* that should match one to one with `contents()`. If the user selects the kth element of `contents()`, the kth i-action from *listname* is invoked. See *3.5 LIST* for information on creating *listname*. If the elements of *listname* do not match one to one with the elements of `contents()`, extra elements are ignored, and if there are too few elements, the last element will be invoked for the extra elements of `contents()`. `onselchangelist()` should not be specified with `dropdown`.

`option(`*optionname*`)` is a communication option that associates *optionname* with the element chosen from the list.

`tooltip("`*string*`")` specifies the text to be displayed as a tip or hint when the user hovers over the control with the mouse.

Example

```
LIST namelist
    BEGIN
        John
        Sue
        Frank
    END
. . .
DIALOG . . .
    BEGIN
        . . .
        TEXT ourlab    10  10 200   ., label("Pick one or more names")
        COMBOBOX names  @ +20 150 200, contents(namelist) append
        . . .
    END
```

3.6.9 BUTTON special input control

Syntax

BUTTON *newcontrolname x y xsize ysize* [, label("*string*") error("*string*")
onpush(*iaction*) tooltip("*string*")]

Member functions

setlabel *string*	sets the label for the button
setfocus	causes the control to obtain keyboard focus
settooltip *string*	sets the tooltip text to *string*

The standard member functions hide, show, disable, enable, and setposition are also provided.

Returned values for use in PROGRAM

None.

Description

BUTTON creates a push button that performs instantaneous actions. Push buttons do not indicate a state, such as on or off, and do not return anything for use by the u-action PROGRAM. Buttons are used to invoke i-actions.

Options

label("*string*") specifies the text to display on the button. You should specify text that contains verbs that describe the action to perform.

error("*string*") specifies the text to be displayed describing this field to the user in automatically generated error boxes.

onpush(*iaction*) specifies the i-action to be invoked when the button is clicked on. If onpush() is not specified, the button does nothing.

`tooltip("`*string*`")` specifies the text to be displayed as a tip or hint when the user hovers over the control with the mouse.

Example

```
BUTTON help 10 10 80 ., label("Help") onpush("view help example")
```

3.6.10 TEXT static control

Syntax

`TEXT` *newcontrolname x y xsize ysize* [`, label("`*string*`")` [`left` | `center` | `right`]]

Member functions

`setlabel` *string* sets the text shown

The standard member functions `hide`, `show`, `disable`, `enable`, and `setposition` are also provided.

Returned values for use in PROGRAM

None.

Description

`TEXT` displays text.

Options

`label("`*string*`")` specifies the text to be shown.

`left`, `center`, and `right` are alternatives that specify the horizontal alignment of the text with respect to *x*. `left` is the default.

Example

```
TEXT dvlab 10 10 200 ., label("Dependent variable")
```

3.6.11 TEXTBOX static control

Syntax

`TEXTBOX` *newcontrolname x y xsize ysize* [`, label("`*string*`")` [`left` | `center` | `right`]]

Member functions

`setlabel` *string* sets the text shown

The standard member functions `hide`, `show`, `disable`, `enable`, and `setposition` are also provided.

Returned values for use in PROGRAM

None.

Description

TEXTBOX displays multiline text.

Options

label("*string*") specifies the text to be shown.

left, center, and right are alternatives that specify the horizontal alignment of the text with respect to *x*. left is the default.

Example

```
TEXT tx_note 10 10 200 45, label("Note ...")
```

3.6.12 GROUPBOX static control

Syntax

GROUPBOX *newcontrolname x y xsize ysize* [, label("*string*")]

Member functions

setlabel *string* sets the text shown above the group box

The standard member functions hide, show, disable, enable, and setposition are also provided.

Returned values for use in PROGRAM

None.

Description

GROUPBOX displays a frame (an outline) with text displayed above it. Group boxes are used for grouping related controls together. The grouped controls are sometimes said to be inside the group box, but there is no meaning to that other than the visual effect.

Options

label("*string*") specifies the text to be shown at the top of the group box.

Example

```
GROUPBOX weights 10 10 300 200, label("Weight type")
    RADIO w1 . . . , . . . label(fweight) first . . .
    RADIO w2 . . . , . . . label(aweight) . . .
    RADIO w3 . . . , . . . label(pweight) . . .
    RADIO w4 . . . , . . . label(iweight) last . . .
```

3.6.13 FRAME static control

Syntax

FRAME *newcontrolname x y xsize ysize* [, label("*string*")]

Member functions

There are no special member functions provided.

The standard member functions `hide`, `show`, `disable`, `enable`, and `setposition` are also provided.

Returned values for use in PROGRAM

None.

Description

FRAME displays a frame (an outline).

Options

`label("`*string*`")` specifies the label for the frame, which is not used in any way, but some programmers use it to record comments documenting the purpose of the frame.

Remarks

The distinction between a frame and a group box with no label is that a frame draws its outline using the entire dimensions of the control. A group box draws its outline a few pixels offset from the top of the control, whether there is a label or not. A frame is useful for horizontal alignment with other controls.

Example

```
FRAME box 10 10 300 200
    RADIO w1 . . . , . . . label(fweight) first . . .
    RADIO w2 . . . , . . . label(aweight) . . .
    RADIO w3 . . . , . . . label(pweight) . . .
    RADIO w4 . . . , . . . label(iweight) last . . .
```

3.6.14 COLOR input control

Syntax

COLOR *newcontrolname x y xsize ysize* [, label("*string*") error("*string*")
default(*rgbvalue*) nomemory onchange(*iaction*) option(*optionname*)
tooltip("*string*")]

Member functions

`setvalue` *rgbvalue*	sets the rgb value of the color selector
`setoption` *optionname*	associates *optionname* with the selected color
`setdefault` *rgbvalue*	sets the default rgb value of the color selector; this does not change the selected color
`settooltip` *string*	sets the tooltip text to *string*

The standard member functions `hide`, `show`, `disable`, `enable`, and `setposition` are also provided.

Returned values for use in PROGRAM

Returns *rgbvalue* of the selected color as a string.

Description

`COLOR` defines a button to access a color selector. The button shows the color that is currently selected.

Options

`label("`*string*`")` specifies a description for the control, but it does not display the label next to the button. If you want to label the color control, you must use a `TEXT` static control.

`error("`*string*`")` specifies the text to be displayed describing this field to the user in automatically generated error boxes.

`default(`*rgbvalue*`)` specifies the default color of the color control. If not specified, `default(255 0 0)` is assumed.

`nomemory` specifies that the color control not remember the set color between invocations.

`onchange(`*iaction*`)` specifies the i-action to be invoked when the color is changed. The default i-action is to do nothing. Note that the `onchange()` i-action will be invoked the first time the color control is displayed.

`option(`*optionname*`)` is a communication option that associates *optionname* with the selected color.

`tooltip("`*string*`")` specifies the text to be displayed as a tip or hint when the user hovers over the control with the mouse.

Example

```
COLOR box_color   10 10 40  ., default(0 0 0)
```

3.6.15 EXP expression input control

Syntax

`EXP` *newcontrolname x y xsize ysize* [`,` `label("`*string*`")` `error("`*string*`")`
`default(`*defstrval*`)` `nomemory` `onchange(`*iaction*`)` `option(`*optionname*`)`
`tooltip("`*string*`")`]

Member functions

`setlabel` *string*	sets the label for the button
`setvalue` *strvalue*	sets the value shown in the edit field
`append` *string*	appends *string* to the value in the edit field
`prepend` *string*	prepends *string* to the value of the edit field
`insert` *string*	inserts *string* at the current cursor position of the edit field
`smartinsert` *string*	inserts *string* at the current cursor position in the edit field with leading and trailing spaces around it
`setoption` *optionname*	associates *optionname* with the contents of the edit field
`setdefault` *string*	sets the default value for the edit field; this does not change what is displayed
`settooltip` *string*	sets the tooltip text to *string*

The standard member functions `hide`, `show`, `disable`, `enable`, and `setposition` are also provided.

Returned values for use in PROGRAM

Returns *string*, the contents of the edit field.

Description

`EXP` defines an expression control that consists of an edit field and a button for launching the Expression Builder.

Options

`label("`*string*`")` specifies the text for labeling the button.

`error("`*string*`")` specifies the text to be displayed describing this field to the user in automatically generated error boxes.

`default(`*defstrval*`)` specifies the default contents of the edit field. If not specified, `default("")` is assumed.

`nomemory` specifies that the edit field not remember how it was filled in between invocations.

`onchange(`*iaction*`)` specifies the i-action to be invoked when the contents of the edit field are changed. The default i-action is to do nothing. Note that the `onchange()` i-action will be invoked the first time the expression control is displayed.

`option(`*optionname*`)` is a communication option that associates *optionname* with the contents of the edit field.

`tooltip("`*string*`")` specifies the text to be displayed as a tip or hint when the user hovers over the control with the mouse.

Example

```
TEXT tlab   10  10 200   ., label("Expression:")
EXP exp     @  +20   @   ., label("Expression")
```

3.6.16 HLINK hyperlink input control

Syntax

HLINK *newcontrolname x y xsize ysize* [, label("*string*") [left | center | right]
onpush(*iaction*)]

Member functions

setlabel *string* sets the text shown

The standard member functions hide, show, disable, enable, and setposition are also provided.

Returned values for use in PROGRAM

None.

Description

HLINK creates a hyperlink that performs instantaneous actions. Hyperlinks do not indicate a state, such as on or off, and do not return anything for use by the u-action PROGRAM. Hyperlinks are used to invoke i-actions.

Options

label("*string*") specifies the text to be shown.

left, center, and right are alternatives that specify the horizontal alignment of the text with respect to *x*. left is the default.

onpush(*iaction*) specifies the i-action to be invoked when the hyperlink is clicked on. If onpush() is not specified, the hyperlink does nothing.

Example

```
HLINK help 10 10 80 ., label("Help") onpush("view help example")
```

3.7 OK, SUBMIT, CANCEL, and COPY u-action buttons

Syntax

{ OK | SUBMIT | COPY } *newbuttonname* [, label("*string*") uaction(*programname*)
target(*target*)]

CANCEL *newbuttonname* [, label("*string*")]

Description

OK, CANCEL, SUBMIT, and COPY define buttons that, when clicked on, invoke a u-action. At least one of the buttons should be defined (or the dialog will have no associated u-action); only one of each button may be defined; and usually, good style dictates defining all four.

OK executes *programname*, removes the dialog box from the screen, and submits the resulting command produced by *programname* to *target*. If no other buttons are defined, clicking on the close icon of the dialog box does the same thing.

SUBMIT executes *programname*, leaves the dialog box on the screen, and submits the resulting command produced by *programname* to *target*.

CANCEL removes the dialog from the screen and does nothing. If this button is defined, clicking on the close icon of the dialog box does the same thing.

COPY executes *programname*, leaves the dialog box on the screen, and copies the resulting command produced by *programname* to *target*. By default, the *target* is the *clipboard*.

You do not specify the location or size of these controls. They will be placed in the dialog box where the user would expect to see them.

Options

`label("`*string*`")` defines the text to appear on the button. The default `label()` is `OK`, `Submit`, and `Cancel` for each individual button.

`uaction(`*programname*`)` specifies the `PROGRAM` to be executed. `uaction(command)` is the default.

`target(`*target*`)` defines what is to be done with the resulting string (command) produced by *programname*. The alternatives are

`target(`*stata*`)`: The command is to be executed by Stata. This is the default.

`target(`*stata hidden*`)`: The command is to be executed by Stata, but the command itself is not to appear in the Results window. The output from the command will appear normally. **This option may change in the future and should be avoided when possible.**

`target(`*cmdwin*`)`: The command is to be placed in the Command window so that the user can edit it and then press *Enter* to submit it.

`target(`*clipboard*`)`: The command is to be placed on the clipboard so that the user can paste it into the desired editor.

Example

```
OK ok1
CANCEL can1
SUBMIT sub1
COPY copy1
```

3.8 HELP and RESET helper buttons

Syntax

HELP *newbuttonname* [, view("*viewertopic*")]

RESET *newbuttonname*

Description

HELP defines a button that, when clicked on, presents *viewertopic* in the Viewer. *viewertopic* is typically specified as "view *helpfile*".

RESET defines a button that, when clicked on, resets the values of the controls in the dialog box to their initial state, just as if the dialog box were invoked for the first time. Each time a user invokes a dialog box, its controls will be filled in with the values the user last entered. RESET restores the control values to their defaults.

You do not specify the location, size, or appearance of these controls. They will be placed in the lower-left corner of the dialog box. The HELP button will have a question mark on it, and the RESET button will have an R on it.

Option

view("*viewertopic*") specifies the topic to appear in the Viewer when the user clicks on the button. The default is view("help contents").

Example

```
HELP hlp1, view("help mycommand")
RESET res1
```

3.9 Special dialog directives

Syntax

{ MODAL | SYNCHRONOUS_ONLY }

Description

MODAL instructs the dialog to have modal behavior.

SYNCHRONOUS_ONLY allows the dialog to invoke stata hidden immediate at special times during the initialization process. See *5.5.1 stata* for more information on this topic.

4. SCRIPT

Syntax

```
SCRIPT newscriptname
    BEGIN
        iaction
        . . .
    END
```

where *iaction* is

```
.
action memberfunction
gaction dialogname.controlname.memberfunction
```

dialogname.*controlname*.*memberfunction*

`script` *scriptname*

`view` *topic*

`program` *programname*

See *2.5 I-actions and member functions* for more information on *iactions*.

Description

`SCRIPT` defines the *newscriptname*, which in turn defines a compound i-action. I-actions are invoked by the `on *()` options of the input controls. When a script is invoked, the lines are executed sequentially, and any errors are ignored.

Remarks

`CHECKBOX` provides `onclickon(`*iaction*`)` and `onclickoff(`*iaction*`)` options. Let's focus on the `onclickon(`*iaction*`)` option. If you wanted to take just one action when the box was checked—say, disabling `d1.sw2`—you could code

```
CHECKBOX . . . , . . . onclickon(d1.s2.disable) . . .
```

If you wanted to take two actions, say, disabling `d1.s3` as well, you would have to use a `SCRIPT`. On the `CHECKBOX` command, you would code

```
CHECKBOX . . . , . . . onclickon(script buttonsoff) . . .
```

and then somewhere else in the `.dlg` file (it does not matter where), you would code

```
SCRIPT buttonsoff
    BEGIN
        d1.s2.disable
        d1.s3.disable
    END
```

5. PROGRAM

Syntax

```
PROGRAM programname
    BEGIN
        [ program_line | INCLUDE ]
        [ . . . ]
    END
```

Description

`PROGRAM` defines a dialog program. Dialog programs are used to describe complicated i-actions and to implement u-actions.

Remarks

Dialog programs are used to describe complicated i-actions when flow control (if/then) is necessary or when you wish to create heavyweight i-actions that are like u-actions because they invoke Stata commands; otherwise, you should use a SCRIPT. Used this way, programs are invoked when the specified *iaction* is program *programname* in an on*() option of an input control command; for instance, you could code

```
CHECKBOX . . . , . . . onclickon(program complicated) . . .
```

or use a SCRIPT:

```
CHECKBOX . . . , . . . onclickon(script multi) . . .
        . . .
        SCRIPT multi
            BEGIN
                 . . .
                 program complicated
                 . . .
            END
```

The primary use of dialog programs, however, is to implement u-actions. The program constructs and returns a *string*, which the dialog-box manager will then interpret as a Stata command. The program is invoked by the uaction() options of OK and SUBMIT; for instance,

```
OK . . . , . . .  uaction(program command) . . .
```

The u-action program is nearly always named command because, if the uaction() option is not specified, command is assumed. The u-action program may, however, be named as you please.

Here is an example of a dialog program being used to implement an i-action with if/then flow control:

```
PROGRAM testprog
    BEGIN
        if sample.cb1 & sample.cb2 {
            call sample.txt1.disable
        }
        if !(sample.cb1 & sample.cb2) {
            call sample.txt1.enable
        }
    END
```

Here is an example of a dialog program being used to implement the u-action:

```
PROGRAM command
    BEGIN
        put "mycmd "
        varlist main.vars    // varlist [main.vars] would make optional
        ifexp main.if
        inrange main.obs1 main.obs2
        beginoptions
            option options.detail
            optionarg options.title
        endoptions
    END
```

Using programs to implement heavyweight i-actions is much like implementing u-actions, except the program might not be a function of the input controls, and you must explicitly code the stata command to execute what is constructed. Here is an example of a dialog program being used to implement a heavyweight i-action:

```
PROGRAM heavyweight
    BEGIN
        put "myeditcmd, resume"
        stata
    END
```

5.1 Concepts

5.1.1 Vnames

Vname stands for value name and refers to the "value" of a control. Vnames are of the form *dialogname.controlname*; for example, `d2.s2` and `d2.list` would be vnames if input controls `s2` and `list` were defined in `DIALOG d2`:

```
DIALOG d2 . . .
    BEGIN
        . . .
        CHECKBOX s2 . . .
        EDIT list . . .
        . . .
    END
```

A vname can be numeric or string depending on the control to which it corresponds. For `CHECKBOX`, it was documented under "Returned value for use in `PROGRAM`" that `CHECKBOX` "returns numeric, 0 or 1, depending on whether box is checked", so `d2.s2` is a numeric. For the `EDIT` input control, it was documented that `EDIT` returns a string representing the contents of the edit field, so `d2.list` is a string.

Different words are sometimes used to describe whether *vname* is numeric or string, including

vname is numeric

vname is string

vname is a numeric control

vname is a string control

vname returns a numeric result

vname returns a string result

In a program, you may not assign values to vnames; you may only examine their values and, for u-action (and heavyweight i-action) programs, output them. Thus dialog programs are pretty relaxed about types. You can ask whether `d2.s2` is true or `d2.list` is true, even though `d2.list` is a string. For a string, it is true if it is not `""`. Numeric vnames are true if the numeric result is not 0.

5.1.2 Enames

Enames are an extension of vnames. An *ename* is defined as

vname
`or(`*vname vname . . . vname*`)`
`radio(`*dialogname controlname . . . controlname*`)`

`or()` returns the *vname* of the first in the list that is true (filled in). For instance, the `varlist` u-action dialog-programming command "outputs" a varlist (see *5.1.3 rstrings: cmdstring and optstring*). If you knew that the varlist was in either control `d1.field1` or `d1.field2` and knew that both could not be filled in, you might code

```
varlist or(d1.field1 d1.field2)
```

which would have the same effect as

```
if d1.field1 {
    varlist d1.field1
}
if (!d1.field1) & d2.field2 {
    varlist d2.field2
}
```

`radio()` is for dealing with radio buttons. Remember that each radio button is a separate control, and yet, in the set, we know that exactly one is clicked on. `radio` finds the clicked one. Typing

```
option radio(d1 b1 b2 b3 b4)
```

would be equivalent to typing

```
option or(d1.b1 d1.b2 d1.b3 d1.b4)
```

which would be equivalent to typing

```
option d1.b2
```

assuming that the second radio button is selected. (The `option` command outputs the option corresponding to a control.)

5.1.3 rstrings: cmdstring and optstring

Rstrings, `cmdstring` and `optstring`, are relevant only in u-action and heavyweight i-action programs.

The purpose of a u-action program is to build and return a string, which Stata will ultimately execute. To do that, dialog programs have an *rstring* to which the dialog-programming commands implicitly contribute. For example,

```
put "kappa"
```

would add "kappa" (without the quotes) to the end of the rstring currently under construction, known as the current rstring. Usually, the current rstring is `cmdstring`, but within a `beginoptions/endoptions` block, the current rstring is switched to `optstring`:

```
beginoptions
    put "kappa"
endoptions
```

The above would add "kappa" (without the quotes) to `optstring`.

When the program concludes, the `cmdstring` and the `optstring` are put together—separated by a comma—and that is the command Stata will execute. In any case, any command that can be used outside `beginoptions/endoptions` can be used inside them, and the only difference is the rstring to which the output is directed. Thus if our entire u-action program read

```
PROGRAM command
    BEGIN
        put "kappa"
        beginoptions
            put "kappa"
        endoptions
    END
```

the result would be to execute the command "`kappa, kappa`".

The difference between a u-action program and a heavyweight i-action program is that you must, in your program, specify that the constructed command be executed. You do this with the `stata` command. The `stata` command can also be used in u-action programs if you wish to execute more than one Stata command:

```
PROGRAM command
     BEGIN
        put, etc.                  // construct first command
        stata                      // execute first command
        clear                      // clear cmdstring and optstring
        put, etc.                  // construct second command
                                   // execution will be automatic
     END
```

5.1.4 Adding to an rstring

When adding to an *rstring*, be aware of some rules in using spaces. Call *A* the rstring and *B* the string being added (say "kappa"). The following rules apply:

1. If *A* does not end in a space and *B* does not begin with a space, the two strings are joined to form "*AB*". If *A* is "`this`" and *B* is "`that`", the result is "`thisthat`".
2. If *A* ends in one or more spaces and *B* does not begin with a space, the spaces at the end of *A* are removed, one space is added, and *B* is joined to form "rightstrip(*A*) *B*". If *A* is "`this `" and *B* is "`that`", the result is "`this that`".
3. If *A* does not end in a space and *B* begins with one or more spaces, the spaces at the beginning of *B* are ignored and treated as if there is one space, and the two strings are joined to form "*A* leftstrip(*B*)". If *A* is "`this`" and *B* is "` that`", the result is "`this that`".
4. If *A* ends in one or more spaces and *B* begins with one or more spaces, the spaces at the end of *A* are removed, the spaces at the beginning of *B* are ignored, and the two strings are joined with one space in between to form "rightstrip(*A*) leftstrip(*B*)". If *A* is "`this `" and *B* is "` that`", the result is "`this that`".

These rules ensure that multiple spaces do not end up in the resulting string so that the string will look better and more like what a user might have typed.

When string literals are put, they are nearly always put with a trailing space

```
put "kappa "
```

to ensure that they do not join up with whatever is put next. If what is put next has a leading space, that space will be ignored.

5.2 Flow-control commands

5.2.1 if

Syntax

```
if ifexp {
    ...
}
```

or

```
if ifexp {
    ...
}
else {
    ...
}
```

where *ifexp* may be

ifexp	Meaning
(*ifexp*)	order of evaluation
!*ifexp*	logical not
ifexp | *ifexp*	logical or
ifexp & *ifexp*	logical and
vname	true if *vname* is not 0 and not ""
vname.booleanfunction	true if *vname.booleanfunction* evaluates to true
_rc	see *5.5 Command-execution commands*
_stbusy	true if Stata is busy
H(*vname*)	true if *vname* is hidden or disabled
default(*vname*)	true if *vname* is its default value

Note the recursive definition: An *ifexp* may be substituted into itself to produce more complicated expressions, such as ((!d1.s1) & d1.s2) | d1.s3.isdefault().

Also note that the order of evaluation is left to right; use parentheses.

booleanfunction	Meaning
`isdefault()`	true if the value of *vname* is its default value
`isenabled()`	true if *vname* is enabled
`isnumlist()`	true if the value of *vname* is a *numlist*
`isvisible()`	true if *vname* is visible
`isvalidname()`	true if the value of *vname* is a valid Stata name
`isvarname()`	true if the value of *vname* is the name of a variable in the current dataset
`iseq(`*argument*`)`	true if the value of *vname* is equal to *argument*
`isneq(`*argument*`)`	true if the value of *vname* is not equal to *argument*
`isgt(`*argument*`)`	true if the value of *vname* is greater than *argument*
`isge(`*argument*`)`	true if the value of *vname* is greater than or equal to *argument*
`islt(`*argument*`)`	true if the value of *vname* is less than *argument*
`isle(`*argument*`)`	true if the value of *vname* is less than or equal to *argument*
`isNumlistEQ(`*argument*`)`	true if every value of *vname* is equal to *argument*, where *vname* may be a *numlist*
`isNumlistLT(`*argument*`)`	true if every value of *vname* is less than *argument*, where *vname* may be a *numlist*
`isNumlistLE(`*argument*`)`	true if every value of *vname* is less than or equal to *argument*, where *vname* may be a *numlist*
`isNumlistGT(`*argument*`)`	true if every value of *vname* is greater than *argument*, where *vname* may be a *numlist*
`isNumlistGE(`*argument*`)`	true if every value of *vname* is greater than or equal to *argument*, where *vname* may be a *numlist*
`isNumlistInRange(`arg_1`,`arg_2`)`	true if every value of *vname* is in between arg_1 and arg_2 inclusive, where *vname* may be a *numlist*
`startswith(`*argument*`)`	true if the value of *vname* starts with *argument*
`endswith(`*argument*`)`	true if the value of *vname* ends with *argument*
`contains(`*argument*`)`	true if the value of *vname* contains *argument*
`iseqignorecase(`*argument*`)`	true if the value of *vname* is equal to *argument* ignoring case

An *argument* can be a dialog control, a dialog property, or a literal. If the *argument* is a literal it can be either string or numeric, depending on the type of control the *booleanfunction* references. String controls require that literals be quoted, and numeric controls require that literals not be quoted.

Description

`if` executes the code inside the braces if *ifexp* evaluates to true and skips it otherwise. When an `else` has been specified, the code within its braces will be executed if *ifexp* evaluates to false. `if` commands may be nested.

Example

```
if d1.v1.isvisible() {
    put "thing=" d1.v1
}
else {
    put "thing=" d1.v2
}
```

5.2.2 while

Syntax

```
while condition {
    ...
}
```

where *condition* may be

condition	Meaning
(*condition*)	order of evaluation
!*condition*	logical not
condition | *condition*	logical or
condition & *condition*	logical and

Description

A `while` loop is for circumstances where you want to do the same thing repeatedly. It is controlled by a counter. For a `while` loop to execute correctly, you must do the following:

1. Initialize a start value for the counter before the loop.
2. Specify a condition that tests the value of the counter against its expected final value such that the logical condition evaluates to false and the loop is forced to end at some point.
3. Specify a command that modifies the value of the counter inside the loop.

Example

```
PROGRAM testprog
    call create DOUBLE i
    call create ARRAY testlist
    while(i.islt(10)) {
            call i.withvalue testlist.Arrpush @
            call i.increment
    }
END
```

5.2.3 call

Syntax

`call` *iaction*

where *iaction* is

.
`action` *memberfunction*
`gaction` *dialogname*`.`*controlname*`.`*memberfunction*
dialogname`.`*controlname*`.`*memberfunction*
`script` *scriptname*
`view` *topic*
`program` *programname*

iaction "`action` *memberfunctionname*" is invalid in u-action programs because there is no concept of a current control.

Description

`call` executes the specified *iaction*. If an *iaction* is not specified, `gaction` is assumed.

Example

```
PROGRAM testprog
     BEGIN
         if sample.cb1 & sample.cb2 {
                 call gaction sample.txt1.disable
         }
         if !(sample.cb1 & sample.cb2) {
                 call gaction sample.txt1.enable
         }
     END
```

5.2.4 exit

Syntax

`exit` $\left[\#\right]$

where $\# \geq 0$. The following exit codes have special meaning:

#	Definition
0	exit without error
>0	exit with error
101	program exited because of a missing required object

Description

`exit` causes the program to exit and, optionally, to return #.

`exit` without an argument is equivalent to "`exit 0`". In u-action programs, the `cmdstring, optstring` will be sent to Stata for execution.

`exit` #, # $>$ 0, indicates an error. In u-action programs, the `cmdstring, optstring` will not be executed. `exit 101` has special meaning. When a u-action program exits, Stata checks the exit code for that program and, if it is `101`, presents an error box stating that the user forgot to fill in a required element of the dialog box.

Example

```
if !sample.var1 {
    exit 101
}
```

5.2.5 close

Syntax

```
close
```

Description

`close` causes the dialog box to close.

5.3 Error-checking and presentation commands

5.3.1 require

Syntax

`require` *ename* [*ename* [. . .]]

where each *ename* must be *string*.

Description

`require` does nothing on each *ename* that is disabled or hidden.

For other *enames*, `require` requires that the controls specified not be empty (`""`) and produces a stop-box error message such as "dependent variable must be defined" for any that are empty. The "dependent variable" part of the message will be obtained from the control's `error()` option or, if that was not specified, from the control's `label()` option; if that was not specified, a generic error message will be displayed.

Example

```
require main.grpvar
```

5.3.2 stopbox

Syntax

`stopbox` { `stop` | `note` | `rusure` } ["*line1*" ["*line2*" ["*line3*" ["*line4*"]]]]

Description

`stopbox` displays a message box containing up to four lines of text. Three types are available:

`stop`: Displays a message box in which there is only one button, `OK`, which means that the user must accept that he or she made an error and correct it. The program will exit after `stopbox stop`.

`note`: Displays a message box in which there is only one button, `OK`, which confirms that the user has read the message. The program will continue after `stopbox note`.

`rusure`: Displays a message box in which there are two buttons, `Yes` and `No`. The program will continue if the user clicks on `Yes` or exit if the user clicks on `No`.

Also see [P] **window stopbox** for more information.

Example

```
stopbox stop "Nothing has been selected"
```

5.4 Command-construction commands

The command-construction commands are

```
by
bysort
put
varlist
ifexp
inrange
weight
beginoptions/option/optionarg/endoptions
allowxi/xi
clear
```

Most correspond to the part of Stata syntax for which they are named:

`by` *varlist*`:` *cmd* *varlist* [*if*] [*in*] [*weight*] [`,` *options*]

`put` corresponds to *cmd* (although it is useful for other things as well), and `allowxi`/`xi` corresponds to putting `xi:` in front of the entire command; see [R] **xi**.

The command-construction commands (with the exception of `xi`) build `cmdstring` and `optstring` in the order the commands are executed (see *5.1.3 rstrings: cmdstring and optstring*), so you should issue them in the same order they are used in Stata syntax.

Added to the syntax diagrams that follow is a new header:

Use of `option()` communication.

This refers to the `option()` option on the input control definition, such as `CHECKBOX` and `EDIT`; see *2.6 U-actions and communication options*.

5.4.1 by

Syntax

`by` *ename*

where *ename* must contain a string and should refer to a `VARNAME`, `VARLIST`, or `EDIT` control.

Use of `option()` communication: None.

Description

`by` adds nothing to the current rstring if *ename* is hidden, disabled, or empty. Otherwise, `by` outputs "`by` *varlist*`:`", followed by a blank, obtaining a varlist from *ename*.

Example

```
by d2.by
```

5.4.2 bysort

Syntax

`bysort` *ename*

where *ename* must contain a string and should probably refer to a `VARNAME`, `VARLIST`, or `EDIT` control.

Use of `option()` communication: None.

Description

`bysort` adds nothing to the current rstring if *ename* is hidden, disabled, or empty. Otherwise, `bysort` outputs "`by` *varlist*`, sort :`", followed by a blank, obtaining a varlist from *ename*.

Example

```
bysort d2.by
```

5.4.3 put

Syntax

`put` [*%fmt*] *putel* [[*%fmt*] *putel* [...]]

where *putel* may be

`""`

"*string*"

vname

`/hidden` *vname*

`/on` *vname*

`/program` *programname*

The word "output" means "add to the current result" in what follows. The `put` directives are defined as

`""` and "*string*"
Outputs the fixed text specified.

vname
Outputs the value of the control.

`/hidden` *vname*
Outputs the value of the control, even if it is hidden or disabled.

`/on` *vname*
Outputs nothing if *vname*`==0`. *vname* must be numeric and should be the result of a `CHECKBOX` or `RADIO` control. `/on` outputs the text from the control's `option()` option. Also see *5.4.8.1 option* for an alternative using the `option` command.

`/program` *programname*
Outputs the `cmdstring, optstring` returned by *programname*.

If any *vname* is disabled or hidden and not preceded by `/hidden`, `put` outputs nothing.

If the directive is preceded by *%fmt*, the specified *%fmt* is always used to format the result. Otherwise, string results are displayed as is, and numeric results are displayed in `%10.0g` format and stripped of resulting leading and trailing blanks. See [D] **format**.

Use of `option()` communication: See `/on` above.

Description

`put` adds to the current rstring (outputs) what is specified.

Remarks

`put` "*string*" is often used to add the Stata command to the current rstring. When used in that way, the right way to code is

```
put "commandname "
```

Note the trailing blank on *commandname*; see *5.1.4 Adding to an rstring*.

`put` displays nothing if any element specified is hidden or disabled. For instance,

```
put "thing=" d1.v1
```

will output nothing (not even "thing=") if d1.v1 is hidden or disabled. This saves you from having to code

```
if !H(d1.v1) {
    put "thing=" d1.v1
}
```

5.4.4 varlist

Syntax

varlist *el* [*el* [...]]

where an *el* is *ename* or [*ename*] (brackets significant).

Each *ename* must be string and should be the result from a VARLIST, VARNAME, or EDIT control.

If *ename* is not enclosed in brackets, it must not be hidden or disabled.

Use of option() communication: None.

Description

varlist considers it an error if any of the specified *enames* that are not enclosed in brackets are hidden or disabled or empty (contain "").

In these cases, *varlist* displays a stop-message box indicating that the varlist must be filled in and exits the program.

varlist adds nothing to the current rstring if any of the specified *enames* that are enclosed in brackets are hidden or disabled.

Otherwise, varlist outputs with leading and trailing blanks the contents of each *ename* that is not hidden, is not disabled, and does not contain "".

Remarks

varlist is most often used to output the varlist of a Stata command, such as

```
varlist main.depvar [main.indepvars]
```

varlist can also be used for other purposes. You might code

```
if d1.vl {
    put " exog("
    varlist d2.vl
    put ") "
}
```

although coding

```
optionarg d2.vl
```

would be an easier way to achieve the same effect.

5.4.5 ifexp

Syntax

`ifexp` *ename*

where *ename* must be a string control.

Use of `option()` communication: None.

Description

`ifexp` adds nothing to the current rstring if *ename* is hidden, disabled, or empty. Otherwise, output is "`if` *exp*", with spaces added before and after.

Example

```
if d2.if
```

5.4.6 inrange

Syntax

`inrange` *ename_1* *ename_2*

where *ename_1* and *ename_2* must be numeric controls.

Use of `option()` communication: None.

Description

If *ename_1* is hidden or disabled, results are as if *ename_1* were not hidden and contained 1. If *ename_2* is hidden or disabled, results are as if *ename_1* were not hidden and contained `_N`, the number of observations in the dataset.

If *ename_1*`==1` and *ename_2*`==_N`, nothing is output (added to the current rstring).

Otherwise, "`in` *range*" is output with spaces added before and after, with the range obtained from *ename_1* and *ename_2*.

Example

```
inrange d2.in1 d2.in2
```

5.4.7 weight

Syntax

`weight` *ename_t* *ename_e*

where *ename_t* may be a string or numeric control and must have had `option()` filled in with a weight type (one of `weight`, `fweight`, `aweight`, `pweight`, or `iweight`), and *ename_e* must be a string evaluating to the weight expression or variable name.

Use of `option()` communication: *ename_t* must have `option()` filled in the weight type.

Description

`weight` adds nothing to the current rstring if *ename_t* or *ename_e* are hidden, disabled, or empty. Otherwise, output is [*weighttype*=*exp*] with leading and trailing blanks.

Remarks

`weight` is typically used as

```
weight radio(d1 w1 w2 . . . wk) d1.wexp
```

where `d1.w1`, `d1.w2`, . . . , `d1.wk` are radio buttons, which could be defined as

```
DIALOG d1 . . .
    BEGIN
    . . .
        RADIO w1 . . . , . . . label(fweight) first . . .
        RADIO w2 . . . , . . . label(aweight) . . .
        RADIO w3 . . . , . . . label(pweight) . . .
        RADIO w4 . . . , . . . label(iweight) last . . .
        . . .
    END
```

Not all weight types need to be offered. If a command offers only one kind of weight, you do not need to use radio buttons. You could code

```
weight d1.wt d1.wexp
```

where `d1.wt` was defined as

```
CHECKBOX wt . . . , . . . label(fweight) . . .
```

5.4.8 beginoptions and endoptions

Syntax

```
beginoptions
    any dialog-programming command except beginoptions
    . . .
endoptions
```

Use of `option()` communication: None.

Description

`beginoptions/endoptions` indicates that you wish what is enclosed to be treated as Stata options in constructing `cmdstring, optstring`.

The current rstring is, by default, `cmdstring`. `beginoptions` changes the current rstring to `optstring`. `endoptions` changes it back to `cmdstring`. So there are two strings being built. When the dialog program exits normally, if there is anything in `optstring`, trailing spaces are removed from `cmdstring`, a comma and a space are added, the contents of `optstring` are added, and all that is returned. Thus a dialog program can have many `beginoptions/endoptions` blocks, but all the options will appear at the end of the `cmdstring`.

The command-construction commands `option` and `optionarg` are documented below because they usually appear inside a `beginoptions/endoptions` block, but they can be used outside `beginoptions/endoptions` blocks, too. Also all the other command-construction commands can be used inside a `beginoptions/endoptions` block, and using `put` is particularly common.

5.4.8.1 option

Syntax

`option` *ename* [*ename* [`. . .`]]

where *ename* must be a numeric control with 0 indicating that the option is not desired.

Use of `option()` communication: `option()` specifies the name of the option.

Description

`option` adds nothing to the current rstring if any of the *enames* specified are hidden or disabled. Otherwise, for each *ename* specified, if *ename* is not equal to 0, the contents of its `option()` are displayed.

Remarks

`option` is an easy way to output switch options such as `noconstant` and `detail`. You simply code

```
option d1.sw
```

where you have previously defined

```
CHECKBOX sw . . . , option(detail) . . .
```

Here `detail` will be output if the user checked the box.

5.4.8.2 optionarg

Syntax

`optionarg` [*style*] *ename* [[*style*] *ename* [`.  .  .`]]

where each *ename* may be a numeric or string control and *style* is

style	Meaning
`/asis`	do not quote
`/quoted`	do quote
`/oquoted`	quote if necessary
%fmt	for use with numeric

Use of `option()` communication: `option()` specifies the name of the option.

Description

`optionarg` adds nothing to the current rstring if any of the *enames* specified are hidden or disabled. Otherwise, for each *ename* specified, if *ename* is not equal to `""`, the *ename*'s `option()` is output, followed by "`(`", the *ename*'s contents, and "`)`" with blanks added before and after.

Remarks

optionarg is an easy way to output single-argument options such as title() or level(); for example,

```
optionarg /oquoted d1.ttl
if ! d1.level.isdefault() {
    optionarg d1.level
}
```

where you have previously defined

```
EDIT    ttl   . . . , . . . label(title) . . .
SPINNER level . . . , . . . label(level) . . .
```

5.5 Command-execution commands

Commands are executed automatically when a program is invoked by an input control's uaction() option. Programs so invoked are called u-action programs. No command is executed when a program is invoked by an input control's iaction() option. Programs so invoked are called i-action programs.

The stata and clear commands are for use if

1. you want to write a u-action program that executes more than one Stata command, or
2. you want to write an i-action program that executes one or more Stata commands (also known as heavyweight i-action programs).

5.5.1 stata

Syntax

```
stata
stata hidden [immediate|queue]
```

Use of option() communication: None.

Description

stata executes the current cmdstring, optstring and displays the command in the Results window before execution, just as if the user had typed it.

stata hidden executes the current cmdstring, optstring but does not display the command in the Results window before execution. stata hidden may optionally be called with either of two modifiers: queue or immediate. If neither modifier is specified, immediate is implied.

immediate causes the command to execute at once, waits for the command to finish, and sets _rc to contain the return code. Because the command is to be executed immediately, the dialog engine will complain if Stata is not idle.

queue causes the command to be placed into the command buffer, allowing it to be executed as soon as Stata becomes idle. The behavior of stata and stata hidden queue are identical except that stata hidden queue does not echo the command.

Important notes about stata hidden immediate

A unique situation can occur when `stata hidden immediate` is used in an initialization script or program. Stata dialogs are considered asynchronous, meaning that Stata dialogs can be loaded through the menu and help systems even when Stata is busy processing an ado program. Because `stata hidden immediate` relies on ado processing and because ado processing is synchronous, dialogs that call `stata hidden immediate` during initialization can only be used synchronously. That means these types of dialogs cannot be loaded while Stata is busy processing other tasks. Because of this, the dialog must be notified that it is special in this regard. This is done by placing the dialog directive `SYNCHRONOUS_ONLY` in the dialog box program just after the `VERSION` statement.

`SYNCHRONOUS_ONLY` performs one other important function. Dialogs that are launched by using the `db` command cause Stata to become busy and remain busy until the dialog is completely loaded. After all, `db` is an ado program, and until the dialog loads and `db` subsequently exits execution, Stata is busy. The `SYNCHRONOUS_ONLY` directive lets the dialog engine know that executing `stata hidden immediate` during initialization routines is allowed even when the dialog is launched with an ado program.

5.5.2 clear

Syntax

`clear` [`curstring` | `cmdstring` | `optstring`]

Use of `option()` communication: None.

Description

`clear` is seldom used and is typically specified without arguments. `clear` clears (resets to `""`) the specified return string or, if it is specified without arguments, clears `cmdstring` and `optstring`. If `curstring` is specified, `clear` clears the current return string, which is `cmdstring` by default or `optstring` within a `beginoptions/endoptions` block.

5.6 Special scripts and programs

Sometimes, it may be useful to have a script or program run automatically, either just before dialog-box controls are created or just after. The following scripts and programs are special, and when they are defined, they run automatically.

Name	Function
`PREINIT_SCRIPT`	script that runs before any dialog box controls are created
`PREINIT_PROGRAM`	program that runs before any dialog box controls are created
`POSTINIT_SCRIPT`	script that runs after all dialog box controls are created
`POSTINIT_PROGRAM`	program that runs after all dialog box controls are created
`PREINIT`	shortcut for `PREINIT_SCRIPT`
`POSTINIT`	shortcut for `POSTINIT_SCRIPT`
`ON_DOTPROMPT`	program that runs when Stata returns from executing an interactive command; `ON_DOTPROMPT` program should never call the dialog system's `stata` command, because that would result in infinite recursion

Often it is desirable to encapsulate individual dialog tabs into `.idlg` files, particularly when a dialog tab is used in many different dialog boxes. In those circumstances, a dialog tab can use its own initialization script or program. The following naming conventions are used to define these scripts and programs.

Name	Function
*tabname*_PREINIT_SCRIPT	script that runs before controls on dialog *tabname* are created
*tabname*_PREINIT_PROGRAM	program that runs before controls on dialog *tabname* are created
*tabname*_POSTINIT_SCRIPT	script that runs after controls on dialog *tabname* are created
*tabname*_POSTINIT_PROGRAM	program that runs after controls on dialog *tabname* are created
*tabname*_PREINIT	shortcut for *tabname*_PREINIT_SCRIPT
*tabname*_POSTINIT	shortcut for *tabname*_POSTINIT_SCRIPT

The order of execution for dialog initialization is as follows:

1. Execute PREINIT script or program for the dialog box.
2. Execute PREINIT scripts and programs for each dialog tab using the order in which the tabs are created.
3. Create all controls for the entire dialog box.
4. Execute POSTINIT scripts and programs for each dialog tab using the order in which the tabs are created.
5. Execute POSTINIT script or program for the dialog box.

6. Properties

Properties are used to store information that is useful for dialog box programming. Properties may be of type STRING, DOUBLE, or BOOLEAN and do not have a visual representation on the dialog box. Special variants of these basic types are available. These variants, PSTRING, PDOUBLE, and PBOOLEAN, are considered persistent and are identical to their counterparts. The contents of these persistent types do not get destroyed when a dialog is reset. Usually, the base types should be used. Application of the persistent types should be reserved for special circumstances. See `create` for information about creating new instances of a property.

Member functions

STRING

propertyname`.setvalue` *strvalue*
propertyname`.setstring` *strvalue*; synonym for `.setvalue`
propertyname`.append` *strvalue*
propertyname`.tokenize` *classArrayName*
propertyname`.tokenizeOnStr` *classArrayName strvalue*
propertyname`.tokenizeOnChars` *classArrayName strvalue*
propertyname`.expandNumlist`
propertyname`.storeDialogClassName`
propertyname`.storeClsArrayToQuotedStr` *classArrayName*

DOUBLE

propertyname`.setvalue` *value*
propertyname`.increment`

	propertyname`.decrement` *propertyname*`.storeClsArraySize` *classArrayName*
`BOOLEAN`	*propertyname*`.settrue` *propertyname*`.setfalse` *propertyname*`.storeClsObjectExists` *objectName*

Special definitions

strvalue	Definition
"*string*"	quoted string literal
`literal` *string*	same as *string*
`c(`*name*`)`	contents of `c(`*name*`)`; see [P] **creturn**
`r(`*name*`)`	contents of `r(`*name*`)`; see [P] **return**
`e(`*name*`)`	contents of `e(`*name*`)`; see [P] **ereturn**
`s(`*name*`)`	contents of `s(`*name*`)`; see [P] **return**
`char` *varname*`[`*charname*`]`	value of characteristic; see [P] **char**
`global` *name*	contents of global macro `$`*name*
`class` *objectName*	contents of a class system object; object name may be a fully qualified object name, or it may be given in the scope of the dialog box

value	Definition
#	a numeric literal
`literal` #	same as #
`c(`*name*`)`	value of `c(`*name*`)`; see [P] **creturn**
`r(`*name*`)`	value of `r(`*name*`)`; see [P] **return**
`e(`*name*`)`	value of `e(`*name*`)`; see [P] **ereturn**
`s(`*name*`)`	value of `s(`*name*`)`; see [P] **return**
`global` *name*	value of global macro `$`*name*
`class` *objectName*	contents of a class system object. The object name may be a fully qualified object name or it may be given in the scope of the dialog box.

7. Child dialogs

Syntax

`create CHILD` *dialogname* [`AS` *referenceName*] [`, nomodal allowsubmit allowcopy message(`*string*`)`]

Member functions

`settitle` *string*	sets the title text of the child dialog box
`setExitString` *string*	informs the child where to save the command string when the `OK` or `Submit` button is clicked on
`setOkAction` *string*	informs the child that it is to invoke a specific action in the parent when the `OK` button is clicked on and the child exits
`setSubmitAction` *string*	informs the child that it is to invoke a specific action in the parent when the `Submit` button is clicked on
`setExitAction` *string*	informs the child that it is to invoke a specific action in the parent when the `OK` or `Submit` button is clicked on; note that `setExitAction` has the same effect as calling both `setOkAction` and `setSubmitAction` with the same argument
`create` *property*	allows the parent to create properties in the child; see *6. Properties*
`callthru` *gaction*	allows the parent to call global actions in the context of the child

Description

Child dialogs are dialogs that are spawned by another dialog. These dialogs form a relationship where the initial dialog is referred to as the parent and all dialogs spawned from that parent are referred to as its children. In most circumstances, the children collect information and return that information to the parent for later use. Unless `AS` *referencename* has been specified, children are referenced through the *dialogname*.

Options

`nomodal` suppresses the default modal behavior of a child dialog unless the `MODAL` directive was specifically used inside the child dialog resource file.

`allowsubmit` allows for the use of the `Submit` button on the dialog box. By default, the `Submit` button is removed if it has been declared in the child dialog resource file.

`allowcopy` allows for the use of the `Copy` button on the dialog box. By default, the `Copy` button is removed if it has been declared in the child dialog resource file.

`message(`*string*`)` specifies that *string* be passed to the child dialog box, where it can be referenced from `STRING` property named `__MESSAGE`.

7.1 Referencing the parent

While it is normally not necessary, it is sometimes useful for a child dialog box to give special instructions or information to its parent. All child dialog boxes contain a special object named `PARENT`, which can be used with a member program named `callthru`. `PARENT.callthru` can be used to call any intermediate action in the context of the parent dialog box.

8. Example

The following example will execute the `summarize` command. In addition to the copy below, a copy can be found among the Stata distribution materials. You can type

```
. which sumexample.dlg
```

to find out where it is.

```
——————————————— sumexample.dlg ———————————————————————————————————————
// sumexample
// version 1.0.0
VERSION 13
POSITION . . 320 200
DIALOG main, title("Example simple summarize dialog") tabtitle("Main")
BEGIN
    TEXT     lab      10   10  300    ., label("Variables to summarize:")
    VARLIST  vars      @  +20    @    ., label("Variables to sum")
END
DIALOG options, tabtitle("Options")
BEGIN
    CHECKBOX detail   10   10  300    ., ///
          label("Show detailed statistics") ///
          option("detail") ///
          onclickoff(`""options.status.setlabel "(detail is off)""') ///
          onclickon(`""gaction options.status.setlabel "(detail is on)""')
    TEXT     status    @  +20    @    ., ///
          label("This label won't be seen")
    BUTTON   btnhide   @  +30  200    ., ///
          label("Hide other controls") push("script hidethem")
    BUTTON   btnshow   @  +30    @    ., ///
          label("Show other controls") push("script showthem")
    BUTTON   btngrey   @  +30    @    ., ///
          label("Disable other controls") push("script disablethem")
    BUTTON   btnnorm   @  +30    @    ., ///
          label("Enable other controls") push("script enablethem")
END
```

```
SCRIPT hidethem
BEGIN
    gaction main.lab.hide
    main.vars.hide
    options.detail.hide
    options.status.hide
END

SCRIPT showthem
BEGIN
    main.lab.show
    main.vars.show
    options.detail.show
    options.status.show
END

SCRIPT disablethem
BEGIN
    main.lab.disable
    main.vars.disable
    options.detail.disable
    options.status.disable
END

SCRIPT enablethem
BEGIN
    main.lab.enable
    main.vars.enable
    options.detail.enable
    options.status.enable
END

OK      ok1, label("Ok")
CANCEL  can1
SUBMIT  sub1
HELP    hlp1, view("help summarize")
RESET   res1

PROGRAM command
BEGIN
    put "summarize"
    varlist main.vars   /* varlist [main.vars] to make it optional */
    beginoptions
        option options.detail
    endoptions
END
```

sumexample.dlg

Appendix A: Jargon

action: See *i-action* and *u-action*.

browser: See *file chooser*.

button: A type of input control; a button causes an i-action to occur when it is clicked on. Also see *u-action buttons*, *helper buttons*, and *radio buttons*.

checkbox: A type of numeric input control; the user may either check or uncheck what is presented; suitable for obtaining yes/no responses. A checkbox has value 0 or 1, depending on whether the item is checked.

combo box: A type of string input control that has an edit field at the top and a list box underneath. Combo boxes come in three flavors:

A regular combo box has an edit field and a list below it. The user may choose from the list or type into the edit field.

A drop-down combo box also has an edit field and a list, but only the edit field shows. The user can click to expose the list. The user may choose from the list or type in the edit field.

A drop-down-list combo box is more like a list box. An edit field is displayed. The list is hidden, and the user can click to expose the list, but the user can only choose elements from the list; he or she cannot type in the edit field.

control: See *input control* and *static control*.

control status: Whether a control (input or static) is disabled or enabled, hidden or shown.

dialog(s): The main components of a dialog box in that the dialogs contain all the controls except for the u-action buttons.

dialog box: Something that pops up onto the screen that the user fills in; when the user clicks on an action button, the dialog box causes something to happen (namely, Stata to execute a command).

A dialog box is made up of one or more dialogs, u-action buttons, and a title bar.

If the dialog box contains more than one dialog, only one of the dialogs shows at a time, which one being determined by the tab selected.

dialog program: See *PROGRAM*.

disabled and enabled: A control that is disabled is visually grayed out; otherwise, it is enabled. The user cannot modify disabled input controls. Also see *hidden and exposed*.

.dlg file: The file containing the code defining a dialog box and its actions. If the file is named *xyz*`.dlg`, the dialog box is said to be named *xyz*.

dlg-program: The entire contents of a `.dlg` file; the code defining a dialog box and its actions.

edit field: A type of string input control; a box in which the user may type text.

enabled and disabled: See *hidden and exposed*.

exposed and hidden: See *hidden and exposed*.

file browser: See *file chooser*.

file chooser: A type of string input control; presents a list of files from which the user may choose one or type a filename.

frame: A type of static control; a rectangle drawn around a group of controls.

group box: A type of static control; a rectangle drawn around a group of controls with descriptive text at the top.

helper buttons: The buttons **Help** and **Reset**. When **Help** is clicked on, the help topic for the dialog box is displayed. When **Reset** is clicked on, the control values of the dialog box are reset to their defaults.

hidden and exposed: A control that is removed from the screen is said to be hidden; otherwise, it is exposed. Hidden input controls cannot be manipulated by the user. A control would also not be shown when it is contained in a dialog that does not have its tab selected in a multidialog dialog box; in this case, it may be invisible, but whether it is hidden or exposed is another matter. Also see *hidden and exposed*.

i-action: An intermediate action usually caused by the interaction of a user with an input control, such as hiding or showing and disabling or enabling other controls; opening the Viewer to display something; or executing a `SCRIPT` or a `PROGRAM`.

input control: A screen element that the user fills in or sets. Controls include checkboxes, buttons, radio buttons, edit fields, spinners, file choosers, etc. Input controls have (set) values, which can be string, numeric, or special. These values reflect how the user has "filled in" the control. Input controls are said to be string or numeric depending on the type of result they obtain (and how they store it).

Also see *static control*.

label or title: See *title or label*.

list: A programming concept; a vector of elements.

list box: A type of string input control; presents a list of items from which the user may choose. A list box has (sets) a string value.

numeric input control: An input control that returns a numeric value associated with it.

position: Where something is located, measured from the top left by how far to the right it is (*x*) and how far down it is (*y*).

PROGRAM: A programming concept dealing with the implementation of dialogs. `PROGRAM`s may be used to implement i-actions or u-actions. Also see *SCRIPT*.

radio buttons: A set of numeric input controls, each a button, of which only one may be selected at a time; suitable for obtaining categorical responses. Each radio button in the set has (sets) a numeric value, 0 or 1, depending on which button is selected. Only one in the set will be 1.

SCRIPT: A programming concept dealing with the implementation of dialogs. An array of i-actions to be executed one after the other; errors that occur do not stop subsequent actions from being attempted. Also see *PROGRAM*.

size: How large something is, measured from its top-left corner, as a width (*xsize*) and height (*ysize*). Height is measured from the top down.

spinner: A type of numeric input control; presents a numeric value that the user may increase or decrease over a range. A spinner has (sets) a numeric value.

static control: A screen element similar to an input control, except that the end user cannot interact with it. Static controls include static text and lines drawn around controls visually to group them together (group boxes and frames). Also see *control* and *input control*.

static text: A static control specifying text to be placed on a dialog.

string input control: An input control that returns a string value associated with it.

tabs: The small labels at the top of each dialog (when there is more than one dialog associated with the dialog box) and on which the user clicks to select the dialog to be filled in.

title or label: The fixed text that appears above or on objects such as dialog boxes and buttons. Controls are usually said to be labeled, whereas dialog boxes are said to be titled.

u-action: What a dialog box causes to happen after the user has filled it in and clicked on a u-action (ultimate action) button. The point of a dialog box is to result in a u-action.

u-action buttons: The buttons `OK`, `Submit`, `Cancel`, and `Copy`; clicking on one causes the ultimate action (u-action) associated with the button to occur and, perhaps, the dialog box to close.

varlist or varname control: A type of string input control; an edit field that also accepts input from the Variables window. This control also contains a combo-box-style list of the variables. A varlist or varname control has (sets) a string value.

Appendix B: Class definition of dialog boxes

Dialog boxes are implemented in terms of class programming; see [P] **class**.

The top-level class instance of a dialog box defined in *dialogbox*`.dlg` is `.`*dialogbox*`_dlg`. Dialogs and controls are nested within that, so `.`*dialogbox*`_dlg.`*dialogname* would refer to a dialog, and `.`*dialogbox*`_dlg.`*dialogname*`.`*controlname* would refer to a control in the dialog.

`.`*dialogbox*`_dlg.`*dialogname*`.`*controlname*`.value` is the current value of the control, which will be either a `string` or a `double`. You must not change this value.

The member functions of the controls are implemented as member functions of `.`*dialogbox*`_dlg.`*dialogname*`.`*controlname* and may be called in the standard way.

Appendix C: Interface guidelines for dialog boxes

One of Stata's strengths is its strong support for cross-platform use—datasets and programs are completely compatible across platforms. This includes dialogs written in the dialog-programming language. Although Mac, Windows, and X Windows share many common graphical user-interface elements and concepts, they all vary slightly in their appearance and implementation. This variation makes it difficult to design dialogs that look and behave the same across all platforms. Dialogs should look pleasant on screen to enhance their usability, and achieving this goal often means being platform specific when laying out controls. This often leads to undesirable results on other platforms.

The dialog-programming language was written with this in mind, and dialogs that appear and behave the same across multiple operating systems and appear pleasant can be created by following some simple guidelines.

Use default heights where applicable: Varying vertical-size requirements of controls across different operating systems can cause a dialog that appears properly on one platform to display controls that overlap one another on another platform. Using the default *ysize* of `.` takes these variations into account and allows for much easier placement and alignment of controls. Some controls (list boxes, regular combo boxes, group boxes, and frames) still require their *ysize* to be specified because their vertical size determines how much information they can reveal.

Use all horizontal space available: Different platforms use different types of fonts to display text labels and control values. These variations can cause some control labels to be truncated (or even word wrapped) if their *xsize* is not large enough for a platform's system font. To prevent this from happening, specify an *xsize* that is as large as possible. For each column of controls, specify the entire column width for each control's *xsize*, even for controls where it is obviously unnecessary. This reduces the chances of a control's label being truncated on another platform and also allows you to make changes to the label without constantly having to adjust the *xsize*. If your control barely fits into the space allocated to it, consider making your dialog slightly larger.

Use the appropriate alignment for static text controls: The variations in system fonts also make it difficult to horizontally align static text controls with other controls. Placing a static text control next to an edit field may look good on one platform but show up with too much space between the controls on another or even show up truncated.

One solution is to place static text controls above controls that have an edit field and make the static text control as wide as possible. This gives more room for the static text control and makes it easier to left-justify it with other controls.

When placing a static text control to the left of a control is more appropriate (such as From: and To: edit fields), use right-alignment rather than the default left-alignment. The two controls will then be equally spaced apart on all platforms. Again be sure to make the static text control slightly

wider than necessary—do not try to left-justify a right-aligned static text control with controls above and below it because it may not appear left-justified on other platforms or may even be truncated.

Do not crowd controls: Without making your dialog box unnecessarily large, use all the space that is available. Organize related controls close together, and put some distance between unrelated ones. Do not overload users with lots of controls in one dialog. If necessary, group controls in separate dialogs. Most importantly, be consistent in how you layout controls.

All vertical size and spacing of controls involves multiples of 10 pixels: The default *ysize* for most controls is 20 pixels. Related controls are typically spaced 10 pixels apart, and unrelated ones are at least 20 pixels apart.

Use the appropriate control for the job: Checkboxes have two states: on or off. A radio-button group consisting of two radio buttons similarly has two states. A checkbox is appropriate when the action taken is either on or off or easy to infer (for example, **Use constant**). A two-radio-button group is appropriate when the opposite state cannot be inferred (for example, **Display graph** and **Display table**).

Radio-button groups should contain at least two radio buttons and no more than about seven. If you need more choices, consider using a drop-down-list combo box or, if the number of choices is greater than about 12, a list box. If you require a control that allows multiple selections, consider a regular combo box or drop-down combo box. Drop-down combo boxes can be cumbersome to use if the number of choices is great, so use a regular combo box unless space is limited.

Understand control precedence for mouse clicks: Because of the limited size of dialogs, you may want to place several controls within the same area and hide and show them as necessary. It is also useful to place controls within other controls, such as group boxes and frames, for organizational and presentational purposes. However, the order of creation and placement and size of controls can affect which controls receive mouse clicks first or whether they receive them at all.

The control where this can be problematic is the radio button. On some platforms, the space occupied by a group of radio buttons is not the space occupied by the individual radio buttons. It is inclusive to the space occupied by the radio button that is closest to the top-left corner of the dialog, the widest radio button, and the bottommost radio button. To prevent a group of radio buttons from preventing mouse clicks being received by other controls, Stata gives precedence to all other controls except for group boxes and frames. The order of precedence for controls that can receive mouse clicks is the following: first, all controls other than radio buttons and checkbox group boxes, then radio buttons, then checkbox group boxes.

If you intend to place two or more groups of radio buttons in the same area and show and hide them as necessary, be sure that when you hide the radio buttons from a group, you hide all radio buttons from a group. The radio-button group with precedence over other groups will continue to have precedence as long as any of its radio buttons are visible. Mouse clicks in the space occupied by nonvisible radio buttons in a group with precedence will not pass through to any other groups occupying the same space.

It is always safe to place controls within frames, group boxes, and checkbox group boxes because all other controls take precedence over those controls.

In practice, you should never hide a radio button from a group without hiding the rest of the radio buttons from the group. Consider simply disabling the radio button or buttons instead. It is also not a good idea to hide or show radio buttons from different groups to make them appear that they are from the same group. That simply will not work on some platforms and is generally a bad idea, anyway.

Radio buttons have precedence over checkbox group boxes. You may place radio buttons within a checkbox group box, but do not place a checkbox group box within the space occupied by a group of radio buttons. If you do, you may not be able to click on the checkbox control on some platforms.

Frequently asked questions

See dialog programming FAQs on the Stata website.

Also see

[P] **window programming** — Programming menus and windows

[R] **db** — Launch dialog