New dialogs interface in AlphaTcl

Vince Darley and Lars Hellström

November 11, 2001

Abstract

This document describes the programmer's interface to the generic dialog procedures available in AlphaTcl. This is quite independent of the numerous, but rather special, *Alpha* preference dialogs, which are instead based entirely on information given in the preference declarations.

Besides describing the interface, this document also contains the (highly documented) master source for the implementation of this interface.

Contents

Conventions used in this paper

In syntax descriptions, a `typewriter` font is used for explicit text. A named syntactic unit is written as ⟨*unit*⟩. In the special but very common case that the syntactic unit is precisely a word for Tcl, it is instead written as {*word*}, i.e., with braces instead of angle brackets. Optional and repeated elements in syntax descriptions are denoted as in regular expressions, using question marks, asterisks, and plus signs, e.g.

`set` {*var-name*} {*value*}$^?$
`list` {*item*}*
`append` {*var-name*} {*string*}$^+$

Parentheses can be used to group syntax elements, e.g.

`return` $\big($`-code` {*code*}$\big)^?$ {*string*}

The same conventions are used for specifying the structure of lists.

1 Usage

`dialog::make` (proc) The generic dialogs interface provides the two general purpose dialog creators `dialog::make` and `dialog::make_paged`. The basic syntax of the former procedure is

`dialog::make` ⟨*option*⟩* {*page*}$^+$

where each {*page*} is a list with the structure

{*page name*} {*item*}*

and each {*item*} in turn is a list with the structure

{*type*} {*name*} {*value*} {*help*}$^?$

Each {*item*} gives a logical description (type, name, and initial value, but no metric information) of an item in a dialog. `dialog::make` generates from these the corresponding *dialog material* (argument sequence for the `dialog` command), calls `dialog` with those data, and interprets the result. Then `dialog::make` returns the list of the final edited values of the dialog items (just a flat list), or returns an error if the dialog was cancelled.

An example should serve to clarify this. The command

```
dialog::make\
  {First {var Hey 1} {flag blah 0} {folder hey ""}}\
  {Second {var Hey 2}}
```

will create a dialog with two pages, named First and Second. The first page contains three dialog items: a variable (editable text box), a flag (checkbox), and a folder item. These are named Hey, blah, and hey respectively, and have current values `1`, `0` (not checked), and `""` (empty string, i.e., not set) respectively. The second page contains a single variable also named Hey which has the current value `2`. Immediately clicking OK (the dialog has one OK and one Cancel button) will return the list

```
1 0 {} 2
```

but if you first type e.g. `Hay` in the first Hey box, types `hey hey hey` in the second, and checks the blah checkbox before you click OK then the returned list will instead be

```
Hay 1 {} {hey hey hey}
```

`dialog::make_paged` (proc) The `dialog::make_paged` procedure is similar, but the argument structure is slightly different. The basic syntax is similarly

`dialog::make_paged` ⟨*option*⟩* {*page*}+

but here each {*page*} is a list with the structure

{*page name*} {*keyval list*} {*item list*}

and the return value is a list with the structure

({*page name*} {*keyval list*})*

The idea here is that the data structure that the values are stored in is the same in both input and output, so that the caller can almost completely avoid reconstructing large structures. This is of course given that the item values are normally stored as {*keyval list*}s, but that is a very convenient format in Tcl, thanks to the `array get` and `array set` commands.

In general, a {*keyval list*} is a list with the structure

({*key*} {*value*})*

i.e., with alternating {*key*} and {*value*} elements. The relative order of these pairs is irrelevant, the only thing that matters is which {*key*} goes with which {*value*}. When such a list is given to `array set` it will use the {*key*}s as indices into an array and set those entries to the corresponding {*value*}s. This makes it fairly simple to get the value corresponding to a given key: after `array set local` {*keyval list*}, you can access the value with key `$key` as `$local($key)`. It is also simple to make modifications when the data is stored in that format: after `set local($key) $newval`, an `array get\ local` will return a modified {*keyval list*} (note however that this may return the pairs in a different order than before). Using `array get` and `array set` in this way is not significantly slower than `lreplace` on a list of only the values (it might even be faster in some cases) but it is much easier to program. The keyval list also has the important advantage of being a much more flexible data structure, since each item (key–value pair) is independent of the others (whereas in a list the index of an item depends on how many other items there are before it), hence items can be added or removed without requiring much changes to existing code.

Returning to the subject of `dialog::make_paged`, the structure of the {*item list*} still remains to be explained. Each item in this list corresponds to one dialog item of the page in question. The items are themselves lists with the structure

{*key*} {*type*} {*name*} {*help*}[?]

where the {*key*} identifies the value in the {*keyval list*} that should be used for this item. The same {*key*} can be used for any number of items as long as they are on different pages. Thus if `make_paged` is used instead in the above example, the command could be

```
dialog::make_paged\
  {First {a 1 b 0 c ""} {{a var Hey} {b flag blah} {c folder hey}}}\
  {Second {c 2} {{c var Hey}}}
```

and the return value if no item is edited would be

```
First {a 1 b 0 c {}} Second {c 2}
```

whereas the same editing as before would produce

```
First {a Hay b 1 c {}} Second {c {hey hey hey}}
```

An obvious question now is of course which of the `dialog::make` and `dialog::make_paged` procedures one should choose for each specific task. The answer is that this depends mainly on how items on different pages are related to each other. If each page is a unit of its own then `make_paged` is preferable, but if items on different pages are no less related than items on the same page then `make` should work just as well. The `editGroup` procedure, whose implementation can be found in Subsection 3.4, gives the canonical example of the former situation. For single page dialogs, where the above rule gives no guidance, one should rather look at what happens to the item values immediately before and after the call. If they are simply fetched from some variable and then stored back into it then `make_paged` is probably a more convenient choice, but if you need to pre- or postprocess the item values then `make` probably has less overhead. Dialogs with many uneditable items (such as those produced by the Get Info commands in the Mac Menu) or with only a few values altogether are probably easier to create using `make`.

`dialog::editGroup` (proc)

1.1 Dialog item types

Most {*type*}s consist simply of a single word; these are called *simple* types. All types in the above example are simple. The currently defined simple types are

`appspec` An application specifier, for use with e.g. `exec`, `launch`, or AppleEvent commands (depending on platform). The value is viewed as the file path of the application executable, but that is only one of the two forms that the value can take. If the value is six characters long and the first and last character both are apostrophes, then the four characters between them are interpreted as the Mac OS 'sig' (creator code) of the application. This latter format is preferred when it can be used. It could also be that more formats will have to be added if support for the Tk commands `send` and/or `dde` (both of which are very non-Mac OS) is needed.

At the time of writing, there is no direct support for application specifiers in other parts of AlphaTcl, but the API stuff [2] by Frédéric Boulanger will provide this support. If you do not use that, will have to do some converting before you can use the value of an `appspec`.

`binding` A key binding. It is viewed as plain text, e.g. ‘Cmd-Opt-L’, but the format is the one used to put key bindings in menus. Use `keys::toBind` to turn the value into something suitable for the `Bind` command.

`colour` A popup menu from which you can choose amongst the named colours that are defined (`blue`, `green`, etc.).

`date` A date and time of day. This is viewed and entered in a human-readable ‘short date format’, but the value of the item is in seconds relative to an “epoch” that depends on what version of *Alpha* or *Alphatk* you are using, just as is the case with the value returned by e.g. the `now` command.

It has been suggested that these values should instead be in ISO 8601 format, i.e.,

$$\langle yyyy\rangle\langle mm\rangle\langle dd\rangle\mathtt{T}\langle HH\rangle\langle MM\rangle\langle SS\rangle$$

where

$\langle yyyy\rangle$	is the year AD (four digits),
$\langle mm\rangle$	is the month (01–12),
$\langle dd\rangle$	is the day of month (01–31),
$\langle HH\rangle$	is the hour (00–23),
$\langle MM\rangle$	is the minute (00–59), and
$\langle SS\rangle$	is the second (00–59).

This has the important advantage of being decipherable without the assistance of Tcl. It is also independent of which the current epoch is, which could help avoiding some Y2K-type errors.

`file` The file path of an existing file.

`flag` Simple checkbox. Can assume the values `0` (not checked) and `1` (checked).

`folder` The path to an existing folder.

`mode` A popup menu from which you can choose amongst the installed modes, with names given as in the mode menu on the status bar. There is also a `<none>` item in the menu.

`modeset` A list (or set) of modes, with the same name format as for the `mode` type (except that there isn’t a `<none>` mode). The value is viewed as a list and edited in a multi-choice listpick dialog.

`password` An editable text string, but shown in a box that is too small for anyone to see what is typed. Meant for passwords and similar material that shouldn’t be shown on the screen.

Note: As a precaution, the text that is in this box when the dialog is opened is not the actual value. Thus you cannot edit this value, you can only retype it.

`searchpath` A list of folders, each of which can be added, removed, or changed independently of the others.

`static` The value is simply shown, but cannot be edited. Useful for informative purposes. There is no result from this kind of item.

`text` The name is shown, but the value is ignored. Could be used as a subheading in a dialog page. There is no result from this kind of item.

`thepage` An item of this type is not shown in the dialog and its initial value is ignored, but it returns the name of the page that was current when the dialog was closed. (That is significant in e.g. the standard installation dialog.)

`url` An universal resource locator (URL). You can type it in explicitly, pick a local file, or use the frontmost page in your browser.

`var` Editable text string.

`var2` Editable text string, whose box is two lines tall.

In general, a {*type*} is a list whose first element serves as type identifier (selecting which code should make the item) whereas the other elements contain additional data needed to completely specify the type. In addition to the above simple types, there are also a couple of complex types, as listed below.

`menu` A popup menu. The format of this {*type*} is

`menu` {*item list*}

where the {*item list*} is the list of items to put in the menu. The value will be one of the elements in the {*item list*}.

`menuindex` A popup menu. The format of this {*type*} is

`menuindex` {*item list*}

where the {*item list*} is the list of items to put in the menu. The value will be an *index* into the {*item list*}.

`multiflag` A group of checkboxes. The format of this {*type*} is

`multiflag` {*checkbox title list*}

where the {*checkbox title list*} gives the titles given to the individual checkboxes. The value of this item is a *list*, with the same number of items as the {*checkbox title list*}, and in which each element is either a `0` or a `1`. The name of the `multiflag` item is put as a heading above the group of checkboxes, which are placed in two columns.

`subset` A subset of a given set, which is chosen in a multichoice listpick dialog. The format of this {*type*} is

`subset` {*item list*}

where the {*item list*} is the list of items to show in the listpick dialog. The value will be a sublist (which can be empty) of the {*item list*}.

There are also two other complex types `global` and `hidden` defined, but those are kind of special and to not contribute anything to what the user can see.

Some effort has been put into ensuring that additional types can be defined without procedure redefinitions. See Subsection 2.4 for details and examples.

1.2 Dialog command options

What remains to be explained about the `make` and `make_paged` procedures is their
`-defaultpage` option ⟨*option*⟩s. The `-defaultpage` option has the syntax

`-defaultpage` {*page name*}

It specifies on which page the dialog should open. If the option is omitted then the dialog
`-title` option opens on the first page. The `-title` option sets the title of the dialog window; it has the syntax

`-title` {*dialog title*}

This option has no effect in *Alpha 7*, where the dialog window has no title.[1]

`-width` option The `-width` option sets the width of the dialog window (the height is determined automatically and depends on the height of the dialog items). The syntax is

`-width` {*dialog width*}

`-ok` option where {*dialog width*} is in screen pixels. The default value is `400`. The `-ok` and `-cancel`
`-cancel` option options can be used to set the names on the OK and Cancel buttons. The syntaxes are

`-ok` {*name of ok button*}
`-cancel` {*name of cancel button*}

`-addbuttons` option The most complex option is the `-addbuttons` option, which adds buttons other than the default OK and Cancel buttons to the dialog. The value for this option is a "button list", which has the structure

$$\big(\{name\}\ \{help\}\ \{script\}\big)^{+}$$

where each triple {*name*} {*help*} {*script*} describes one additional button. {*name*} is the button name, i.e., the text that will be shown on the button. The button will be made wide enough to contain the whole {*name*}. {*help*} is the help text for the button. {*script*} is a script that is evaluated when the button is clicked. See below for the basic details on the context in which button scripts are evaluated. If some button script does not work as
`-debug` option expected then it might help use the `-debug` option. This has the syntax

`-debug` {*debug level*}

where {*debug level*} is an integer. The default is to use debug level `0`. Currently the only other debug level is `1`: this causes the actual script, the error, and the `$errorInfo` to be printed using `tclLog` when a script terminates with an error.

Among the things button scripts can do is adding or removing pages from the dialog (as it is shown to the user). In `make` the effect is simply that some pages are hidden. Since this is most often useful if the dialog opens in a state where some pages are hidden, there
`-hidepages` option is an option `-hidepages` that hides one or several pages. The syntax is

`-hidepages` {*page list*}

[1] It might be observed that it is often possible to use the page name as a "title" for a dialog; hence the loss is probably not that significant.

where the {*page list*} is a list of names of pages. It makes no difference to the caller whether a page is hidden or not, since the code that compiles the return value only looks at the {*page*} arguments to `make`. The situation is different in `make_paged`, since that has a more "what you see is what you get" approach to pages: a hidden page would not be included in the return value and thus it effectively would not exist.

`-changedpages` option
`-changeditems` option

The `-changedpages` and `-changeditems` options of `make_paged` can be used by the caller to request information about on which pages some value was changed and which items had their values changed, respectively. The syntaxes are

`-changedpages` {*var-name*}
`-changeditems` {*var-name*}

With `-changedpages`, the {*var-name*} variable is set to a list of the names of pages on which some item value was changed. With `-changeditems`, the {*var-name*} variable is set to a list with the structure

$$\big(\{\textit{page name}\}\ \{\textit{key list}\}\big)^{?}$$

Here, for each page where the value of some item has been changed, the keys for those items are listed in the {*key list*}.

1.3 Button scripts

Button scripts are evaluated in the local context of the `make` or `make_paged` procedure (depending on which you called). They do a lot of their work by modifying local variables in these procedures and hence you should familiarize yourself with the actual implementations in Subsection 2.5 if you are going to write anything but the simplest button scripts. Some of the basic principles can however be outlined.

First of all, the button scripts are not evaluated while the actual dialog window is open. Instead the dialog window is closed when the button is clicked, the item values are then stored in an array, the button script is evaluated, and finally the dialog is rebuilt and the dialog window is reopened, waiting for the user to do something else. This means that you will not have to worry about any lower level descriptions of the dialog than that used in the call to `make` or `make_paged`, since there is no such thing at the time a button script is evaluated. A button script that needs to *logically* close the dialog, i.e., cause `make` or `make_paged` to return, should do this by setting the `retCode` variable (this is in fact how the OK and Cancel buttons are implemented). The value of `retCode` will become the `-code` argument of `return`, so `0` means normal return and `1` means an error. For normal returns, the return value is constructed as usual, but for other types of returns it is the responsibility of the button script to construct a return value and store it in the `retVal` variable. As an example, the Cancel button is handled by a button script that simply does

`retCode` (var.)
`retVal` (var.)

```
set retCode 1
set retVal "cancel"
```

The `make` and `make_paged` procedures keep most of their data in arrays and most of these have one entry per item. The indices into these arrays have the form

⟨*page name*⟩ , ⟨*item name*⟩

(you should be aware that these indices often contain spaces). Of particular interest is the array that contains the item values. For technical reasons that is a global array which should only be accessed using special procedures. To get the value of an item you should use the `valGet` procedure and to change it you should use the `valChanged` procedure. The syntaxes of these are

`dialog::valGet` (proc)
`dialog::valChanged` (proc)

`dialog::valGet` {*dialog ref.*} {*index*}
`dialog::valChanged` {*dialog ref.*} {*index*} {*value*}

The {*dialog ref.*} is a reference to the current dialog; the `make` and `make_paged` procedures keep their value for this in the `dial` variable. The {*index*} is ⟨*page name*⟩,⟨*item name*⟩ as described above. The {*value*} is the new value for the item and `valGet` returns the current value.

`dial` (var.)

Another thing that button scripts can do is hide or show individual items. Technically that is done by changing their type to and from the following complex type

`hidden` An item which isn't shown and whose value does not change, but which still returns a value. The format of this {*type*} is

`hidden` {*anything*}$^+$

where the {*anything*} is completely ignored.

The idea here is that any type of item can be hidden by prepending a `hidden` to the {*type*} of that item, and that removing the `hidden` will return it to the original type. There are two procedures `hide_item` and `show_item` which do precisely that. Their basic syntaxes are

`dialog::hide_item` (proc)
`dialog::show_item` (proc)

`dialog::hide_item` {*page*} {*name*}
`dialog::show_item` {*page*} {*name*}

(They do take an extra optional argument which might be needed if they are not called from the local context of the `make` or `make_paged` procedures.)

Examples of button scripts and how they can be used can be found in Subsection 3.3.

1.4 Preferences and dialogs

Historically there is a strong connection between dialogs for editing values and preferences in AlphaTcl, and most values one might want to edit this way are still preferences. Hence it is convenient to have a procedure which determines the dialog item type that corresponds to a preference. This is what the `dialog::prefItemType` procedure is for. It has the call syntax

`dialog::prefItemType` (proc)

`dialog::prefItemType` {*preference name*}

and returns a valid {*type*} for the preference.

Note: `prefItemType` does not yet handle all preference types. Contributions of code that lets it handle additional types are appreciated.

1.5 The width of dialog text

The built-in dialog commands of *Alpha* are different from most other commands in that they require you to know the *width* in screen pixels of most text strings you use. `dialog::make` handles most of that internally, but there are some restrictions you should keep in mind:

- Item names should fit on a single line in the dialog. The names of `text` and `multiflag` items are exceptions from this, as they will be broken on several lines if necessary. The names of items that have Set... buttons should be short enough to leave adequate room for this button.
- Page names should preferably fit on a half dialog line.
- Button names should fit on a single line, but will probably look ridiculous already if their width is half that of a dialog line.

You don't generally need to be concerned about the width of values however, as the displayed forms of most values are automatically abbreviated to fit on one line. (This happens especially often to file names.)

`dialog::text_width` (proc)

The `dialog::text_width` procedure is what `dialog::make` uses to actually determine the width of a string. It has the syntax

`dialog::text_width` {*string*}

and returns (an upper bound on) the width of the {*string*}. In *Alphatk* this procedure is implemented using `font measure` and returns the exact width. In *Alpha* the procedure computes the width based on the width table for characters in the Chicago font at 12 pt; this gives a valid upper bound also if Charcoal is used as system font. No notice is taken of kerning, but there doesn't seem to be any in these fonts. Only the width of characters in the MacRoman encoding is known to the *Alpha* `dialog::text_width` procedure; this might become a problem in *Alpha 8*, but the table of character widths is easily extended.

`dialog::width_linebreak` (proc)

For pieces of text that can be expected to be more than one line long, there is the `width_linebreak` procedure. It takes a string and a width (in pixels) limit as arguments, breaks the string into lines in such a way that no line is wider than the specified limit, and returns the list of lines that the string was broken into. The syntax is

`dialog::width_linebreak` {*string*} {*width*}

The linefeed (`\n`) and carriage return (`\r`) characters are given special treatment: a linefeed forces a linebreak at that position, whereas a carriage return separates two paragraphs. A paragraph separator is marked in the return value by a line only containing a carriage return. Spaces and tabs are discarded around linebreaks.

`dialog::width_abbrev` (proc)

There is also a `dialog::width_abbrev` procedure which, if necessary, replaces part of a string by an ellipsis character '...' so that the width of the resulting string does not exceed a given bound. The syntax is

`dialog::width_abbrev` {*string*} {*width*} {*ratio*}?

and the returned value is the abbreviated string. {*string*} is the string to abbreviate, {*width*} is the maximal width that the result may have, and {*ratio*} is a real number in the interval $[0,1]$ which controls where in the string the abbreviation will take place.

`dialog::ellipsis` (var.)

Finally, the actual string used for an ellipsis character by the procedures in this file is stored in the `dialog::ellipsis` variable. The initialization of this variable should be correct both for *Alphatk* and *Alpha* with a MacRoman character set, but it might need to be modified if some other character set is used. This can then be done in the *Alpha* `prefs.tcl` file.

2 Implementation

The code below lives in the `dialog` namespace.

```
1 ⟨*core⟩
2 namespace eval dialog {}
```

There are a few docstrip guards[2] that distinguishes certain parts of the code below. Their meanings are as follows:

core Main guard around code for the AlphaTcl core.

notinstalled This guards things that is useful when testing the code, but shouldn't be included in a version that is installed as part of AlphaTcl. Typical contents are `auto_load` commands to ensure that definitions here are not overwritten by some file that *Alpha* sources automatically, and hacks of procedures defined elsewhere.

log1 This guards some code that logs what is happening using the terminal package. Mainly useful while debugging.[3]

examples Surrounds some code examples.

The sooner the `auto_load` is done the better, so here it is.

```
3 ⟨notinstalled⟩auto_load dialog::make
```

2.1 The `dialog` command

`dialog` (command)

The `dialog` command is probably one of the most complicated *Alpha* commands there are (and features are still being added to it!). The basic syntax is a simple

`dialog` ⟨*option*⟩$^+$

but the number of options is quite large and their natures are rather diverse. Most option forms add a control (push-button, checkbox, radio button, popup menu, or editable text box) to the dialog. Some options add some graphic material that is not a control, such as for example a piece of static text. The graphic elements in a dialog are called *atoms* in this paper.

[2] See [4] or [5] for an explanation of this concept.

[3] One advantage of the docstrip format is that you never really have to remove such code from the sources. If it's just in a suitable module then docstrip won't include it.

The `dialog` command returns the list of values that the controls had when the dialog was closed. The values appear in this list in the same order as the corresponding options did in the argument list of `dialog`. Warning: In *Alpha 7*, there is a bug in how `dialog` quotes items. If some value contains an unmatched left or right brace, or ends with a backslash, then the result of `dialog` is probably not a valid Tcl list.

All the atom-generating ⟨*option*⟩s for `dialog` end with four arguments {*left*}, {*top*}, {*right*}, and {*bottom*}: these specify the *rectangle* associated with the atom. If nothing further is said then this rectangle can be understood to be the bounding rectangle of the atom. The coordinates are all integers, the unit is screen pixels, *x*-coordinates ({*left*} and {*right*}) increase while going to the right, and *y*-coordinates ({*top*} and {*bottom*}) increase while going *down*. The background rectangle of the dialog window has its upper left corner at the point $(-3,-3)$, but the negative coordinate pixels are technically part of the window frame and `dialog` does not draw anything there.

Inside Macintosh [3, p. 6:34] prescribes that atoms in a dialog should be separated by either 13 or 23 pixels of white space. Examples there suggest using 13 pixels for separation between atoms, as well as for the top, right, and bottom margins. The left margin is however 23 pixels. Bold frames (such as that around the default button) should not be included in these measurements. On the other hand, the 3 pixels wide white boarder that the dialog manager itself adds on each side of a modal dialog (which is what the `dialog` command creates) and should be counted as part of the margin. The dialogs constructed in Subsection 2.4 below actually have vertical separation of only 7 pixels between the editable items in a dialog, as the 13 pixels prescribed by Inside Macintosh seems a bit much for the short pieces of text that they constitute. There's no particular reason for using exactly 7 pixels, though; it was picked pretty much at random. Full-size buttons do however get a separation of 13 pixels.

In *Alphatk* and *Alpha 8*, some atom-generating options take suboptions which can be used to further specify the behaviour of the atom. These are then placed immediately before the {*left*} argument of the atom. *Alpha 7* does not understand these, and hence one should only include them if one has checked what program AlphaTcl is being run on.

2.1.1 Basic `dialog` options

`-w` option
`-h` option

The `-w` and `-h` options set the width and height respectively of the dialog window. Their syntaxes are

`-w` {*width*}
`-h` {*height*}

where {*width*} and {*height*} are in screen pixels. The Toolbox automatically adds a three pixels wide white border on all sides around the {*width*} by {*height*} rectangle specified using these options, but that area cannot be drawn in.

`-b` option

The `-b` option creates a push-button (usually simply called button). It has the syntax

`-b` {*title*} (`-set` {*callback*})$^{?}$ {*left*} {*top*} {*right*} {*bottom*}

but the `-set` suboption is not implemented in *Alpha 7*. Without the `-set` suboption, the button has one value which is either `0` (button was not clicked) or `1` (button was clicked).

As clicking a button closes the dialog, there can be at most one button in the dialog which has value `1`. Conversely, every dialog must contain at least one button, as the only way to close the dialog is to click a button. The first button to be defined will be the *default* button: it has a double frame and pressing the Return or Enter key will be equivalent to clicking this button. If there is a button named 'Cancel' then pressing the Escape key will be equivalent to clicking that button.

The `-set` suboption is not supported in *Alpha 7*. Clicking a button with a such a suboption does not close the dialog, but tells *Alpha* to evaluate a script that is part of the {*callback*} (more on this below). The button still contributes a value (always `0`) to the result of `dialog` however.

Inside Macintosh [3] recommends the height 20 pixels for buttons. In AlphaTcl, there is a tradition of giving "minor" buttons a height of 15 pixels.

`-c` option

The `-c` option creates a checkbox control. It has the syntax

$$\texttt{-c}\ \{\textit{title}\}\ \{\textit{value}\}\ \left(\texttt{-font}\ \{\textit{font}\}\right)^{?}\ \{\textit{left}\}\ \{\textit{top}\}\ \{\textit{right}\}\ \{\textit{bottom}\}$$

The value of the checkbox is either `0` (not checked) or `1` (checked). The bounding rectangle encloses both the checkbox and its title. If several checkboxes are placed in a column then not only the {*left*}, but also the {*right*}, coordinates of all these buttons should coincide. This is due to localization issues.

The `-font` suboption is not supported in *Alpha 7*. The syntax for a {*font*} is unclear, current examples always use `2` for this.

`-t` option

The `-t` option creates a static text atom in the dialog. This option has the syntax

$$\texttt{-t}\ \{\textit{text}\}\ \left(\texttt{-dnd}\ \{\textit{dial}\}\ \{\textit{varinfo}\}\right)^{?}\ \{\textit{left}\}\ \{\textit{top}\}\ \{\textit{right}\}\ \{\textit{bottom}\}$$

The `-dnd` suboption (see below) gives drag-and-drop functionality to the text atom, but is not supported by *Alpha 7*. There is no control result from a `-t` atom.

If the measured width of the {*text*} is *right* – *left* pixels or more then it is broken on several lines (note that it needs *not* be strictly wider than the rectangle for this to happen) and set flush left. The height of one line of text is (with standard fonts) 15 pixels, of which 12 are above the baseline and 3 below. There is a 1 pixel space between two lines. The top of the first first line coincides with the top of the rectangle. In *Alpha*, the {*text*} may be at most 255 characters (this restriction exists for most options, but it is easiest encountered for `-t` items).

`-e` option

The `-e` option creates an editable text atom (TextEdit box) in the dialog. This option has the syntax

$$\texttt{-e}\ \{\textit{text}\}\ \{\textit{left}\}\ \{\textit{top}\}\ \{\textit{right}\}\ \{\textit{bottom}\}$$

where {*text*} is the default text to put in the box. The value of this control is the text that is in the box when the dialog closes.

The bounding rectangle of the box extends 3 pixels further in all directions than the item rectangle specifies, due to the frame around the box. The item rectangle corresponds instead to the text in the box—changing `-e` to `-t` will loose the editability and the frame, but leave the text in exactly the same position as long as it is not being edited. When the cursor is positioned in an `-e` atom box, the text is instead aligned with the *bottom* of the rectangle.

`-r` option

The `-r` option creates a radio button atom. It has the syntax

`-r` {*title*} {*value*} {*left*} {*top*} {*right*} {*bottom*}

all of which work just as for checkboxes. The difference is that clicking one radio button sets its value to `1` and the values of all other radio buttons *in the entire dialog* to `0`. Hence it is impossible to have more than one group of radio buttons in a dialog, and they aren't used in any of the standard dialogs.

`-p` option

The `-p` option has the syntax

`-p` {*left*} {*top*} {*right*} {*bottom*}

It used to create a "grey outline" (visual element which does not return any control value), but current versions of *Alpha* and *Alphatk* seems to ignore it.

`-m` option

The `-m` option creates a popup menu atom in the dialog. The syntax is

`-m` {*menu items*} {*left*} {*top*} {*right*} {*bottom*}

where {*menu items*} is a list with the format

{*default item*} {*menu item*}$^+$

The {*menu item*}s are the items shown in the menu. The {*default item*} is the item that will be the initial choice, provided that it equals one of the {*menu item*}s—otherwise the first {*menu item*} will be the initial choice. The control value returned is the chosen menu item. See the `-n` option for information about the relation between the dialog pages and popup menus.

The bounding rectangle for the popup menu atom extends one pixel to the left of {*left*}, one pixel above {*top*}, two pixels to the right of {*right*}, and 18 pixels below {*top*}, whereas {*bottom*} is ignored. Furthermore the bounding rectangle will not extend all the way to {*right*} unless there is some menu item which is that wide. Hence it is not feasible to line up the right edge of a menu with something, one can only prevent that it extends too far.

`-n` option

The `-n` option starts a new dialog page, so that all atoms after it (and before the next `-n` option, if there is another) will be put on a specific dialog page. The syntax is

`-n` {*page name*}

where the {*page name*} is primarily an internal identifier for the page. The `-n` option does not produce any control value. Options that appear before the first `-n` option will produce atoms which are visible on all pages of the dialog.

When there is an `-n` option, the popup menu from the first `-m` option will work as a page selector, so that the page for which atoms are currently shown is the one with the same name as the currently selected item in the first popup menu. Items in this menu that are not names of pages defined using `-n` will be treated as if they had been defined but don't contain any items. The dialog created by the `dialog::getAKey` procedure (defined in `dialogs.tcl`) makes a rather ingenious use of this fact.

2.1.2 New `dialog` options

Below are described some new `dialog` options that were first implemented on *Alphatk* and which *Alpha 7* neither supports nor understands. *Alpha 8* implements some of these, and should eventually support them all. The next two options are available both in *Alphatk* and *Alpha 8*.

`-T` option

The `-T` option sets a title for the dialog window. The syntax is

`-T` {*title*}

`-help` option

The `-help` option can be used to provide help texts for items in the dialog. The syntax is

`-help` {*help text list*}

where the {*help text list*} is a list of help texts.

There are also a couple of options which are currently only supported by *Alphatk*, although an *Alpha 8* implementation is probably not too far away. Only a few of them are used anywhere in AlphaTcl and many are "not yet officially supported".

`-l` option

The `-l` option creates a listpick atom in the dialog. The syntax is

`-l` {*value*} {*height*} (`-dnd` {*dial*} {*varinfo*})$^{?}$ {*left*} {*top*} {*right*} {*bottom*}

where {*value*} is the list of strings to show in the listpick. {*height*} is probably the height of the item, in rows. The `-dnd` suboption gives drag-and-drop functionality to the atom.

`-i` option

The `-i` option creates an image atom in the dialog, similarly to e.g. the icons in standard Mac OS alerts. The syntax is

`-i` {*image*} {*left*} {*top*} {*right*} {*bottom*}

where {*image*} is the name of a Tk image object to show in the dialog.

`-mt` option

The `-mt` option creates a popup menu with its own title in the dialog. The syntax is

`-mt` {*title*} {*menu items*} {*left*} {*top*} {*right*} {*bottom*}

where {*value*} is the title of the popup menu and the remaining arguments are handled identically to the `-m` option.

`-copyto` option

The `-copyto` option arranges for the value of the preceding dialog item to be copied and displayed in another, whenever the first changes. The dialog handled by the `prompt` command in *Alpha 7* hardcodes what can be achieved with this option. The syntax is

`-copyto` {*atom number*}

where {*atom number*} is a string containing either the number of the atom in the dialog (counting from zero) into which the value should be copied, or if {*atom number*} begins with a `+` or `-` then it is relative to the previous atom in the dialog (so either `+0` or `-0` would copy the value onto itself).

2.1.3 The drag-and-drop muddle

`-dnd` option

The `-dnd` suboption activates drag and drop functionality for a dialog atom. The general format for this suboption is:

`-dnd` {*dial*} {*varinfo*}

where {*varinfo*} in turn is a list with the format

{*varname*} {*type*}

When such an option is present, it has the effect that the atom we're currently creating (usually a `-t` atom) will accept drops. It would be reasonable to make it so that it could initiate drags as well, but that hasn't been examined yet.

The {*dial*} is simply a unique identifier for this dialog (so that all dialogs code is reentrant). It just needs to be passed along to appropriate routines later so you don't need to worry about it. The {*varname*} is the identifier for a specific item that e.g. the `valGet` and `valChanged` procedures take as argument along with {*dial*}. The {*type*}, finally, is the type of the entry (`folder`, `searchpath`, `file`, etc.). This is what decides which piece of code will control how the atom behaves with respect to dropping.

As for dialog controls in general, most of the details in dragging and dropping lies well outside the scope of what an AlphaTcl programmer needs to be concerned about. There are however two points of every drag-and-drop at which the mechanisms in the `dialog` command needs help from AlphaTcl, and for these must be provided two callbacks. The most obvious point is that of the actual drop—`dialog` has received a value from the GUI, but (in the case of a `-t` atom) has nothing to return it in—and therefore it instead immediately passes the value on to a callback. This way it is up to AlphaTcl to take care of the value and it usually does this by storing it in a suitable variable.

A less obvious, but no less important, point of interaction occurs when dragging. In general the user may be dragging around all sorts of things, but only a few may be suitable for dropping onto any given item. A piece of data is said to be *acceptable* for an item if it makes sense to drop it onto that item. It is part of the rules for drag-and-drop that the GUI must signal to the user when a drag passes over an item for which it would be acceptable, but the `dialog` command cannot test for acceptability without help. Therefore it relies on AlphaTcl to provide it with a callback that implements the relevant test.

One might expect[4] at this point that `dialog` should simply take these callbacks as arguments to the `-dnd` suboption and be done with it, but the mechanism actually implemented calls upon a number of AlphaTcl procedures with fixed names to *construct* the real callbacks! The drop callback is constructed as

`dialog::itemSet` {*update*} `{`⟨*base*⟩ {*dial*} {*varinfo*}`}` {*data*}*

whereas the acceptability callback is constructed as

`dialog::itemAcceptable` {*varinfo*} `{`{*data*}*`}`

Here {*update*} is some information the program uses to identify what atom should be updated (this is simply passed as an argument to `dialog::setControlValue`). The ⟨*base*⟩

`dialog::valGetDropAction` (proc)

is what `dialog::valGetDropAction` returns when called with {*varinfo*} as argument; it can be more than one word. {*dial*} and {*varinfo*} are taken from the arguments to `-dnd`, whereas the {*data*}s are the values that were dropped or are being dragged respectively.

`dialog::itemAcceptable` (proc)

The `itemAcceptable` procedure is fairly simple. The syntax is as shown above. The return value is an empty string if the thing being dragged is acceptable, or else a string that explains what is wrong with it (*Alphatk* shows these strings on the status bar). The current implementation performs tests if the {*type*} is `searchpath`, `file`, or `folder`, and accepts anything for all other types.

The `itemSet` procedure is much more obscure, but primarily it evaluates the command

⟨*base*⟩ {*dial*} {*varinfo*} {*data*}*

which (with the current `dialog::valGetDropAction`) is

`dialog::modifiedAdd` {*dial*} {*varinfo*} {*data*}*

when the {*type*} is `searchpath` and

`dialog::modifiedAdjust` {*dial*} {*varinfo*} {*data*}*

otherwise. With the exception for an extra round of checking the {*data*} using `itemAcceptable` and some messages, both `modifiedAdjust` and `modifiedAdd` boil down to

`dialog::modifiedAdjust` (proc)

`dialog::modifiedAdd` (proc)

`dialog::modified` {*dial*} {*varname*} {*newval*} {*type*}

where {*newval*} is the {*data*} in the case of `modifiedAdjust` and the concatenation of the old value with the {*data*} in the case of `modifiedAdd`. This simply means "update the variable in which the value of this dialog item is stored" and thus we've finally managed to accomplish one of the things that the drop should do. What remains is to change the text that is actually shown in the dialog, so that the user will see that the value has changed.

`dialog::itemSet` (proc)

That too is done in the call to `modified`, but only because the interpreter took the route via `itemSet` to get there! Each time `itemSet` is called, it first registers a hook under the name `dialog`, which the `dialog::modified` procedure tries to call whenever the {*type*} string is nonempty, and as its last action `itemSet` deregisters the hook. The combined effect is that the command

`dialog::setControlValue` {*update*} {*varname*} {*newval*} {*type*}

gets evaluated once for each drop. This command updates the value that is shown in the dialog, but not always correctly. This is mainly due to the distinction between item values as returned by e.g. `dialog::make` and item values as shown in a dialog window (this distinction is most obvious for `appspec`, `binding`, and `menuindex` items, but currently none of these have drag and drop functionality). Since `setControlValue` is called as a side-effect of storing the value that will be returned rather than as a conscious act by a callback selected for the particular type of item that is being updated, it only receives the former kind of value. The two kinds of values happen to be equal for those {*type*}s which currently have drag-and-drop, but not for any of the others.

`dialog::setControlValue` (command)

For the record, it should be remarked that the original idea with the `setControlValue` command was that it should change the value shown in an atom (which in the case of a `-t` atom means the text) so that the dialog should become as it would have been if that value had been used instead in the original call to `dialog`. To do that, it would only need the {*update*} and {*newval*} arguments, and in fact the other arguments are currently not used!

Related to this is the matter of adapting the value-as-shown to various physical restrictions imposed by the dialog itself. In particular file names and URLs are frequently wider than the dialog window and thus should somehow be compressed so that they will fit in the designated dialog atom. Since most of these restrictions are due to graphical properties of text that AlphaTcl only has vague concepts of, the ideal would be that the `dialog` command handled this on its own.[5] For *Alpha 7* one would of course instead have to explicitly abbreviate the value before it is given to `dialog`, and that is currently done in the generic dialogs by the `dialog::makeStaticValue` procedure, but right now that is done for *Alphatk* and *Alpha 8* as well. Automatic adaptation of a value-as-shown currently only happens in *Alphatk* to those that are set using `setControlValue`, and this uses yet another fixed callback (to the `dialog::abbreviate` procedure).

Is that all? No, but we're nearly there. It turns out that most GUIs insist on that all items that are dragged also have a type and that drop targets similarly must have a type. To determine the drag-and-drop type for an item, *Alphatk* calls the AlphaTcl procedure `valGetMimeType`, which has the syntax

`dialog::valGetMimeType` (proc)

`dialog::valGetMimeType` {*varinfo*}

and returns the wanted type. The current `valGetMimeType` returns `text/uri-list` when the {*type*} part of the {*varinfo*} is `file`, `folder`, `url`, or `searchpath` and an empty string in all other cases. As it happens, the empty string is not a valid type and therefore *Alphatk* ignores the `-dnd` suboption unless the {*type*} is one of these four.

`-set` option

Having sorted drag-and-drop out, one might as well do the `-set` suboption to `-b` as well, since that is quite similar. The syntax is

`-set` {*callback*}

where the {*callback*} is a two-element list with the structure

{*script*} {*atom number*}

The {*script*} is a script that is evaluated when the button is clicked. The {*atom number*} is as for the `-copyto` option, and specifies an atom whose value the {*script*} should be allowed to change. *Alphatk* does not provide for the {*script*} to change more than one atom, and it uses the same indirect method here as for drag-and-drop. The real callback is

`dialog::itemSet` {*update*} {*script*}

[4] I certainly would, but apparently Vince had other plans. /LH

[5] For really tough cases, such as a long URL or file name, it might be necessary to omit parts of the value. This is then best handled by a callback since what part is best to omit depends on the type of the value. Many such callbacks could probably be `dialog::width_abbrev` straight off.

(where {*update*} is computed from the {*atom number*}) and the {*script*} is supposed to call `dialog::modified` to update the item value both in memory and as shown in the dialog window.

2.2 Measuring text

`dialog::text_width` (proc)
`charwidth(`*character*`)`

The `dialog::text_width` procedure computes the width in screen pixels of the string it gets as argument.

```
4 if {${alpha::platform}=="alpha"} then {
```

In *Alpha*, the procedure uses the character widths stored in the `charwidth` array: `$charwidth(z)` is the width of the character `z`. The initial values in this array are for 12 point Chicago. The corresponding table for Charcoal is mostly the same, although some widths there would be smaller. No character is wider in Charcoal than in Chicago.

```
5    set code 0
6    foreach w {0 6 12 12 6 14 11 14 0 4 16 14 14 0 6 6 9 11 11 9 11 6 6\
         16 12 9 12 11 13 6 6 6 4 6 7 10 7 11 10 3 5 5 7 7 4 7 4 7 8 8 8 8\
         8 8 8 8 8 8 4 4 6 8 6 8 11 8 8 8 8 7 7 8 8 6 7 9 7 12 9 8 8 8 8 7\
         6 8 8 12 8 8 8 5 7 5 8 8 6 8 8 7 8 8 6 8 8 4 6 8 4 12 8 8 8 8 6 7\
        6 8 8 12 8 8 8 5 5 5 8 6 8 8 8 7 9 8 8 8 8 8 8 8 8 7 8 8 8 8 4 4 4\
        4 8 8 8 8 8 8 8 8 8 8 5 6 7 9 7 7 9 8 10 10 11 6 6 9 11 8 14 7 6 6\
        8 10 8 9 10 11 6 7 7 10 12 8 8 6 7 12 6 8 9 9 9 14 8 8 8 8 11 12 6\
         10 7 7 4 4 7 9 8 8 3 8 6 6 10 10 5 4 4 7 15 8 7 8 7 7 6 6 6 6 8 8\
                                          11 8 8 8 8 4 6 8 6 6 6 6 6 6 6 6} {
15      if {[info tclversion] < 8.1} then {
16         set charwidth([format %c $code]) $w
17      } else {
18         set charwidth([encoding convertfrom [format %c $code]]) $w
19      }
20      incr code
21   }
22   proc dialog::text_width {str} {
23      global charwidth
24      set w 0
25      foreach ch [split $str ""] {incr w $charwidth($ch)}
26      set w
27   }
28 } else {
```

In *Alphatk*, the procedure is instead implemented using the Tk command `font\ measure`. I'm not sure `system` is the right font in this case, though.

```
29   proc dialog::text_width {str} {font measure system $str}
30 }
```

`dialog::width_abbrev` (proc)

The `dialog::width_abbrev` abbreviates a string (such as for example a file name) until it fits within a specified width. The syntax is

`dialog::width_abbrev` {*string*} {*width*} {*ratio*}?

and the result is the abbreviated string. {*string*} is the string to abbreviate, {*width*} is the maximal width of the result, and {*ratio*} controls how much of the result should be from before or after the point of abbreviation. The default is `0.33`, which means twice as much is kept after the point of abbreviation as after it.

```
if {${alpha::platform} == "alpha"} then {
```

The implementation for *Alpha* uses the `charwidth` array.

```
  proc dialog::width_abbrev {str width {ratio 0.33}} {
     global charwidth dialog::ellipsis
     set w 0
     set tw [expr {$width - [dialog::text_width ${dialog::ellipsis}]}]
     set abbr ""
     set t [expr {$ratio * $tw}]
     foreach ch [split $str ""] {
        incr w $charwidth($ch)
        if {$w < $t} then {append abbr $ch}
     }
     if {$w <= $width} then {return $str}
     append abbr ${dialog::ellipsis}
     set t [expr {(1-$ratio) * $tw}]
     foreach ch [split $str ""] {
        if {$w < $t} then {append abbr $ch}
        incr w -$charwidth($ch)
     }
     set abbr
  }
} else {
```

The implementation for *Alphatk* uses instead the `font measure` command and a binary search.

```
  proc dialog::width_abbrev {str width {ratio 0.33}} {
     global dialog::ellipsis
     if {[font measure system $str] <= $width} then {return $str}
     set tw [expr {$width - [font measure system ${dialog::ellipsis}]}]
     set lower -1
     set upper [expr {[string length $str] - 1}]
     set t [expr {$ratio * $tw}]
     while {$upper - $lower > 1} {
        set middle [expr {($upper + $lower) / 2}]
        if {[font measure system [string range $str 0 $middle]] > $t}\
                     then {set upper $middle} else {set lower $middle}
     }
     set abbr [string range $str 0 $lower]
     append abbr ${dialog::ellipsis}
     set upper [string length $str]
     set t [expr {(1 - $ratio) * $tw}]
     while {$upper - $lower > 1} {
        set middle [expr {($upper + $lower) / 2}]
        if\
           {[font measure system [string range $str $middle end]] > $t}\
```

```
                                then {set lower $middle} else {set upper $middle}
72      }
73      append abbr [string range $str $upper end]
74   }
75}
```

`dialog::ellipsis` (var.) This variable stores the ellipsis (“three dots”) character used for showing that “this leads to another dialog”. Hopefully this might get around some platform-related problems. If you don’t like the automatic guess, you can set it in your prefs file.

```
76if {![info exists dialog::ellipsis]} then {
77   if {[info tclversion] >= 8.1} then {
78      set dialog::ellipsis \u2026
79   } else {
80      set dialog::ellipsis \xc9
81   }
82}
```

`dialog::width_linebreak` (proc) The `width_linebreak` procedure takes a string and breaks it into lines in such a way that no line is wider than a specified limit (unless there is a character that is wider than this limit). Then it returns the list of lines in the broken string. The syntax is

`dialog::width_linebreak` {*string*} {*width*}

where {*string*} is the string to break and {*width*} is the width limit for a line (no line may be that wide or wider).

It is possible that more arguments should be added to allow customisation of what is considered a permissible breakpoint. Currently a linefeed is interpreted as a forced breakpoint, a carriage return is interpreted as a paragraph separator, and spaces and tab characters are considered permissible breakpoints. Whitespace is discarded before and after a linebreak. A paragraph separator becomes a line consisting of one carriage return character.

```
83proc dialog::width_linebreak {str w} {
84   if {![string length $str]} then {return {}}
85   set res [list]
86   foreach s [split $str \r] {
87      lappend res \r
88      foreach s2 [split $s \n] {
89         eval [list lappend res]\
                              [dialog::width_linebreak2 [string trim $s2] $w]
91      }
92   }
93   lrange $res 1 end
94}
```

`dialog::width_linebreak2` (proc) The `width_linebreak2` procedure is what does most of the work for `width_linebreak`. It has the same syntax as that procedure, but linefeeds and carriage returns aren’t allowed in the input string.

```
95if {${alpha::platform} == "alpha"} then {
96   proc dialog::width_linebreak2 {str w} {
```

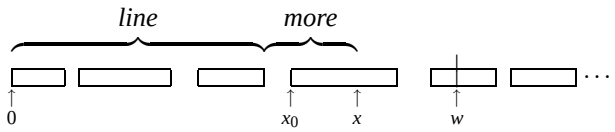

Figure 1: Variables in `dialog::width_linebreak2`

With *Alpha*, even determining the width of a string requires a loop over the characters of that string. Hence the most efficient implementation is to break the string into lines during such a loop, but then of course one must keep track of much more than the just the total width so far. Most of these are explained in Figure 1. Apart from these substrings of the argument string `str` and horizontal positions, the result is collected in `res` and the `was` variable kind of keeps track of the state: it is `1` if the last character was a whitespace character and `0` otherwise.

```
97      global charwidth
98      set res [list]
99      set line ""
100     set more ""
101     set x 0
102     set was 1
103     foreach ch [split $str ""] {
104        set is [expr {$ch==" " || $ch=="\t"}]
105        if {!$is && $was} then {
```

A new word has begun.

```
106           if {![string length $line]} then {
107              set more ""
108              set x 0
109           }
110           set x0 $x
111        } elseif {$is && !$was} then {
```

A word just ended.

```
112           append line $more
113           set more ""
114        }
115        set was $is
116        incr x $charwidth($ch)
117        if {$x>=$w} then {
```

Need to break the line before the current character.

```
118           if {[string length $line]} then {
```

Normal case: breaking at whitespace.

```
119              lappend res $line
120              set line ""
121              set more [string trimleft $more]
122              set x [expr {$x-$x0}]
```

The last set gives rise to a nice exercise: to prove that `x0` must have been set if the program enters this branch of the `if`.

```
123            } else {
```

Abnormal case: the current word is longer than a line. The break is put before the current character.

```
124               lappend res $more
125               set more ""
126               set x $charwidth($ch)
127            }
128            set x0 0
129         }
130         append more $ch
131      }
```

End of `foreach` loop. Now it only remains to include the last line (if there is one) in the result.

```
132      set line [string trim "$line$more"]
133      if {[string length $line]} then {lappend res $line}
134      return $res
135   }
136} else {
```

The *Alphatk* implementation is instead based on incrementally testing the possible breakpoints. It uses some Tcl 8 regexp features.

```
137   proc dialog::width_linebreak2 {str w} {
138      set res [list]
139      set idx -1
140      while\
          {[regexp -indices -start [expr {$idx+1}] -- {\S($|\s)} $str t]}\
                                                                        {
```

This loop steps through the ends of words, one by one.

```
141         if {$w >\
                [dialog::text_width [string range $str 0 [lindex $t 0]]]}\
                                                                  then {
143            set idx [lindex $t 0]
144         } elseif {$idx>=0} then {
```

When an end of a word position which is too far away to fit on the current line, a break is taken at the previous end of a word.

```
145            lappend res [string range $str 0 $idx]
146            set str\
                       [string trim [string range $str [expr {$idx+1}] end]]
147            set idx -1
148         } else {
```

Except for the case when a single word is wider than a line. In this case, the maximal breakpoint is found using an interval search.

```
149            set upper [lindex $t 0]
150            set lower 0
```

```
151             while {$upper-$lower>1} {
152                set middle [expr {($upper+$lower)/2}]
153                if {$w >\
                         [dialog::text_width [string range $str 0 $middle]]}\
                            then {set lower $middle} else {set upper $middle}
156             }
157             lappend res [string range $str 0 $lower]
158             set str\
                      [string trim [string range $str [expr {$lower+1}] end]]
159             set idx -1
160          }
161       }
```

End of loop over the words.

```
162       if {$idx>=0} then {lappend res [string range $str 0 $idx]}
163       return $res
164    }
165 }
```

2.3 Storing and updating values in dialogs

The procedures in this subsection used to be in `dialogUtils.tcl`, so we need to make sure that that is sourced before the new definitions are given.

166 ⟨notinstalled⟩*auto_load dialog::flag*

The dialog procedures keep the values of items in a global array, so that they can be accessed by callback scripts that are evaluated in the global context. (This happens for example for the `bind` scripts that *Alphatk* uses.) Each dialog managing procedure must allocate one of these arrays before doing any interaction with the user, and then deallocate it when it's done. The reason for this set-up is that (i) the dialog procedures should be reentrant and (ii) the values would be impossible to access for some pieces of code if they weren't kept in the global scope.

`dialog::tcldial`⟨*num*⟩ (array)
`dialog::changed_tcldial`⟨*num*⟩ (var.)
`dialog::globalCount` (var.)

Global arrays named `dialog::tcldial`⟨*num*⟩, where ⟨*num*⟩ is an integer, are allocated for dialogs to store values in. Each such array is accompanied by a list named `dialog::changed_tcldial`⟨*num*⟩ in which is stored the names of all elements in the array which have been explicitly changed. The `dialog::globalCount` variable stores the number of the most recently allocated `dialog::tcldial`⟨*num*⟩ array.

```
167 ensureset dialog::globalCount 0
```

`dialog::create` (proc)
`dialog::cleanup` (proc)

The `create` procedure allocates a new array to store dialog values in. It takes no arguments and return a reference string that should be used to access the array. The `cleanup` procedure takes a reference string as argument and deallocates the corresponding array.

```
168 proc dialog::create {} {
169     global dialog::globalCount
170     incr dialog::globalCount
171     upvar #0 "dialog::changed_tcldial${dialog::globalCount}" chvar
172     set chvar [list]
173     return "tcldial${dialog::globalCount}"
```

```
}

proc dialog::cleanup {mod} {
    global dialog::${mod} dialog::changed_${mod}
    if {[info exists dialog::${mod}]} {
        unset dialog::${mod}
    }
    if {[info exists dialog::changed_${mod}]} {
        unset dialog::changed_${mod}
    }
}
```

The identifier returned by `create` will have to be communicated to all procedures that access item values.

`dialog::valGet` (proc)
`dialog::valSet` (proc)
`dialog::valExists` (proc)

Basic access to the arrays for storing dialog values should be via the `valGet`, `valSet`, and `valExists` procedures. Their respective syntaxes are

`dialog::valGet` {*dialog*} {*name*}
`dialog::valSet` {*dialog*} {*name*} {*value*}
`dialog::valExists` {*dialog*} {*name*}

where {*dialog*} is a reference string returned by the `create` procedure and {*name*} specifies the item. `valGet` returns the value of the item. `valSet` sets the item value to {*value*} without marking the item as changed and doesn't return anything particular. `valExists` returns `1` if the item has been set and `0` otherwise.

```
proc dialog::valGet {mod name} {
    uplevel #0 [list set dialog::${mod}($name)]
}
proc dialog::valSet {mod name val} {
    uplevel #0 [list set dialog::${mod}($name) $val]
}
proc dialog::valExists {mod name} {
    uplevel #0 [list info exists dialog::${mod}($name)]
}
```

The {*name*} is usually formed as ⟨*page*⟩ , ⟨*item title*⟩ so that items on different pages can share the same title; there are cases in which each item title is reused on every page of a dialog.

`dialog::valChanged` (proc)

The `valChanged` procedure has the same syntax as `valSet`, but if the new value is different from the old then it additionally includes the item in the list of items whose names have been changed.

```
proc dialog::valChanged {mod name val} {
   global dialog::${mod} dialog::changed_${mod}
   if {$val != [set dialog::${mod}($name)]} then {
      set dialog::${mod}($name) $val
      lunion dialog::changed_${mod} $name
   }
}
```

`dialog::modified` (proc) The `modified` procedure is like `valChanged`, but it can also call a hook to make sure various GUI details are updated accordingly. This is mainly used by the `dialog::specialSet::`⟨*type*⟩ procedures.

```
⟨∗notinstalled⟩
proc dialog::modified {mod name val {type ""}} {
    dialog::valChanged $mod $name $val
    if {[string length $type]} {
```

We have some code registered which would like to know what changed. *Alphatk* uses such hooks to update dialog items from Set... buttons automatically, but it would be better if the code that called `dialog::modified` could do that explicitly.

```
        hook::callAll dialog modified $name $val $type
    }
}
⟨/notinstalled⟩
```

`dialog::changed_items` (proc) The `changed_items` procedure returns the current list of items whose values have been changed. The syntax is

`dialog::changed_items` {*dialog*}

where {*dialog*} is a reference string returned by `create`.

```
proc dialog::changed_items {mod} {
    uplevel #0 [list set dialog::changed_${mod}]
}
```

2.4 Building and handling dialog material

`dialog::handle` (proc) The `handle` procedure provides the glue between the built-in `dialog` command and the item-oriented interface to the dialog procedures. Its basic job is to open a new single/multipage dialog with specified items, handle user modifications of those items, and then return when the user presses a non-item button. Item definitions are taken from arrays in the caller's local context. Item values are taken from and then stored in a global array accessed using `valGet` and `valChanged`.

The syntax is

`dialog::handle` {*pages*} {*type-var*} {*dialog id*} {*help-var*} {*current-page-var*} {*option list*} {*button group*}$^+$

and the returned value is a string that depends on which button was pressed to end the dialog. The {*pages*} argument is a list with the structure

$\big(\{$*page name*$\}$ $\{$*item name list*$\}\big)^+$

which selects what items to show in the dialog. Each {*page name*} creates a new page with that name. The {*item name list*} contains the names of the items which will be shown on that page. Note that the page may contain more items than those specified in this list; those will then be ignored. This is useful in cases where some higher level setting has rendered some of the items irrelevant.

The {*type-var*} and {*help-var*} arguments are the names of arrays in the caller's local context, which are expected to contain the types and help texts (if there are any) respectively for the items in the dialog. These indices into these arrays have the form ⟨*page name*⟩ , ⟨*item name*⟩. The {*dialog id*} is an identifier to use with `valGet` and `valChanged` to access the values of items. The {*current-page-var*} argument is the name of a variable in the caller's local context. If, upon entry, this variable is set to the name of a page in the dialog then that will be the default page of the dialog. Upon return, this variable is set to the name of the current page.

The {*option list*} is a key–value list of extra options for the `dialog::handle` procedure. Unknown options are ignored and no option is mandatory. Currently the following options are recognized:

`-title` Title for the dialog window; by default an empty string. This is ignored in *Alpha 7*.

`-width` Width of the dialog window, in pixels; this defaults to 400.

A {*button group*}, finally, is a list with the structure

{*button list*} {*option*}*

and the {*button list*}, in turn, has the structure

({*title*} {*help*} {*return*})+

Each triple in the {*button list*} describes one button. The {*title*} is the button title, the {*help*} is the button help text, and the {*return*} is the value that `dialog::handle` will return when this button is pressed. An {*option*} can be anything; currently the following are understood:

`right` Put buttons in this group flush right (default is flush left).

`first` Put the buttons in this group first in the dialog material. This makes one of them the default button.

The procedure starts by making various global variables available and parsing some easy arguments.

```
211 proc dialog::handle {pages typevar dial helpvar pagevar optionL args} {
212    global dialog::indentsame dialog::indentnext dialog::simple_type\
                                          dialog::complex_type alpha::platform
214    upvar 1 $typevar typeA $helpvar helpA $pagevar currentpage
215    if {![info exists currentpage]} then {
216       set currentpage [lindex $pages 0]
217    }
218    set opts(-title) ""
219    set opts(-width) 400
220    array set opts $optionL
```

Next comes a loop which is needed since *Alpha 7* uses post-processing scripts to process item buttons. The loop will eventually be removed.

```
221    while {1} {
```

Now the dialog material can be constructed. This makes up most of `dialog::handle`. The dialog material is collected in the `res` variable, which will be a partial list of arguments to pass to the `dialog` command. Material is generally collected top to bottom, so that it is sufficient to know the bottommost position of an item to avoid putting two items on top of one another. The `y` variable generally says where the next item may be put. The `ymax` variable stores the maximal `y` value reached on any page processed so far. The `left` and `right` variables store the *x*-coordinates for the left and right respectively margin for dialog material.

```
222       set res [list]
223       set ymax 4
224       set left 20
225       set right [expr {$opts(-width) - 10}]
```

`multipage` is a flag which is `1` if a multipage dialog is being built and `0` otherwise. `pagemenu` is a list that will be used for the page menu in a multipage dialog. `helpL` is a list of help messages for the dialog items and `postprocL` is a list of *post-processing* scripts for the dialog items. More on that below.

```
226       set multipage [expr {[llength $pages] > 2}]
227       set pagemenu [list $currentpage]
228       set helpL [list]
229       set postprocL [list]
```

The outermost loop when constructing dialog material is over the pages. In multipage dialogs, an `-n` {*page name*} atom appears in the material to start each new page. Another difference is that there is a popup menu (19 pixels tall) at the top if a multipage dialog, but only a static text (15 pixels tall) at the top of a single page dialog. Hence `y` starts at different values.

```
230       newforeach {page items} $pages {
231 ⟨log1⟩         terminal::print_word emptyline "Page: $page" newline
232          if {$multipage} then {
233             lappend res -n $page
234             lappend pagemenu $page
235             set y 42
236          } else {
237             set y 38
238          }
```

The inner loop in material construction is over the items. Since material construction is a *very* diverse activity, and since it should be easy to add definitions of new types, the actual construction is handled by a legion of *construction scripts* that are selected according to the type of the item. These scripts access a number of `dialog::handle` variables, which are described in Subsubsection 2.4.1 below.

```
239          foreach name $items {
240 ⟨log1⟩           terminal::print_word newline " Item: $name" newline
241             set type $typeA($page,$name)
242 ⟨log1⟩           terminal::print_word newline "  Type: $type" newline
243             set val [dialog::valGet $dial $page,$name]
244 ⟨log1⟩           terminal::print_word newline "  Value: $val" newline
245             set help {}
```

```
246             catch {set help $helpA($page,$name)}
247 ⟨log1⟩           terminal::print_word newline "  Help: $help" newline
248             set script [list dialog::valChanged $dial $page,$name]
249             append script { [lindex $res $count]}
250             set visible 1
```

The following `while` loop exists to allow construction scripts to restart the construction of an item using the construction script for another type. Currently only the `global` type makes use of this. Normally the `break` is evaluated on the first iteration of the loop.

```
251             while {1} {
252                if {[llength $type] == 1} then {
253                   if {![info exists dialog::simple_type($type)]} then\
                                                              {set type var}
255                   eval [set dialog::simple_type($type)]
256                } elseif\
                      {[info exists dialog::complex_type([lindex $type 0])]}\
                                                                   then {
258                   eval [set dialog::complex_type([lindex $type 0])]
259                } else {
260                   dialog::cleanup $dial
261                   error "Unsupported item type '$type'"
262                }
263                break
264             }
```

The bulk of work done by the construction script is to append material to `res` and increment `y` by the height of that, but they may also set the `script` and `help` variables.

```
265             if {$visible} then {
266                incr y 7
267                lappend helpL $help
268             }
269 ⟨log1⟩           terminal::print_word newline "  Script: $script" newline
270             lappend postprocL $script
271          }
272          if {$y > $ymax} {set ymax $y}
273       }
274       incr ymax 6
```

This ends the loops over items and pages, respectively, and now all item-related material is in `res`! The `ymax` variable incremented to get full separation before the buttons (the construction of which comes next on the agenda).

Since the buttons should appear on every page of the dialog, their atoms must appear before all the material currently in `res`. Therefore dialog material for buttons is collected in a separate variable `button` which will then be concatenated with `res`. The button-building routines also make use of the `button_help` and `button_press` variables, in which the help texts and return values (when the button has been pressed) respectively are stored. The `l` and `r` variables contain the minimal and maximal *x*-coordinate that is available for button placement without increasing `ymax`; these are managed completely by `dialog::makeSomeButtons`.

```
275       set buttons [list]
```

```
276       set button_help [list]
277       set button_press [list]
278       set l $left
279       set r $right
280       foreach group $args {
281          set b_names [list]
282          set b_help [list]
283          set b_press [list]
284          newforeach {name help val} [lindex $group 0] {
285             lappend b_names $name
286             lappend b_help $help
287             lappend b_press $val
288          }
289 ⟨*log1⟩
290          terminal::print_word emptyline "Buttons:" newline
291          terminal::print_word newline " Names: $b_names" newline
292          terminal::print_word newline " Helps: $b_help" newline
293          terminal::print_word newline " Values: $b_press" newline
294 ⟨/log1⟩
295          set group [lrange $group 1 end]
296          set b_names [dialog::makeSomeButtons $b_names\
                  [expr {[lsearch -exact $group "right"] >= 0}] $left l r\
                                                              $right ymax]
299          if {[lsearch -exact $group "first"] < 0} then {
300             eval [list lappend buttons] $b_names
301             eval [list lappend button_help] $b_help
302             eval [list lappend button_press] $b_press
303          } else {
304             set buttons [concat $b_names $buttons]
305             set button_help [concat $b_help $button_help]
306             set button_press [concat $b_press $button_press]
307          }
308       }
309       if {![llength $button_press]} then {
310          dialog::cleanup $dial
311          error "No buttons in dialog."
312       }
313       incr ymax 33
```

If no buttons had been specified then the user would be unable to close the dialog, so that is an error. `ymax` is incremented from the top of the bottommost row of buttons to 13 pixels below the bottom of that row of buttons.

The final atom to make for the dialog material is the page title, which is static text in a single page dialog but a popup menu in a multipage ditto. As a title should, the title will not only appear topmost but also first in the dialog material.

```
314       if {$multipage} then {
315          set res\
                  [concat [list -m $pagemenu 100 10 300 29] $buttons $res]
316          set helpL [concat {Use this popup menu or the cursor keys to\
               go to a different page of the dialog.} $button_help $helpL]
```

```
320       } else {
321          set currentpage [lindex $pages 0]
322          set res\
                  [concat [list -t $currentpage 100 10 300 25] $buttons $res]
324          set helpL [concat $button_help $helpL]
325       }
```

Then it is time for the climax of this procedure: the call to `dialog`!

```
326       if {[info tclversion] >= 8.0} then {
327          set res [eval [list dialog -w $opts(-width) -h $ymax -T\
                                          $opts(-title) -help $helpL] $res]
329       } else {
330          if {[catch\
                  [concat [list dialog -w $opts(-width) -h $ymax] $res] res]}\
                                                                    then {
```

Unlike some of the built-in dialog commands in *Alpha*, `dialog` doesn't raise an error when e.g. Cancel is pressed, but the *Alpha 7* `dialog` command does raise an error if it is overstrained. That it can be overstrained is a bug.

```
333             dialog::cleanup $dial
334             alertnote "Sorry, you encountered a bug in Alpha 7's\
                          'dialog' command, which cannot handle very complex\
                       dialogs. If you are trying to edit many items at once,\
                                     try to edit them just one at a time."
338             error "Internal bug in 'dialog'."
339          }
340       }
```

Now the result of `dialog` must be parsed. In a multipage dialog the first item is the name of the current page, but in a single page dialog that item is missing. The following updates `currentpage` if necessary and ensures that `res` has the multipage structure.

```
341       if {$multipage} then {
342          set currentpage [lindex $res 0]
343       } else {
344          set res [linsert $res 0 $currentpage]
345       }
346 ⟨log1⟩       terminal::print_word emptyline "Result: $res" newline
```

The next `[llength $button_press]` elements in `res` are the control values of the buttons, but those are parsed last. Remaining results come from the various dialog items; these are parsed by the post-processing scripts found in the `postprocL` variable. During that, the `count` variable is the index of the first unparsed value in `res`. It is normally incremented by `1` after each item, but e.g. items which don't have a control value can issue a `continue` command in their post-processing scripts to skip that.

```
347       set count [expr {[llength $button_press] + 1}]
348       foreach script $postprocL {
349          eval $script
350          incr count
351       }
```

Finally the button results are parsed. This employs the fact that at most one of them can be 1 (and all others must be 0).

```
352       set count\
                  [lsearch -exact [lrange $res 1 [llength $button_press]] 1]
354       if {$count>=0} then {return [lindex $button_press $count]}
355    }
356 }
```

End of `while {1}` loop, and end of procedure.

`dialog::makeSomeButtons` (proc)

The `dialog::makeSomeButtons` procedure builds dialog material for a list of full-size buttons, while trying to keep them on the same line. The dialog material is returned and some variables are updated. The syntax is

> `dialog::makeSomeButtons` {*title list*} {*justification*} {*xmin*} {*left-var*} {*right-var*} {*xmax*} {*y-var*} {*minwidth*}?

where the '-var' arguments are names of variables in the caller's local context, whereas the other arguments are direct data. {*justification*} is 0 if the buttons should be put flush left and 1 if they should be put flush right. {*title list*} is the list of button titles.

The procedure tries to put (the top of) the buttons at the *y*-coordinate given by {*y-var*} and the *x*-coordinates between those given by {*left-var*} and {*right-var*}. If that doesn't work then it increases the {*y-var*} to the next line and resets the {*left-var*} and {*right-var*} to {*xmin*} and {*xmax*} respectively. Depending on {*justification*}, either the {*left-var*} or the {*right-var*} is incremented after a button has been added.

Buttons are made 20 pixels high and at least 17 pixels wider than the title. {*minwidth*} is the minimal width of a button; it defaults to 58. Buttons are put 13 pixels from each other.

```
357 proc dialog::makeSomeButtons\
      {titleL justification xmin leftvar rightvar xmax yvar {minwidth 58}} {
359    upvar 1 $leftvar left $rightvar right $yvar y
360    set widthL [list]
361    foreach title $titleL {
362       set w [expr {[dialog::text_width $title] + 17}]
363       if {$w < $minwidth} then {set w $minwidth}
364       lappend widthL $w
365    }
366    if {[expr [join $widthL "+13+"]] > $right - $left && ($xmin<$left ||\
                                                          $right<$xmax)} then {
368       incr y 33
369       set left $xmin
370       set right $xmax
371    }
372    set n 0
373    foreach title $titleL {
374       set w [lindex $widthL $n]
375       if {$w > $right - $left && ($xmin<$left || $right<$xmax)} then {
376          incr y 33
377          set left $xmin
```

```
378         set right $xmax
379      }
380      lappend res -b $title
381      if {$justification} then {
382         lappend res [expr {$right-$w}] $y $right [expr {$y+20}]
383         set right [expr {$right - $w - 13}]
384      } else {
385         lappend res $left $y [incr left $w] [expr {$y+20}]
386         incr left 13
387      }
388   }
389   set res
390 }
```

2.4.1 Construction and post-processing scripts

`dialog::simple_type` (array)
`dialog::complex_type` (array)

The `dialog::simple_type` and `dialog::complex_type` arrays are where the code defining the various item types is stored. The indices into these arrays are the type names (first item in the actual type, when seen as a list) and each entry contains the *construction script* for that item type; this script is responsible for inserting an item of the type in question into the dialog.

The following local variables are available when the scripts are evaluated:

`res`	The list to which the dialog material for the item should be appended.
`dial`	The identifier for accessing values in the current dialog.
`type`	The item type.
`page`	The item page.
`name`	The item name.
`help`	The user-supplied help text for the item, or an empty string if there was none.
`script`	The *post-processing script* for the item. This is initialised to code which makes the next control value the new value of this item, but items with Set... buttons will have to redefine it.
`val`	The default value for the item.
`left`	The *x*-coordinate of the left margin for the items: this is where the left edge of the item name should be put.
`right`	The *x*-coordinate of the right margin for the items. The dialog material that is generated should be between the *x*-coordinates `$left` and `$right`.
`y`	The *y*-coordinate of the top side of the item. After inserting the item, this variable should be incremented to equal the *y*-coordinate of the bottom of the bounding rectangle of the item's material.

`visible` A boolean for whether this item produces any visible material. It defaults to `1`, but if it is set to `0` then the `y` variable will not be incremented after the item and the help text will be ignored. The item can still have a post-processing script, but that should end with `continue` since there isn't a control value result for the item.

In addition, the following local variables must be left alone: `items`, `pages`, `typeA`, `helpA`, `currentpage`, `opts`, `ymax`, `multipage`, `pagemenu`, `helpL`, and `postprocL`. This list may change in the future, but variable names at most three characters long should be safe.

The following global variables have been made accessible via the `global` command:

`dialog::indentsame`	The recommended minimal indentation (from *x*-coordinate `$left`) for item values that are printed on the same "line" as their names.
`dialog::indentnext`	The recommended minimal indentation (from *x*-coordinate `$left`) for item values that are not printed on the same "line" as their names.
`dialog::simple_type`	Obvious?
`dialog::complex_type`	Obvious?
`alpha::platform`	The platform that AlphaTcl is being run on, either `alpha` or `tk`.

Other global or local variables may be used in any way the script pleases, but don't expect local variables to be the same as the last time the script was evaluated.

The advantage with keeping contruction scripts in arrays like this in comparison with having a procedure with a large `switch` command is that it is much easier to add definitions of new types. The advantage in comparision with keeping several procedures in a designated namespace is that you don't have to spend a lot of code on passing information between the caller and the callee.

Post-processing scripts, on the other hand, are usually built on the fly by the construction scripts. In some cases they are the same for all items of the same type, but it is often necessary to embed the page and item names into the script. This is fairly straightforward if the script simply is a single procedure call, since the script can then be built as the list of words in that command. This might look like

```
set script [list myPostprocProc $dial $page $name]
```

which puts in `script` a command with the structure

`myPostprocProc` {*dial*} {*page*} {*name*}

where {*dial*}, {*page*}, and {*name*} are the values these variables had when the script was built—the `list` command even takes care of quoting the arguments when necessary. The default post-processing script and the *Alpha 7* post-processing scripts for items with Set... buttons are both constructed in this way (with a slight extra twist).

For more complex post-processing scripts this might be unfeasible. In that case, the following construction is useful:

```
set script [list set T $page,$name]
append script {
    ⟨bulk of the script⟩
}
```

The script will then begin with a `set` command into which the ⟨*page*⟩,⟨*name*⟩ construction has been embedded, and thus the ⟨*bulk of the script*⟩, which is a fixed string, may refer to this string as `$T`. Note however that the newline before the ⟨*bulk of the script*⟩ is necessary: it separates the `set` command returned by `list` from the first command in the ⟨*bulk of the script*⟩.

There are however a couple of variables which a post-processing script do, and usually need to, have access to. These are:

`res` The list of control values returned by `dialog`.

`count` The index into `res` of the first value not yet parsed. Unless a post-processing script does a `continue`, this variable will be incremented by `1` after the script has been evaluated. An item for which there are several control values returned by `dialog` must itself modify `count` accordingly.

`dial` The identifier of the current dialog, for value access.

The variables that construction scripts should avoid should also be avoided by post-processing scripts.

`dialog::indentsame` (var.)
`dialog::indentnext` (var.)

The `dialog::indentsame` and `dialog::indentnext` variables are lower bounds for how much the value of a dialog item is indented relative to the name. `indentsame` is used for values on the same line as the item name, whereas `indentnext` is used for values whose names are on the next line. The unit is screen pixels.

```
391 set dialog::indentsame 80
392 set dialog::indentnext 40
```

2.4.2 TextEdit item types

`dialog::makeEditItem` (proc)

The `dialog::makeEditItem` procedure generates the dialog material for an item whose value is edited as explicit text, in a box. The syntax is

`dialog::makeEditItem` {*mat-var*} {*script-var*} {*left*} {*right*} {*y-var*} {*name*} {*value*} {*lines*}? {*minwidth*}? {*maxwidth*}?

where {*mat-var*}, {*script-var*}, and {*y-var*} are names of variables in the caller's local context, whereas the other arguments are direct data. {*mat-var*} collects the dialog material and {*script-var*} the post-processing commands that should be applied for this item.[6] {*left*}, {*right*}, and {*y-var*} is the coordinates of the left, right, and top sides of a rectangle by which the material of the item should be bounded. {*y-var*} is incremented to equal the bottom of this rectangle. {*name*} and {*value*} are the name and the initial value

[6] Currently this variable is neither changed nor inspected. I'm not sure why I addded the argument in the first place. /LH

of the item, respectively. {*lines*} is the height of the edit box in lines and defaults to 1. {*minwidth*} is the minimal width in pixels of the box and defaults to 110. {*maxwidth*} is the maximal width of the box in pixels and defaults to `$right-$left`.

The 19 below are 13 for the standard item separation and $3+3$ for the frame around a TextEdit item. The default `minwidth` is arbitrarily chosen.

```
393 proc dialog::makeEditItem {mvar svar left right yvar name val {lines 1}\
                                                {minwidth 110} {maxwidth {}}} {
395     upvar 1 $mvar M $yvar y
396     global dialog::indentsame dialog::indentnext
397     if {$maxwidth==""} then {set maxwidth [expr {$right-$left}]}
398     set nw [expr {[dialog::text_width $name] + 1}]
399     if {$nw<${dialog::indentsame}-13} then {
400         set nw [expr {${dialog::indentsame}-13}]
401     }
402     if {$lines == 1 && $nw+19+$minwidth < $right-$left ||\
                                    $nw+19+$maxwidth <= $right-$left} then {
404         incr y 3
405         lappend M -t $name $left $y [expr {$left+$nw}] [expr {$y+15}]
406         set ew [expr {$right - $left - $nw - 19}]
407         if {$ew>$maxwidth} then {set $ew $maxwidth}
408         lappend M -e $val [expr {$left+$nw+16}] $y\
                        [expr {$left+$nw+$ew+16}] [expr {$y + 16*$lines - 1}]
410     } else {
411         lappend M -t $name $left $y [expr {$left+$nw}] [expr {$y+15}]
412         incr y 19
413         set ew [expr {$right - $left - ${dialog::indentnext} - 6}]
414         if {$ew>$maxwidth} then {set $ew $maxwidth}
415         lappend M -e $val [expr {$right - 3 - $ew}] $y\
                              [expr {$right - 3}] [expr {$y + 16*$lines - 1}]
417     }
418     set y [expr {$y + 16*$lines + 2}]
419 }
```

`dialog::simple_type(var)`
`dialog:`
`:simple_type(var2)`

The `var` item type provides a box in which the item value can be edited as a string; it could be removed as this is also the default for undefined simple types. The `var2` type is similar, but the text box is two lines high, instead of one as for the `var` type.

```
420 array set dialog::simple_type\
          {var {dialog::makeEditItem res script $left $right y $name $val}}
422 array set dialog::simple_type\
       {var2 {dialog::makeEditItem res script $left $right y $name $val 2}}
```

`dialog::simple_type`
`(password)`

The `password` item type is almost the same as `var`; the only difference is that the editable text box is deliberately so small that the text written in it cannot be read.

At least in some cases, the Mac OS Toolbox routines for TextEdit boxes draw the initial text in them, even when the that means drawing outside the corresponding rectangle. This can result in passwords being clearly written on the screen. To avoid this, the initial text in the TextEdit atom of a `password` item consists entirely of spaces. Passwords that are not edited not changed by the post-processing script.

```
424 array set dialog::simple_type {password {
```

```
425   set nw [expr {[dialog::text_width $name] + 1}]
426   lappend res -t $name $left $y [expr {$left + $nw}] [expr {$y + 15}]
427   incr nw 13
428   if {$nw<${dialog::indentsame}} then {set nw ${dialog::indentsame}}
429   regsub -all {.} $val { } vv
430   lappend res -e $vv [expr {$left + $nw + 3}] [expr {$y + 6}]\
                                        [expr {$right - 3}] [expr {$y + 7}]
432   incr y 15
433   set script [list set T $page,$name]
434   append script {
435      regsub -all {.} [dialog::valGet $dial $T] { } vv
436      if {[lindex $res $count] != $vv} then {
437         dialog::valChanged $dial $T [lindex $res $count]
438      }
439   }
440 }}
```

2.4.3 Uneditable item types

`dialog::lines_to_text` (proc)

The `lines_to_text` procedure takes a list of lines, as returned by e.g. the `width_linebreak` procedure, and returns dialog material for showing those lines as static text. The two important non-trivialities there is are that (i) there is a limit on how long a string in a dialog atom can be and (ii) there is more vertical space between two paragraphs than between two lines in the same paragraph.

The syntax is

`dialog::lines_to_text` {*line list*} {*left*} {*right*} {*y-var*}

{*line list*} is the list of lines. {*left*} and {*right*} are the *x*-coordinates of the respective left and right edges of the text items that are created. It is assumed that each line of text fits between those two positions. The {*y-var*} is the name of a variable in the caller's local context giving the top edge of the first text line. The procedure increments it to give the bottom edge of the last line in the paragraph.

```
441 proc dialog::lines_to_text {lineL left right yvar} {
442   upvar 1 $yvar y
443   global dialog::strlength
444   set res [list]
445   set item_lines [list]
446   set item_length -1
447   foreach line $lineL {
448      if {$line!="\r"} then {
449         incr item_length [expr {1 + [string length $line]}]
450         if {${dialog::strlength}<$item_length} then {
451            lappend res -t [join $item_lines \r] $left $y $right\
                                [incr y [expr {[llength $item_lines] * 16}]]
453            set item_lines [list $line]
454            set item_length [string length $line]
455         } else {
456            lappend item_lines $line
```

```
457          }
458       } else {
459          if {[llength $item_lines]} then {
460             lappend res -t [join $item_lines \r] $left $y $right\
                                  [incr y [expr {[llength $item_lines] * 16}]]
462          }
463          incr y 6
464          set item_lines [list]
465          set item_length -1
466       }
467    }
468    if {[llength $item_lines]} then {
469       lappend res -t [join $item_lines \r] $left $y $right\
                                  [incr y [expr {[llength $item_lines] * 16}]]
471    }
472    if {[llength $res]} then {incr y -1}
473    return $res
474 }
```

`dialog:`
`:simple_type(text)`

A `text` item has no value; it merely prints the {*name*} in the dialog as a static text item. This might for example be used to make subheadings in a dialog.

```
475 array set dialog::simple_type {text {
476     eval [list lappend res] [dialog::lines_to_text\
              [dialog::width_linebreak $name [expr {$right-$left}]] $left\
                                                                    $right y]
479     set script {continue}
480 }}
```

`dialog::simple_type`
`(static)`

A `static` item looks like a `var` item where the value for some reason cannot be edited. It is mainly used for showing information in a dialog.

```
481 array set dialog::simple_type {static {
482     set nw [expr {[dialog::text_width $name] + 1}]
483     if {$nw<${dialog::indentsame}-13} then {
484         set nw [expr {${dialog::indentsame}-13}]
485     }
486     lappend res -t $name $left $y [expr {$left+$nw}] [expr {$y+15}]
487     set vw [expr {[dialog::text_width $val] + 1}]
488     lappend res -t $val
489     if {$nw + 13 + $vw < $right - $left} then {
490         lappend res [expr {$left + $nw + 13}] $y
491     } else {
492         incr y 16
493         lappend res [expr {$left + ${dialog::indentnext}}] $y
494     }
495     lappend res $right [incr y 15]
496     set script {continue}
497 }}
```

`dialog::mute_types` (var.)

The `dialog::mute_types` variable is a list of “mute” item types, i.e., they don’t return

any value.

```
set dialog::mute_types [list text static]
```

2.4.4 Elementary control item types

dialog:
:simple_type(flag)

`flag` items are simple checkboxes. They could be implemented using `dialog::checkbox`, but that wouldn't take notice of the margins that are used.

```
array set dialog::simple_type {flag {
    lappend res -c $name $val
    if {[info tclversion]>=8.0} then {lappend res -font 2}
    lappend res $left $y $right [incr y 15]
}}
```

dialog::complex_type
(multiflag)

`multiflag` items are a group of checkboxes, set in two columns and with the overall item name as a heading. The format is

`multiflag` {*subitems list*}

where each element in the {*subitems list*} is the text to put next to one of the checkboxes. The value of the item is the list of values of the individual checkboxes, so it is a list of zeros and ones.

Verical separation between atoms in the `multiflag` item is 3 pixels, whereas horizontal separation is 10 pixels. Both these distances are as in the package installation dialog.

```
array set dialog::complex_type {multiflag {
   eval [list lappend res] [dialog::lines_to_text\
      [dialog::width_linebreak $name [expr {$right-$left}]] $left $right\
                                                                    y]
   set flag_list [lindex $type 1]
   set y2 $y
   set r [expr {($left+$right)/2 - 5}]
   set l [expr {($left+$right)/2 + 5}]
   for {set n 0} "\$n < ([llength $flag_list]+1)/2" {incr n} {
      lappend res -c [lindex $flag_list $n] [lindex $val $n]
      if {[info tclversion]>=8.0} then {lappend res -font 2}
      lappend res $left [incr y 3] $r [incr y 15]
   }
   for {} "\$n < [llength $flag_list]" {incr n} {
      lappend res -c [lindex $flag_list $n] [lindex $val $n]
      if {[info tclversion]>=8.0} then {lappend res -font 2}
      lappend res $l [incr y2 3] $right [incr y2 15]
   }
   set script [list dialog::modified $dial $page,$name]
   append script { [lrange $res $count [incr count }
   append script [expr {[llength $flag_list] - 1}] {]]}
}}
```

This is also where a type for radio buttons should be defined, if there was one.

2.4.5 Menu item types

dialog::makeMenuItem (proc)

The `dialog::makeMenuItem` procedure builds the dialog material corresponding to a menu item. It has the syntax

> `dialog::makeMenuItem` {*mat-var*} {*script-var*} {*left*} {*right*} {*y-var*} {*name*} {*item list*} {*value*}

where the '-var' arguments are names of variables in the caller's local context and the other arguments provide direct data. In the {*mat-var*} variable the dialog material for the item is collected. The {*script-var*} variable stores the post-processing script for the item, but currently this argument is not used (and it is unclear why it was added in the first place). {*left*}, {*right*}, and the {*y-var*} variable give three sides of the bounding rectangle for the item. {*name*} is the item name, {*item list*} the list of items for the menu, and {*value*} the default value.

If the item name leaves less than 50 pixels for the menu then the menu is put on the line below the item name. This value was chosen quite arbitrarily.

```
526 proc dialog::makeMenuItem {mvar svar left right yvar name itemL value} {
527     upvar 1 $mvar M $yvar y
528     global dialog::indentsame dialog::indentnext
529     set nw [expr {[dialog::text_width $name]+1}]
530     set itemL [linsert $itemL 0 $value]
531     if {$nw<${dialog::indentsame}} then {set nw ${dialog::indentsame}}
532     if {$right - $left - $nw < 50} then {
533        lappend M -t $name $left $y [expr {$left+$nw}] [incr y 15]
534        incr y 5
535        lappend M -m $itemL [expr {$left+${dialog::indentnext}+1}]
536     } else {
537        incr y
538        lappend M -t $name $left $y [expr {$left+$nw}] [expr {$y+15}]
539        lappend M -m $itemL [expr {$left+$nw+14}]
540     }
541     lappend M $y [expr {$right-2}] [incr y 18]
542 }
```

dialog:: :complex_type(menu)

The `menu` types provide a popup menu of items to choose from. In this case the {*type*} has the form

> `menu` {*item list*}

where {*item list*} is the list of items in the menu.

```
543 array set dialog::complex_type {menu {dialog::makeMenuItem res script\
                                  $left $right y $name [lindex $type 1] $val}}
```

dialog::simple_type (colour)

dialog:: :simple_type(mode)

The `colour` and `mode` simple types are variations on the `menu` type in which the item lists are *Alpha*'s lists of colours and modes respectively.

```
546 array set dialog::simple_type {colour {
547     global alpha::colors
548     dialog::makeMenuItem res script $left $right y $name\
                                                    ${alpha::colors} $val
```

```
} mode {
    dialog::makeMenuItem res script $left $right y $name\
                                    [linsert [mode::listAll] 0 "<none>"] $val
}}
```

dialog::complex_type (menuindex)

The `menuindex` types are visually the same as the `menu` types, but the value is the index into the list of the chosen item rather than the actual item. The {*type*} has the form

`menuindex` {*item list*}

Note how the post-processing script is used to convert the control value returned by `dialog` to an index.

```
array set dialog::complex_type {menuindex {
    set script [list dialog::valChanged $dial $page,$name]
    append script { [} [list lsearch -exact [lindex $type 1]]
    append script { [lindex $res $count]]}
    catch {lindex [lindex $type 1] $val} val
    dialog::makeMenuItem res script $left $right y $name\
                                                    [lindex $type 1] $val
}}
```

2.4.6 `specialSet` item types

For many preference types, the `dialog` command provides no convenient method of editing in the dialog, so in order to edit those values, the user is instead taken to an auxiliary dialog which provide a more convenient presentation of the item value. Everything that appears in the main dialog is the item name, a pretty-printed representation of the item value (static text), and a button labelled Set.... The pretty-printed representation is generated by the procedure `dialog::specialView::`⟨*type*⟩. Clicking the Set... button calls a procedure named `dialog::specialSet::`⟨*type*⟩, which puts up a dialog in which the user can edit the item value. These `specialSet` procedures retrieve the values to edit using `dialog::getFlag` and store them after editing using `dialog::modified`, both of which are designed specifically to work with preferences.

That is the way things are in the old preferences dialogs. In the new dialogs, things are handled differently—in particular there is no reason to assume that the values being edited are preferences in the traditional sense—but as much work has been put into designing the auxiliary editing dialogs it is desirable to reuse the `specialSet` procedures as far as possible. For that reason, the new dialogs code stores all values being edited in such a way that `dialog::modified` and `dialog::getFlag` will access them, even though they are not preferences. This way, the `specialSet` procedures will do the right thing for the new dialogs even though they haven't been designed for this.

dialog::makeSetItem (proc)

The `dialog::makeSetItem` procedure builds dialog material for an item with a Set... button; more precisely the material for the item name and button. It does not make anything for the actual item value, but returns the rectangle between the name and button so that the caller may decide on whether the value should be put there. The syntax is

`dialog::makeSetItem` {*mat-var*} {*script-var*} {*left*} {*right*} {*y-var*} {*name*} {*button script*}

where the '-var' arguments are names of variables in the caller's local context and the other arguments provide direct data. In the {*mat-var*} and {*script-var*} variables the dialog material and post-processing script respectively for the item are collected. {*left*}, {*right*}, and the {*y-var*} variable give three sides of the bounding rectangle for the item. {*name*} is the item name. {*button script*} is a script that will be evaluated when the Set... button is pressed.

A tricky matter is that you have to embed the values of `dial`, `page`, and `name` in the {*button script*}. This not so hard if you build each command as a list; see the definition of `dialog::simple_type(binding)` below for an example. See also [1] for a collection of notes on how to build scripts on-the-fly like this.

The implementation assumes that {*name*} and the button fits on one a single line. The extra 17 pixels in the width `$bw` of the button is to get the same width as used in traditional dialogs. The rounded corners in the button use 5 of these pixels on each side.

```
562 proc dialog::makeSetItem {Mvar Svar left right yvar name bscript} {
563    upvar 1 $Mvar M $Svar S $yvar y
564    global dialog::ellipsis dialog::indentsame
565    set nw [expr {[dialog::text_width $name]+1}]
566    set bw [expr {[dialog::text_width "Set${dialog::ellipsis}"] + 17}]
567    lappend M -t $name $left $y [expr {$left + $nw}] [expr {$y + 15}]
568    lappend M -b "Set${dialog::ellipsis}"
569    if {[info tclversion]>=8.0} then {
570       lappend M -set [list $bscript +1]
571       set S {}
572    } else {
573       set S [list if {[lindex $res $count] == 1} then $bscript]
574    }
575    lappend M [expr {$right - $bw}] $y $right [expr {$y + 15}]
576    set nw [expr {$nw+13}]
577    if {$nw<${dialog::indentsame}} then {set nw ${dialog::indentsame}}
578    list [expr {$left + $nw}] $y [expr {$right - $bw - 13}] [incr y 15]
579 }
```

`dialog::makeStaticValue` (proc)

The `dialog::makeStaticValue` procedure builds the dialog material for a static value. The syntax is

> `dialog::makeStaticValue` {*left*} {*right*} {*y-var*} {*value*} {*suboptions*} {*abbr-ratio*}? {*rect*}?

and the returned value is the dialog material. {*value*} is the text to show. {*rect*} is, if it is provided, a rectangle (assumed to be one line tall) in which the procedure tries to fit the {*value*}. If this doesn't work then the value is instead put below all previous dialog material. {*left*} and {*right*} are taken as the left and right sides of the bounding rectangle in which dialog material may be put. The {*y-var*} variable in the caller's local context is assumed to be the *bottom* of the bounding rectangle of all previous dialog material, and it is incremented to accomodate for the returned `-t` item.

The {*abbr-ratio*} argument controls how a {*value*} that is to wide to fit on one line should be abbreviated. The value is a real number that gives the fraction of the abbreviated text that should be before the point of abbreviation. `0` means remove text at the beginning,

1 means remove at the end, and the default 0.33 leaves twice as much after the point of abbreviation as before it.

The {*suboptions*} argument, finally, is used for supplying extra suboptions (most likely -dnd) to the -t option. These are currently only inserted for *Alphatk*.

```
proc dialog::makeStaticValue\
            {left right yvar value subopt {ratio 0.33} {rect {0 0 0 0}}} {
   global dialog::indentnext alpha::platform
   upvar 1 $yvar y
   set vw [expr {[dialog::text_width $value] + 1}]
   if {[lindex $rect 2] - [lindex $rect 0] >= $vw} then {
      set res [list -t $value]
      if {${alpha::platform} != "alpha"} then {
         set res [concat $res $subopt]
      }
      if {[lindex $rect 3] > $y} then {set y [lindex $rect 3]}
      concat $res $rect
   } else {
      set res [list -t]
      lappend res [dialog::width_abbrev $value\
               [expr {$right - $left - ${dialog::indentnext} - 1}] $ratio]
      if {${alpha::platform} != "alpha"} then {
         set res [concat $res $subopt]
      }
      incr y
      lappend res [expr {$left + ${dialog::indentnext}}] $y $right\
                                                              [incr y 15]
   }
}
```

dialog::simple_type (binding)

binding items constitute a straightforward application of the dialog::makeSetItem and dialog::makeStaticValue procedures.

```
array set dialog::simple_type {binding {
    set R [dialog::makeSetItem res script $left $right y $name\
                   [list dialog::specialSet::binding $dial "$page,$name"]]
    set vv [dialog::specialView::binding $val]
    eval [list lappend res]\
                   [dialog::makeStaticValue $left $right y $vv {} 0.33 $R]
}}
```

dialog::simple_type(file)
dialog::simple_type(folder)
dialog::simple_type(url)

The file, folder, and url item types allow the specification of existing files, folders, and URLs.

```
array set dialog::simple_type {file {
    set R [dialog::makeSetItem res script $left $right y $name\
                      [list dialog::specialSet::file $dial "$page,$name"]]
    eval lappend res [dialog::makeStaticValue $left $right y $val\
                   [list "-dnd" $dial [list "$page,$name" $type]] 0.33 $R]
} folder {
    set R [dialog::makeSetItem res script $left $right y $name\
                    [list dialog::specialSet::folder $dial "$page,$name"]]
```

```
620     eval lappend res [dialog::makeStaticValue $left $right y $val\
                    [list "-dnd" $dial [list "$page,$name" folder]] 0.33 $R]
623 } url {
624     set R [dialog::makeSetItem res script $left $right y $name\
                        [list dialog::specialSet::url $dial "$page,$name"]]
626     eval lappend res [dialog::makeStaticValue $left $right y $val\
                    [list "-dnd" $dial [list "$page,$name" $type]] 0.33 $R]
629 }\
```

[If this had been a `.tcl` file then I wouldn't have been able to put a comment here, since this is technically inside a list. The `.dtx` format allows you to put a comment between any two rows of the program, though.]

`dialog:`
`:simple_type(date)`

The `date` item type specifies a time (date and time of day). The format is as returned by `clock scan`.

```
630  date {
631     set R [dialog::makeSetItem res script $left $right y $name\
                        [list dialog::specialSet::date $dial "$page,$name"]]
633     eval lappend res [dialog::makeStaticValue $left $right y\
                                                [clock format $val] {} 1 $R]
636 }}
```

`dialog::simple_type`
`(appspec)`

The `appspec` item type stores references to applications, in a manner similar to that used for preferences whose names end in '`Sig`'. The main difference between appspecs and sigs is that the former may be file names of applications, so that also applications which do not have unique sigs can be specified.

```
637 array set dialog::simple_type {appspec {
638     if {${alpha::platform} == "alpha" &&\
                                    [regexp {^'(....)'$} $val "" sig]} then {
640         if {[catch {nameFromAppl $sig} vv]} then {
641            set vv "Unknown application with sig '$sig'"
642         }
643     } else {
644         set vv $val
645     }
646     set R [dialog::makeSetItem res script $left $right y $name\
                [list dialog::set_appspec $dial $page $name "Select $name"]]
648     eval lappend res\
                    [dialog::makeStaticValue $left $right y $vv {} 0.33 $R]
650 }}
```

`dialog::set_appspec`
(proc)

The `dialog::set_appspec` procedure is a modernised version of `dialog::specialSet::Sig` (or perhaps it is rather `dialog::_findApp`, as that does everything that the user sees). The syntax is

`dialog::set_appspec` {*page*} {*name*} {*prompt*}

where {*page*} is the page of the dialog item, {*name*} is the item name, and {*prompt*} the prompt for the dialog. The procedure reads the old value from the `valueA` array in the caller's local context and stores the new value there as well.

The main improvement in `dialog::set_appspec` as compared to `dialog::_findApp` is that the former doesn't panic when the desktop database wouldn't select the same file as the user did, but instead calmly asks whether it should return the sig or the path.

```
proc dialog::set_appspec {dial page name prompt} {
   global alpha::platform
   set val [dialog::valGet $dial $page,$name]
   if {${alpha::platform} == "alpha" &&\
                                  [regexp {^'(....)'$} $val "" sig]} then {
      catch {nameFromAppl $sig} val
   }
   if {[catch {getfile $prompt $val} val]} then {return ""}
   if {${alpha::platform} == "alpha"} then {
      set sig [getFileSig $val]
      set app [nameFromAppl $sig]
      if {$app != $val} then {
         catch {
            if {[dialog::yesno -y "Path" -n "Sig" -c "Application sig\
                  '$sig' is mapped to '$app', not '$val'. Which should I\
                                                                  use?"]}\
            then {dialog::valChanged $dial $page,$name $val}\
            else {dialog::valChanged $dial $page,$name '$sig'}
         }
      } else {
         dialog::valChanged $dial $page,$name '$sig'
      }
   } else {
      dialog::valChanged $dial $page,$name $val
   }
}
```

`dialog::simple_type (searchpath)`

The `searchpath` type is a list of folders. The *Alpha* implementation is as for most other types with Set... buttons, but *Alphatk* replaces that with an in-dialog listpick list to stop it from growing.

```
if {${alpha::platform} == "alpha"} then {
   array set dialog::simple_type {searchpath {
      set R [dialog::makeSetItem res script $left $right y $name\
               [list dialog::specialSet::searchpath $dial "$page,$name"]]
      if {![llength $val]} then {
         eval [list lappend res] [dialog::makeStaticValue $left $right\
                                 y "No search paths currently set." {} 1 $R]
      } else {
         foreach path $val {
            eval [list lappend res]\
                        [dialog::makeStaticValue $left $right y $path {}]
         }
      }
   }}
} else {
   array set dialog::simple_type {searchpath {
```

```
694       dialog::makeSetItem res script $left $right y $name\
                  [list dialog::specialSet::searchpath $dial "$page,$name"]
696       lappend res "-l" $val 3
697       lappend res "-dnd" $dial [list "$page,$name" searchpath]
698       lappend res [expr {$left + ${dialog::indentnext}}] [incr y]\
                                                          $right [incr y 51]
700    }}
701 }
```

2.4.7 Listpick item types

`dialog::edit_subset` (proc)

The `dialog::edit_subset` procedure is mainly a wrapper around the `listpick` command that is somewhat simpler to use in post-processing and button action scripts. The syntax is

`dialog::edit_subset` {*full set*} {*page*} {*name*} {*prompt*}

where {*full set*} is the list to build the listpick from, {*page*} is the page of the dialog item, {*name*} is the item name, and {*prompt*} the prompt. The procedure reads the old value from the `valueA` array in the caller's local context and stores the new value there as well.

```
702 proc dialog::edit_subset {setL dial page name prompt} {
703    if {![catch {
704       listpick -p $prompt -l -L [dialog::valGet $dial $page,$name] $setL
705    } res]} then {
706       set val [list]
707       catch {
708           foreach item $res {lappend val $item}
709           dialog::valChanged $dial $page,$name $val
710       }
711    }
712 }
```

The reason for the somewhat odd way of storing the selected subset is that `listpick` doesn't quote its result properly: if some item contains a mismatched brace or backslash then `res` needs not be a proper list. It is furthermore a rather ugly list (with braces around every item) and hence it is reconstructed to look more like a sequence of words.

`dialog::complex_type (subset)`

The `subset` types provide the ability to select a subset of a given set (or technically rather a sublist of a given list, which is slightly more general) using a `listpick` dialog. The type format is

`subset` {*set*}

where {*set*} is the list of items in the set. The value is the list of items in the selected subset.

```
713 array set dialog::complex_type {subset {
714     dialog::makeSetItem res script $left $right y $name\
              [list dialog::edit_subset [lindex $type 1] $dial $page $name\
                                                          "Edit subset"]
```

```
    eval [list lappend res]\
                         [dialog::makeStaticValue $left $right y $val {} 1]
}}
```

dialog::simple_type (modeset)

The `modeset` item type is a special case of the `subset` types where the universe is the list of modes.

```
array set dialog::simple_type {modeset {
    dialog::makeSetItem res script $left $right y $name\
             [list dialog::edit_subset [mode::listAll] $dial $page $name\
                                                          "Select modes"]
    eval [list lappend res]\
                         [dialog::makeStaticValue $left $right y $val {} 1]
}}
```

2.4.8 Miscellanea

dialog::complex_type (global)

A `global` type has the structure

`global` {*preference name*}

This essentially causes the item to have the same type as the {*preference name*} preference.

```
array set dialog::complex_type {prefItemType {
    set type [dialog::prefItemType [lindex $type 1]]
    continue
}}
```

dialog::simple_type (thepage)

An `thepage` item simply reports back the name of the current page. The item is invisible and the initial value is ignored.

```
array set dialog::simple_type {thepage {
   set script [list dialog::valChanged $dial $page,$name]
   append script { $currentpage
      continue
   }
   set visible 0
}}
```

dialog::hide_item (proc)
dialog::show_item (proc)
dialog::complex_type (hidden)

Sometimes you might not want to show all the items in a dialog, but only show them if the user clicks an "Advanced settings" (or something) button. This can be accomplished using the `hide_item` and `show_item` procedures, which have the syntaxes

`dialog::hide_item` {*page*} {*name*} {*type-arr*}?
`dialog::show_item` {*page*} {*name*} {*type-arr*}?

{*page*} is the name of the page on which the item can be found and {*name*} is the name on that page of the item. The procedures work by modifying the entry ⟨*page*⟩,⟨*name*⟩ of a variable in the caller's local context; this entry is assumed to be where the type of the item is stored. The {*type-arr*} argument is the name of this array: it defaults to `typeA`

which is correct when `hide_item` and `show_item` are called from within the `make` and `make_paged` procedures.

```
proc dialog::hide_item {page item {typevar typeA}} {
   upvar 1 $typevar typeA
   set typeA($page,$item) [linsert $typeA($page,$item) 0 hidden]
}
proc dialog::show_item {page item {typevar typeA}} {
   upvar 1 $typevar typeA
   if {[lindex $typeA($page,$item) 0]=="hidden"} then {
      set typeA($page,$item) [lreplace $typeA($page,$item) 0 0]
   }
}
```

To make an item hidden by default, you simply prepend a `hidden` to the actual type when you create it. This above works because of how the `hidden` item type is defined. Items of this type are essentially ignored when the dialog contents for the user: there is nothing shown and nothing the user does will change the item value. Furthermore the format of this type is

`hidden` ⟨*type when visible*⟩

e.g. `hidden menu {good better best}`. Thus if you remove the `hidden`, which is what `show_item` does, the item type will become the ⟨*type when visible*⟩ and that can be just about anything.

```
array set dialog::complex_type {hidden {
   set script {continue}
   set visible 0
}}
```

2.5 Main dialogs interface

`dialog::make` (proc) The most basic procedure for making a generic dialog has the syntax

`dialog::make` ⟨*option*⟩* {*page*}$^+$

where each {*page*} is a list with the structure

{*page name*} {*item*}*

and each {*item*} in turn is a list with the structure

{*type*} {*name*} {*value*} {*help*}$^?$

An ⟨*option*⟩ is one of

`-ok` {*OK button title*}
`-cancel` {*cancel button title*}
`-title` {*dialog window title*}
`-defaultpage` {*name of default page*}
`-hidepages` {*list of pages to hide*}
`-addbuttons` {*button list*}
`-width` {*dialog window width*}
`-debug` {*debug level*}

where the {*button list*} has the structure

$$\left(\{\textit{name}\}\ \{\textit{help}\}\ \{\textit{script}\}\right)^{+}$$

Here each tripple {*name*} {*help*} {*script*} describes one additional button. {*name*} is the button name, i.e., the text that will be shown on the button. The button will be made wide enough to contain the whole {*name*}. {*help*} is the help text for the button. {*script*} is a script that is evaluated when the button is clicked.

752 `proc dialog::make {args} {`

There are a number of local variables in `make` that must be explained, since the button scripts passed by the caller may need to access these variables. First there are a couple of arrays in which the page descriptions are stored.

`pageA` The index into this array is the name of a page. An entry contains the list of names of items on that page.

`typeA` The index into this array has the form ⟨*page*⟩`,`⟨*item*⟩, where ⟨*page*⟩ is the name of a page and ⟨*item*⟩ is the name of an item on that page. An entry contains the type of that item.

`helpA` The index has the same form as in the `typeA` array. An entry contains the help text for that entry, but an item needs not have an entry in this array (it can be left unset).

There are a couple of additional scalar variables that are of interest.

`retCode`, `retVal` When the `retCode` variable is set, the dialog is logically closed and the procedure returns. If the variable is set to `0` then `make` executes a normal return and the returned value will be the list of item values. If the variable is set to anything else then that will used for the `-code` option of `return` and the returned value will be taken from the `retVal` variable, which must then be initalised.

`dial` This contains the reference string to use with `valGet`, `valSet`, and friends when accessing the values of items in the dialog.

`currentpage` This contains the name of the current page in the dialog.

`pages` This is a list of pages and items to show in the dialog. It is similar to the result of `array get pageA`, but the order of pages is as specified in the call and hidden pages are not included.

`opts(-addbuttons)` This is {*button list*} specified by the caller. Button scripts can modify this list to change the text on their button.

`state` This is initialized to `0` before the first time the dialog is shown and then the procedure leaves it alone. Button scripts may change it to keep track of what "state" (mostly: which items/pages are currently hidden) the dialog is in.

`optionL` The list of additional options to pass to `dialog::handle`.

The first part of the procedure is all about interpreting the arguments.

```
753   set opts(-ok) OK
754   set opts(-cancel) Cancel
755   set opts(-title) ""
756   set opts(-width) 400
757   set opts(-debug) 0
758   set opts(-hidepages) [list]
759   getOpts {-title -defaultpage -ok -cancel -addbuttons -width -debug\
                                                                  -hidepages}
761   set dial [dialog::create]
762   set pages [list]
763   foreach pagearg $args {
764      set page [lindex $pagearg 0]
765      set pageA($page) [list]
766      foreach item [lrange $pagearg 1 end] {
767         set name [lindex $item 1]
768         set typeA($page,$name) [lindex $item 0]
769         dialog::valSet $dial $page,$name [lindex $item 2]
770         if {[llength $item]>3} then {
771            set helpA($page,$name) [lindex $item 3]
772         }
773         lappend pageA($page) $name
774      }
775      if {[lsearch -exact $opts(-hidepages) $page]<0} then {
776         lappend pages $page $pageA($page)
777      }
778   }
779   if {[info exists opts(-defaultpage)]} then {
780      set currentpage $opts(-defaultpage)
781   } else {
782      set currentpage [lindex $pages 0]
783   }
784   set optionL [list -width $opts(-width) -title $opts(-title)]
785   set main_buttons [list\
786     [list $opts(-ok) "Click here to use the current settings."\
                                                        {set retCode 0}\
788       $opts(-cancel) "Click here to discard any changes you've made to\
                    the settings." {set retCode 1; set retVal "cancel"}]\
791     first right]
```

The second part is the loop which lets the user edit the settings.

```
792   set state 0
793   while {![info exists retCode]} {
794      if {[info exists opts(-addbuttons)]} then {
795         set script [dialog::handle $pages typeA $dial helpA\
              currentpage $optionL [list $opts(-addbuttons)] $main_buttons]
798      } else {
799         set script [dialog::handle $pages typeA $dial helpA\
                                       currentpage $optionL $main_buttons]
801      }
```

```
802       if {[catch $script err]} then {
```

The rest of this loop is simply for gracefully handling errors that occur when button scripts are evaluated.

```
803          global errorInfo
804          set errinfo $errorInfo
805          if {$opts(-debug)} then {
806             tclLog "Error in button script '$script'"
807             tclLog $err
808 ⟨*log1⟩
809             terminal::print_word emptyline "Error in button script\
                                                        $script" newline
811             terminal::print_word newline "Error: $err" newline
812             terminal::print_word newline "Error info: $errorInfo"\
                                                                 newline
813 ⟨/log1⟩
814          }
815          dialog::cleanup $dial
816          return -code 1 -errorinfo $errinfo "Error '$err' when\
                                                  evaluating button script."
818       }
819    }
```

The third part constructs the result to return (at normal returns). It should be observed that it uses `args` (rather than the contents of e.g. `pages`) to get the values in the original order. This ensures that the caller can interpret the flat list returned.

```
820    if {$retCode==0} then {
821       set retVal [list]
822       global dialog::mute_types
823       foreach pagearg $args {
824          set page [lindex $pagearg 0]
825          foreach item [lrange $pagearg 1 end] {
826             if {[lsearch -exact ${dialog::mute_types}\
                      [lindex [lindex $item 0] 0]] < 0} then {lappend retVal\
                             [dialog::valGet $dial "$page,[lindex $item 1]"]}
830          }
831       }
832    }
833    dialog::cleanup $dial
834    return -code $retCode $retVal
835 }
```

`dialog::make_paged` (proc) The `make_paged` procedure is similar to the `make` procedure, but its argument argument structure is slightly different, its return value is very different, and it does have a couple of features that `make` doesn't (such as adding or removing pages or items in a dialog). The basic syntax is the same

`dialog::make_paged` ⟨*option*⟩* {*page*}+

but here each {*page*} is a list with the structure

{*page name*} {*key–value list*} {*item list*}

and each {*item list*} in turn is a list of items, each of with are themselves lists and have the structure

{*key*} {*type*} {*name*} {*help*}$^{?}$

The return value is a list with the structure

$\big(\{$*page name*$\}$ {*key–value list*}$\big)^{+}$

and in both cases the {*key–value list*} has the format of a list returned by `array get`, i.e.,

$\big(\{$*key*$\}$ {*value*}$\big)^{*}$

Rather than (as with `make`) including the value of an item in its {*item*} list, that list contains a {*key*} which references a value stored in the {*key–value list*} of that page. The idea with this is that the input and output formats for values should be the same, so that the caller has little overhead in converting from one data format to another. The {*key–value list*} format is furthermore flexible in that is completely insensitive to changes that add, remove, or rearrange items within a page. Extra key–value pairs in the input are ignored and an empty string is substituted as value for pairs that are missing.

The ⟨*option*⟩s understood by `make_paged` are

`-ok` {*OK button title*}
`-cancel` {*cancel button title*}
`-title` {*dialog window title*}
`-defaultpage` {*name of default page*}
`-addbuttons` {*button list*}
`-width` {*dialog window width*}
`-debug` {*debug level*}
`-changedpages` {*var-name*}
`-changeditems` {*var-name*}

Those that are common with `make` work exactly the same. The `-changedpages` option specifies that the caller wants to know on which pages something was changed. The {*var-name*} is the name of a variable in the caller's local context which will be set to the list of (names of) pages where some item value was changed. The `-changeditems` option is similar, but here the variable will be set to a list with the structure

$\big(\{$*page name*$\}$ {*key list*}$\big)^{*}$

where the {*key list*}s are lists of the *keys* of items on that page whose values were changed.

```
836 proc dialog::make_paged {args} {
```

`make_paged` largely has the same local variables as `make`, but there are some additions. The major arrays are

`pageA` The index into this array is the name of a page. An entry contains the list of names of items on that page.

`typeA` The index into this array has the form ⟨*page*⟩ , ⟨*item*⟩, where ⟨*page*⟩ is the name of a page and ⟨*item*⟩ is the name of an item on that page. An entry contains the type of that item.

`keyA` The index has the same form as in the `typeA` array. An entry contains the {*key*} for that item.

`helpA` The index has the same form as in the `typeA` array. An entry contains the help text for that entry, but an item needs not have an entry in this array (it can be left unset).

There are a couple of additional scalar variables that are of interest.

`retCode`, `retVal` When the `retCode` variable is set, the dialog is logically closed and the procedure returns. If the variable is set to `0` then `make` executes a normal return and the returned value will be the list of item values. If the variable is set to anything else then that will used for the `-code` option of `return` and the returned value will be taken from the `retVal` variable, which must then be initalised.

`dial` This contains the reference string to use with `valGet`, `valSet`, and friends when accessing the values of items in the dialog.

`currentpage` This contains the name of the current page in the dialog.

`delta_pages` This is the list of all pages which have been added to or deleted from the dialog since it was called. The `add_page` and `delete_page` procedures both directly access this list. It is needed to get the information for the `-changedpages` and `-changeditems` correct.

`pages` This is a list of pages and items to show in the dialog. It is similar to the result of `array get pageA`, but the order of pages is as specified in the call and hidden pages are not included.

`opts(-addbuttons)` This is {*button list*} specified by the caller. Button scripts can modify this list to change the text on their button.

`state` This is initialized to `0` before the first time the dialog is shown and then the procedure leaves it alone. Button scripts may change it to keep track of what "state" (mostly: which items/pages are currently hidden) the dialog is in.

`optionL` The list of additional options to pass to `dialog::handle`.

The first part of `dialog::make_paged` processes the arguments.

```
837   set opts(-ok) OK
838   set opts(-cancel) Cancel
839   set opts(-title) ""
840   set opts(-width) 400
841   set opts(-debug) 0
842   getOpts {-title -defaultpage -ok -cancel -addbuttons -width -debug\
                                                  -changedpages -changeditems}
844   set dial [dialog::create]
```

The page arguments are interpreted by the `add_page` procedure. Since these pages aren't new in the sense that is relevant for the `delta_pages` list, that variable is reset afterwards.

```
   set pages [list]
   set delta_pages [list]
   foreach pagearg $args {
      eval [list dialog::add_page] $pagearg
   }
   set delta_pages [list]
   if {[info exists opts(-defaultpage)]} then {
      set currentpage $opts(-defaultpage)
   } else {
      set currentpage [lindex $pages 0]
   }
   set optionL [list -width $opts(-width) -title $opts(-title)]
   set main_buttons [list\
     [list $opts(-ok) "Click here to use the current settings."\
                                                        {set retCode 0}\
       $opts(-cancel) "Click here to discard any changes you've made to\
                    the settings." {set retCode 1; set retVal "cancel"}]\
     first right]
```

The second part is the loop which lets the user edit the settings.

```
   set state 0
   while {![info exists retCode]} {
      if {[info exists opts(-addbuttons)]} then {
         set script [dialog::handle $pages typeA $dial helpA\
            currentpage $optionL [list $opts(-addbuttons)] $main_buttons]
      } else {
         set script [dialog::handle $pages typeA $dial helpA\
                                       currentpage $optionL $main_buttons]
      }
      if {[catch $script err]} then {
```

The rest of this loop is simply for gracefully handling errors that occur when button scripts are evaluated.

```
         global errorInfo
         set errinfo $errorInfo
         if {$opts(-debug)} then {
            tclLog "Error in button script '$script'"
            tclLog $err
⟨*log1⟩
            terminal::print_word emptyline "Error in button script\
                                                        $script" newline
            terminal::print_word newline "Error: $err" newline
            terminal::print_word newline "Error info: $errorInfo"\
                                                                newline
⟨/log1⟩
         }
         dialog::cleanup $dial
```

```
888         return -code 1 -errorinfo $errinfo "Error '$err' when\
                                                  evaluating button script."
890      }
891   }
```

The third part is as in `make` responsible for constructing the result to return (at normal returns). Unlike the case with `make`, the return value covers only the items currently in `pages`. This part is also responsible for constructing the lists of changed pages and items. Two important variables in this are `cS` and `cA`. `cS` is an array which is used to test whether a certain item has been changed (via `valChanged`), but the only thing that matters is whether an entry has been set or not. `cA` is an array indexed by page name, whereas the entries are lists of keys of items on that page which have been changed.

```
892   if {$retCode==0} then {
893      set retVal [list]
894      global dialog::mute_types
895      foreach page $delta_pages {
896         foreach name $pageA($page) {
897            lappend cA($page) $keyA($page,$name)
898         }
899      }
900      foreach item [dialog::changed_items $dial] {set cS($item) ""}
901      newforeach {page items} $pages {
902         set res [list]
903         foreach name $items {
904            set T "$page,$name"
905            if {[lsearch -exact ${dialog::mute_types}\
                                       [lindex $typeA($T) 0]] < 0} then {
907               lappend res $keyA($T) [dialog::valGet $dial $T]
908               if {[info exists cS($T)]} then {
909                  lunion cA($page) $keyA($T)
910               }
911            }
912         }
913         lappend retVal $page $res
914      }
915      if {[info exists opts(-changedpages)]} then {
916         upvar 1 $opts(-changedpages) cp
917         set cp [array names cA]
918      }
919      if {[info exists opts(-changeditems)]} then {
920         upvar 1 $opts(-changeditems) ci
921         set ci [array get cA]
922      }
923   }
924   dialog::cleanup $dial
925   return -code $retCode $retVal
926 }
```

`dialog::add_page` (proc) The `add_page` procedure can be called from within the `make_paged` procedure to add a new page to the dialog. The syntax is

`dialog::add_page` {*page name*} {*key–value list*} {*item list*} {*position*}?

Here the {*page name*}, {*key–value list*}, and {*item list*} coincide with those parts of a {*page*} argument of `make_paged`.

`add_page` works by modifying the arrays `typeA`, `keyA`, `helpA`, and `pageA`, and the lists `pages` and `delta_pages` in the caller's local context. It also uses the value in the `dial` variable there as an argument to `valSet`. All of these variables are assumed to function as they do in the `make_paged` procedure.

The {*position*} argument can be used to specify where in the `pages` list that the new page should be inserted. It defaults to `end`, which puts the new page last. Otherwise the argument should be numeric: `0` means put first, `1` means put second, `2` means put third, etc.

```
927 proc dialog::add_page {page keyvalL itemsL {pos end}} {
928    upvar pageA pageA typeA typeA helpA helpA keyA keyA dial dial pages\
                                                 pages delta_pages delta_pages
930    array set local $keyvalL
931    set pageA($page) [list]
932    lunion delta_pages $page
933    foreach item $itemsL {
934       set key [lindex $item 0]
935       set name [lindex $item 2]
936       set keyA($page,$name) $key
937       if {[info exists local($key)]} then {
938          dialog::valSet $dial $page,$name $local($key)
939       } else {
940          dialog::valSet $dial $page,$name ""
941       }
942       set typeA($page,$name) [lindex $item 1]
943       if {[llength $item]>3} then {
944          set helpA($page,$name) [lindex $item 3]
945       }
946       lappend pageA($page) $name
947    }
948    if {$pos!="end"} then {
949       set pages [linsert $pages [expr {2*$pos}] $page $pageA($page)]
950    } else {
951       lappend pages $page $pageA($page)
952    }
953 }
```

`dialog::delete_pages` (proc)

In one sense, this procedure does the opposite of `add_page`, but it can be used to achieve different effects as well. Basically it takes a list of page names and items, in the format for the first argument of `handle`, and returns the same list with some pages removed. The syntax is

`dialog::delete_pages` {*pages*} {*delete-list*} {*deleted-var*}?

where the {*delete-list*} is the list of names of pages to remove. {*deleted-var*} is, if it is given, the name of a variable in the caller's local context containing a list of page

names. The deleted pages are then unioned with this list. The most common value for {*deleted-var*} is `delta_pages`.

```
proc dialog::delete_pages {pages deleteL {deletedvar {}}} {
   set res [list]
   if {$deletedvar!=""} then {upvar 1 $deletedvar diffL}
   newforeach {page items} $pages {
      if {[lsearch -exact $deleteL $page] == -1} then {
         lappend res $page $items
      } else {
         lunion diffL $page
      }
   }
   return $res
}
```

2.6 Dialog items and preferences

In the classical preferences dialogs, all items were preferences and it was the preference data structures that determined the type of the items. As this is not the case with the new dialogs, there is a need for constructing a dialog item corresponding to a preference.

`dialog::prefItemType` (proc)

The `dialog::prefItemType` preference returns the dialog item type that corresponds to the type of a specified preference. The syntax is

`dialog::prefItemType` {*pref. name*}

This procedure needs to be improved, as it currently only recognises a few preference types.

```
proc dialog::prefItemType {prefname} {
    global flag::list
    if {[info exists flag::list($prefname)]} {
        set l [set flag::list($prefname)]
        if {[regexp "index" [lindex $l 0]]} {
            set res [list menuindex]
        } else {
            set res [list menu]
        }
        lappend res [flag::options $prefname]
    } elseif {[regexp "Colou?r$" $prefname]} {
        return "colour"
    } elseif {[regexp "Mode$" $prefname]} {
        return "mode"
    } else {
        return "var"
    }
}
```

⟨/core⟩

2.7 To do

The generic dialogs code has now seems to have reached a rather mature state. Certainly the details can be polished, new types can be added, and some procedures (such as `dialog::prefItemType`) should be improved, but on the whole they can do everything that we seem to need.

What needs to be improved is instead the *Alphatk* interface for setting up and managing dialogs. Right now it is both complicated (involving a large number of callbacks) and highly specialized (making assumptions that are only valid for a few types), which is most unfortunate. Obviously the interface should rather be simple and general (and how it ever go to be anything else is a source of quite some amazement for me), but achieving that requires that the whole thing is thoroughly thought through rather than pieced together.

3 Examples

This section contiains a couple of examples of how the generic dialogs procedures can be used. All code in the examples module can be found in the file `Dialogs-Examples.tcl`.

`test_make` (proc) The `test_make` procedure is used in the examples below to facilitate presentation of the results. The syntax is

`test_make` {*paged*} {*script*}

where {*script*} is a script that the procedure evaluates and presents the result (or error) of in a new window with the title '`dialog make result`'. If {*paged*} is `0` then the result of the script interpreted as a list and each item is put on a line of its own; this is suitable when the last command in the script was a `dialog::make`. If {*paged*} is `1` then the result is instead formatted so that it looks good if it was generated by `dialog::make_paged`.

985 ⟨∗examples⟩
```
986 proc test_make {format script} {
987    set code [catch $script res]
988    new -n "dialog make result" -info [if {$code} then {
989         set t "Error: $res"
990         global errorInfo
991         append t \n "errorInfo:\n" $errorInfo
992      } elseif {$format} then {
993         set L [list]
994         newforeach {page keyvals} $res {
995            set t \n
996            newforeach {key value} $keyvals {
997               append t "  [list $key $value]\n"
998            }
999            lappend L $page $t
1000        }
1001        set L
1002     } else {
1003        join $res \n
1004     }]
1005 }
```

3.1 An elementary example

This example creates a single-page dialog with a selection of TextEdit item types on, using `dialog::make`. The title '`Example dialog 1`' is only visible in *Alphatk*.

```
test_make 0 {
   dialog::make -title "Example dialog 1" [list "TextEdit types"\
       [list var "A 'var'" "Some text"]\
       [list var "A 'var' with a long name" "Again some text"]\
       [list var "A 'var' with a very very very long name" short]\
       [list var2 "A 'var2'" "This piece of editable text is rather\
                                          long, two lines come in handy."]\
       [list static "A 'static'" "This text cannot be edited."]\
       [list password "A 'password'" Swordfish]\
       [list password "A 'password' with a very long title" Swordfish]\
     ]
}
```

The `static` item is formatted like a `var` item, but the value is put in a static text atom, not a TextEdit atom. Neither is it returned by the procedure.

The same example dialog using `dialog::make_paged` looks instead as follows. Note that the order of items in the {*key–value list*} needs not be the same as that in the {*item list*}.

```
test_make 1 {
   dialog::make_paged -title "Example dialog 1" [list "TextEdit types"\
       [list a "Some text" b "Again some text" c short d "This piece of\
               editable text is rather long, two lines come in handy." e\
                  Swordfish f Swordfish g "This text cannot be edited."]\
       [list\
         [list a var "A 'var'"]\
         [list b var "A 'var' with a long name"]\
         [list c var "A 'var' with a very very very long name"]\
         [list d var2 "A 'var2'"]\
         [list g static "A 'static'"]\
         [list e password "A 'password'"]\
         [list f password "A 'password' with a very long title"]\
       ]\
     ]
}
```

Clearly `dialog::make` is more suitable for such a small dialog. `dialog::make_paged` is most convenient when the {*item list*} has already been constructed. This is for example the case in the `dialog::editGroup` procedure (see below).

3.2 A smorgasbord of types

The generic dialog procedures provide a large variety of item types. The following dialog demonstrates all the visible item types currently defined. Note that packages can define their own types simply by adding elements to the `dialog::simple_type` or `dialog::complex_type` arrays.

```
test_make 0 {
```

```
set page1 [list "Text types"]
lappend page1 [list var "A 'var'" "Some text"]
lappend page1 [list var2 "A 'var2'" "This piece of editable text is\
                                  rather long, two lines come in handy."]
lappend page1 [list text "This is a 'text' item. It can be used for\
       including a paragraph or two of text inside the dialog." "This\
                                                        value is ignored!"]
lappend page1 [list password "A 'password'" No]
lappend page1 [list static "A 'static'" "This is static text"]

set page2 [list "Files and the like"]
global HOME
lappend page2\
          [list file "A 'file'" [file join $HOME Help "Alpha Manual"]]
lappend page2 [list folder "A 'folder'" $HOME]
lappend page2\
               [list url "An 'url'" "http://alphatcl.sourceforge.net/"]
```

appspecs are a bit tricky to give examples of since they are quite platform-dependent.

```
global alpha::platform
if {${alpha::platform}=="alpha"} then {
   lappend page2 [list appspec "An 'appspec'" 'ALFA']
   set s 'WIsH'
   if {[catch {nameFromAppl $s} t]} then {
      set t $s
   } elseif {[regexp -nocase wish $t]} then {
      set t $s
   } else {
      set t [glob -nocomplain -dir [file dirname $t] *Wish*]
      if {[llength $t]} then {set t [lindex $t 0]} else {set t $s}
   }
   lappend page2 [list appspec "Another 'appspec'" $t]
} else {
   global texSig
   lappend page2 [list appspec "An 'appspec'" $texSig]
}
lappend page2 [list searchpath "A 'searchpath'"\
                     [glob -nocomplain -types d -join $HOME {[E-H]*}]]

set page3 [list "Menus and the like"]
lappend page3 [list {menu {One two three}} "A 'menu'" two]
lappend page3 [list {menuindex {nul odin dva tri tjetyre pat sjest}}\
                                                    {A 'menuindex'} 2]
lappend page3 [list colour "A 'colour'" green]
lappend page3 [list mode "A 'mode'" TeX]
lappend page3 [list [list subset\
        [list "Charlie Chaplin" Saturn toothbrush {"yeah, yeah"} 19]]\
                                     {A 'subset'} [list toothbrush 19]]
lappend page3 [list modeset "A 'modeset'" [list TeX Bib Mf]]

set page4 [list "Miscellaneous types"]
lappend page4 [list flag "A 'flag'" 1]
```

```
1086    lappend page4 [list [list multiflag\
                     [list AlphaPrefs Developer Examples Help Tcl Tools]]\
                               {This is a 'multiflag'} [list 0 1 1 0 1 0]]
1090    lappend page4 [list binding "A 'binding'" /Q<O]
1091    lappend page4 [list date "A 'date'" [now]]
1092    lappend page4\
              [list thepage "This item is invisible" "This value is ignored"]
1094    dialog::make $page1 $page2 $page3 $page4
1095 }
```

3.3 Button manœuvres

Another nice feature with the generic dialog interface is the ability to change the name of the OK and Cancel buttons, or to add extra buttons with new functionality. The next example demonstrates this; it is intended as a login dialog for some fancy protocol where the password depends on the time as well as on the user name.

```
1096 test_make 0 {
1097    set page [list "Login parameters"]
1098    lappend page [list static "Curent time" [join [mtime [now] long]]]
1099    lappend page [list var "User name" ""]
1100    lappend page [list password "Password" ""]
1101    dialog::make -ok Login\
1102      -addbuttons [list "Update time"\
               {This button updates the current time shown in the dialog.}\
                      {dialog::valSet $dial "Login parameters,Curent time"\
                                          [join [mtime [now] long]]}]\
1106      $page
1107 }
```

The `valSet` procedure updates the value of the `static` item.

The next example shows how one can use a button to toggle between a "basic settings" and "complete settings" state of a dialog. All values are always reported back, but they are not necessarily shown.

```
1108 test_make 0 {
1109    set page [list "Email settings"]
1110    lappend page [list var "Name" "Jane Doe"]
1111    lappend page [list var "Address" "Jane.Doe@nowhere.edu"]
1112    lappend page [list var "Organisation" "University of Nowhere"]
1113    lappend page [list [list hidden var] "POP server" mail.nowhere.edu]
1114    lappend page [list [list hidden var] "SMTP server" smtp.nowhere.edu]
1115    dialog::make -addbuttons\
          [list "Full settings" {Toggles between basic and full settings.} {
1117       if {!$state} then {
1118          dialog::show_item "Email settings" "POP server"
1119          dialog::show_item "Email settings" "SMTP server"
1120          set opts(-addbuttons)\
                        [lreplace $opts(-addbuttons) 0 0 "Basic settings"]
1122          set state 1
1123       } else {
```

```
            dialog::hide_item "Email settings" "POP server"
            dialog::hide_item "Email settings" "SMTP server"
            set opts(-addbuttons)\
                        [lreplace $opts(-addbuttons) 0 0 "Full settings"]
            set state 0
         }
      }] $page
}
```

Another way of hiding items from the user is to hide the entire page on which they reside.

```
test_make 0 {
   set page1 [list "Basic email settings"]
   lappend page1 [list var "Name" "Jane Doe"]
   lappend page1 [list var "Address" "Jane.Doe@nowhere.edu"]
   lappend page1 [list var "Organisation" "University of Nowhere"]
   set page2 [list "Advanced email settings"]
   lappend page2 [list var "POP server" mail.nowhere.edu]
   lappend page2 [list var "SMTP server" smtp.nowhere.edu]
   dialog::make -addbuttons\
       [list "Full settings" {Toggles between basic and full settings.} {
        if {!$state} then {
           set currentpage "Advanced email settings"
           lappend pages $currentpage $pageA($currentpage)
           set opts(-addbuttons)\
                       [lreplace $opts(-addbuttons) 0 0 "Basic settings"]
           set state 1
        } else {
           set currentpage "Basic email settings"
           set pages [list $currentpage $pageA($currentpage)]
           set opts(-addbuttons)\
                        [lreplace $opts(-addbuttons) 0 0 "Full settings"]
           set state 0
        }
     }] -hidepages [list "Advanced email settings"] $page1 $page2
}
⟨/examples⟩
```

3.4 Editing named configurations

It is not uncommon that the settings for something can be collected in a "configuration" and that the user can have several such configurations stored simultaneously (even though only one is used for each operation); the filesets and (more recently) the SourceForge menu projects are both examples of this. Orinially for use for the latter of these, Vince wrote a generic procedure `dialog::editGroup` which presents a list of configurations as a multipage dialog (one page per configuration) in which all pages have the same set of items, but usually different values. Furthermore the dialog contains two extra buttons: one for adding a new configuration and one for deleting a configuration.

The original definition used a (sort of) hacked `dialog::make`, but the new implementation below uses `dialog::make_paged` instead. Indeed, that there should be

an easy implementation of `editGroup` using the latter was the main design goals for `make_paged`.

`dialog::editGroup` (proc) The `editGroup` procedure lets the user edit configurations stored in an array in the local context of the caller and returns the list of configurations that were changed. The syntax is

`dialog::editGroup` ⟨*option*⟩$^+$ {*item*}$^+$

The {*item*}s are `make_paged` style item descriptions, i.e., lists with the format

{*key*} {*type*} {*name*} {*help*}$^?$

The currently supported ⟨*option*⟩s are

`-array` {*array name*}
`-current` {*current configuration name*}
`-delete` {*ask first?*}
`-new` {*new conf.-cmd*}
`-title` {*title*}

The `-array` option specifies the name of the array in which the configurations are stored. Indices into this array are configuration names and the entries contain key–value lists that give the entries of the array. **Note** that the `-array` option isn't optional at all, but mandatory.

The `-current` option can be used to specify at which configuration the dialog should be opened. The `-delete` option specifies that the dialog should have a Delete button. If the {*ask first?*} is anything but `dontask` then the user is asked for confirmation before the current configuration is actually deleted. The `-new` option specifies that the dialog should have a New button. The {*new conf.-cmd*} is a script that is executed when the user clicks the New button. It should return either a list with the structure

{*new config. name*} {*key–value list*}

or, if the user decides not to create a new configuration, an empty string. The `-title` option specifies a title for the dialog; this defaults to Edit.

```
1160 ⟨∗core⟩
1161 ⟨notinstalled⟩auto_load dialog::getAKey
1162 proc dialog::editGroup {args} {
1163     global dialog::ellipsis
1164     set opts(-current) ""
1165     set opts(-title) "Edit"
1166     getOpts {-array -title -current -new -delete}
1167     upvar $opts(-array) local
```

After processing arguments, the first task is to construct the {*page*} arguments to `make_paged`.

```
1168     set dialog [list]
1169     foreach item [lsort -ignore [array names local]] {
1170         lappend dialog [list $item $local($item) $args]
1171     }
```

The the `-addbuttons` option, if any, to `make_paged` are constructed.

```
    set buttons [list]
    if {[info exists opts(-delete)]} {
        if {$opts(-delete)=="dontask"} then {
           lappend buttons "Delete" "Click here to delete this page"\
```

As there is no need to embed variable data into the script for the Delete button, it is easiest to give it explicitly.

```
             {set pages [dialog::delete_pages $pages\
                                        [list $currentpage] delta_pages]}
        } else {
           lappend buttons "Delete${dialog::ellipsis}" "Click here to\
                                                        delete this page" {
                if {[dialog::yesno "Are you sure you want to delete\
                                                       '$currentpage'?"]} {
                    set pages [dialog::delete_pages $pages\
                                         [list $currentpage] delta_pages]
                }
             }
        }
    }
    if {[info exists opts(-new)]} {
        lappend buttons "New${dialog::ellipsis}" "Click here to add a\
              new page" [list dialog::editGroupNewPage $args $opts(-new)]
```

With the script for the New button, things are different: both the layout of a page and the script which generates the contents for new pages have to be embedded into the button script. It is then easiest to put all processing in a helper procedure and restrict the button script to call that helper.

```
    }
    if {[llength $buttons]} {
        set buttons [list -addbuttons $buttons]
    }
    set res [eval [list dialog::make_paged -title $opts(-title)\
       -defaultpage $opts(-current) -changedpages mods] $buttons $dialog]
```

If the user did not Cancel the dialog, the array specified by the `-array` option is cleared and the new data returned by `make_paged` are stored into it instead. It is necessary to clear the array if some page has been deleted.

```
    unset local
    array set local $res
    return $mods
}
```

`dialog::editGroupNewPage` (proc)

The `editGroupNewPage` procedure is a helper for `editGroup`.

```
proc dialog::editGroupNewPage {layout cmd} {
   set T [eval $cmd]
   if {![llength $T]} then {return}
   newforeach {page items} [uplevel 1 {set pages}] {
      if {$page==[lindex $T 0]} then {
```

```
1208          alternote "That name is already in use!"
1209          return
1210       }
1211    }
1212    uplevel 1 [concat dialog::add_page $T [list $layout]]
1213    uplevel 1 [list set currentpage [lindex $T 0]]
1214 }
1215 ⟨/core⟩
```

References

[1] Jesper Blommaskog: *Is white space significant in Tcl*, The Tcl'ers Wiki page **981**; http://mini.net/tcl/981.html.

[2] Frédéric Boulanger *et al.*: *Driving external applications from Alpha*, discussion thread on the AlphaTcl developers mailing list, October 2001.

[3] Sharon Everson *et al.*: *Inside Macintosh – Macintosh Toolbox Essentials*, Addison–Wesley, 1992; ISBN 0-201-63243-8. Also available as PDF at http://www.devworld.apple.com/techpubs/mac/pdf/MacintoshToolboxEssentials.pdf and in HTML at http://www.devworld.apple.com/techpubs/mac/Toolbox/Toolbox-2.html.

[4] Lars Hellström: *The tclldoc package and class*; CTAN:macros/latex/contrib/supported/tclldoc/tclldoc.dtx. Note: That is the proper home for tclldoc, but I've been so busy with other things that I haven't gotten around to uploading it to CTAN yet. A recent version can alternatively be found in Developer/texmf of a complete AlphaTcl tree.

[5] Frank Mittelbach, Denys Duchier, Johannes Braams, Marcin Woliński, and Mark Wooding: *The DocStrip program*, The LaTeX3 Project; CTAN:macros/latex/base/docstrip.dtx.

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition. By tradition there should also be a lot of numbers in roman that refer to the code lines where the entry is used, but unfortunately I don't have a convenient way of cross-referencing Tcl code to generate that information.

T

V

W