dialog plugin for PowerPro: version 1.19: 21 January 2009

A PowerPro Plugin to Construct and Run Dialogs

by Alan Campbell

1.0 Overview

Using this plugin you can build dialogs from within PowerPro scripts. You can run the dialogs, modify them as they run, and get feedback from then using event-handling PowerPro scripts.

You can also use some dialog services to access and modify controls in dialogs other than those made by this plugin.

1.1 This Document

There are three versions of this document, with the same content. There's an RTF file, which looks nice in Word but is something like four meg in size; and there's a compiled help (CHM) document, which is much smaller if somewhat uglier; and there's a pdf, with bookmarks for each section heading.

In my experiments I've found the RTF file doesn't display correctly in anything but Word (not Keynote, even Wordpad: you've think Microsoft could at least get their rtf engines consistent). So if you don't have Word, better use the chm file.

Both documents have extensive hyperlinks. The table of contents at the front of each document is a set of them.

The chm file has no index.

Sheri Pierce made an indexed pdf out of version 1.07 of this document, available at http://tech.groups.yahoo.com/group/power-pro/files/Plug-ins_and_add-ons.

1.2 What's New In This Version

- Changed parameters to choose_font allowing a *<font_spec>* instead of a <*target*>, and variable name to be specified to set and return colour. choose_font now used in controlFontChanger.powerpro.

- set_font applied to <*window_handle*> now returns the HFONT associated with the window after set_font runs.

- Added a variant to clear service allowing deletion of font resource referred to by an HFONT

- Fixed an error in set_font which caused PowerPro to crash if applied to a window with the system font. Updated documentation for set_font to describe more accurately when it works and when it doesn't when applied to a <*window_handle*>.

- Fixed a bug in choose_font which caused to initialise the font picker dialog with garbage when <*target*> is a <*window_handle*>.

- Added to documentation for get_value(<*window_handle*>, "font") variant; usually doesn't work on anything but controls

- Added variant of get_value with <*property*> "fonth" or "fonthandle"

- set_response now has <*mouse_event*>s "enter" and "exit" (to specify a response to make when mouse enters a control's window, or leaves it).

- For statics, if you invoke set_response to set a response to a <*mouse_event*>s; or set_colour with a <*mouse_state*>*;* The "notify" (SS_NOTIFY) style will automatically be added to the control's styles. Example of static with both a set_colour and a set_response to make a static with a clickable url can be found in dialogPluginDemo7.powerpro.

- Added pdf version of documentation

- Fixed memory leaks in GDI and other resources (thanks to Sheri Pierce for locating them). There are more leaks, will try to plug them in the next version.

- Tooltips don't work in XP if the powerpro manifest file (powerpro.exe.manifest) isn't present. Will try and fix.

1.3 Requirements

Requires PowerPro version 3.4 or later. Test scripts require at least 4.8 RC3. They work in standard configuration.

Some aspects of service calls are subject to fewer restrictions (see for instance here or here) if you have PowerPro version 4.3.10.

If the *<lParam>* of send_message is a handle to a composite (array or struct), you'll have needed the dll plugin to generate that handle.

If you create a dialog from, or export one to, an ini file, you'll need the ini plugin (found at http://tech.groups.yahoo.com/group/power-pro/files/Plug-ins_and_add-ons/0_Registry and Ini Access/). You need version 1.52 or later

The test script **dialogPluginDemo3.powerpro** uses the reg plugin; you can edit it so it doesn't.

dialogPluginDemo6.powerpro uses the odbc and/or sqlite plugins, and the dll plugin (although the latter use isn't essential and can be edited out).

dialogPluginDemo7.powerpro uses clock.avi in your %winDir folder, which I think is there for all versions of windows. Let me know if it innit. It also includes a static with both a set_colour and a set_response to make a clickable hotlink.

The test scripts **regex\regexDialog.powerpro** and **regex\regexDialogScintilla.powerpro** require the regex and dll plugins, and **regexDialogScintilla.powerpro** requires scilexer.dll. If you want to save or load format settings to an ini file you'll need the ini plugin (found at http://tech.groups.yahoo.com/group/power-pro/files/Plug-ins_and_add-ons/0_Registry and Ini Access/).

dialog.dll *doesn't* require MFC support (e.g. MFC42.DLL).

Tested on W2000 sp4.

2.1 Reporting Bugs, Requesting Enhancements

This plugin is complicated. I have by no means tested every possible path through the code, so bugs are likely.

If you hit any problems with this plugin (or any of my plugins, for that matter), it'd be helpful if you reported them via the PowerPro forum (http://groups.yahoo.com/group/power-pro/) in a message with a clear subject line (maybe: "DIALOG PLUGIN: apparent error in:…."). I don't read everything in the forum, but I will see anything flagged with an obvious header. Please include a copy of the script causing problems, and state which version of PowerPro and of the dialog plugin you're using.

2.2 Related plugins

If you create a dialog from, or export one to, an ini file, you'll need the ini plugin (found at http://tech.groups.yahoo.com/group/power-pro/files/Plug-ins_and_add-ons/0_Registry and Ini Access/). You need version 1.52 or later.

You can send messages to controls to modify or query them using the dialog.send_message service. The *<lParam>* of **send_message** can be a handle to a composite (i.e. to an array or struct; e.g., most messages that you can send to the month-calendar date picker control require the use of pointers to SYSTEMTIME structures). To get those handles, you'll have needed the dll plugin (found at http://tech.groups.yahoo.com/group/power-pro/files/Plug-ins_and_add-ons/). There's an example in the test script **dialogPluginDemo6.powerpro**.

Using the dll plugin is not for the faint-hearted (though if you've got to the point where you even know what message you want to send, you're probably knowledgeable enough to do the right thing in the dll plugin). If you need help, ask in the PowerPro forum.

Some example scripts require other plugins; see below, Section 7.0 "Testing".

3.0 File list

dialog.dll

docs\dialogPluginReadme.rtf
docs\dialogPluginReadme.chm
docs\dialogPluginReadme.pdf
docs\dialog-related_defines.txt
docs\scintilla-related_defines.txt
docs\dialogPluginFunctions.txt

scripts\dialogPluginDemo1.powerpro
scripts\dialogSubordinate.powerpro
scripts\dialogPluginDemo2.powerpro
scripts\dialogPluginDemo3.powerpro
scripts\dialogPluginDemo4.powerpro
scripts\dialogPluginDemo5.powerpro
scripts\dialogPluginDemo6.powerpro
scripts\dialogPluginDemo7.powerpro
scripts\dialogPluginDemo8.powerpro
scripts\browse_for_file.powerpro

scripts\dialogPluginDemoFromConfigFile.powerpro
scripts\dialogPluginDemo.txt
scripts\dialogPluginDemo.ini

scripts\dialogPluginDemo.ico

scripts\resources\eye.bmp
scripts\resources\eye.gif
scripts\resources\eye.ico
scripts\resources\atlas.csv
scripts\resources\atlas.dbf
scripts\resources\atlas.db
scripts\resources\test.swf

scripts\regex\regexDialog.ico
scripts\regex\regexDialog.powerpro
scripts\regex\regexDialog.txt
scripts\regex\regexDialogScintilla.powerpro
scripts\regex\regexDialogScintilla.txt

scripts\controlFontChangerTest.powerpro
scripts\controlFontChanger.powerpro

scripts\notificationDialog.powerpro

scripts\dialogViewer.powerpro
scripts\dialogViewerTest.powerpro

scripts\skeletonDialog\skeletonDialog.powerpro
scripts\skeletonDialog\skeletonDialog.txt
scripts\skeletonDialog\skeletonDialog.ini
scripts\skeletonDialog\skeletonDialog.ico
scripts\sample.ini

scripts\dialogPluginDemoNonNative.powerpro

DialogEditor.exe
DialogEditor.ini
dialogEditorToUltraEdit.powerpro
dialogEditorToNotepad.powerpro
dialogEditorToWordpad.powerpro

dialog.ini

4.0 Installation

The dialogPluginX.XX.zip archive is organised in folders. If you unzip it within the Powerpro installation folder and preserve folder structure, the plugin dll will end up in your plugins folder, and the test scripts in your scripts folder. You may however to kake control of installation by unzipping the archive somewhere else, and manually moving stuff around. If you do that, copy dialog.dll from the dialogPluginX.XX.zip archive into your PowerPro directory, or into its Plugins subfolder. If you want to provide an initial configuration of the plugin (see Section 9.1), edit dialog.ini and put it in the folder pointed to by pprofolder; or add its edited contents to plugins.ini in the same folder.

The **.**powerpro scripts can go wherever you want. If you want the test scripts dialogPluginDemo*.powerpro to work correctly, keep them in the same folder as they unzip in; in particular make sure dialogPluginDemo.txt, dialogSubordinate.powerpro and the resources folder are with them in the same folder.

If you want to use the dialog editor DialogEditor.exe, that file and DialogEditor.ini must go in the same folder. If you want to switch from dialog to text editor and back again, an appropriate script to glue to your text editor (e.g. dialogEditorToUltraEdit.powerpro, dialogEditorToNotepad.powerpro, dialogEditorToWordpad.powerpro) must be on your path and may need to be customised for your text editor (you can rename and relocate the scripts if you want to: see the previous link).

If you want dialog.help to automatically open the chm/rtf documentation, you may want to put one or the help files in the plugins folder and maybe rename them. See section 10.32.

dialogPluginFunctions.txt can be appended to pprofunctions.txt supplied in the PowerPro distro (or included in it, using

include <*path to*>\dialogPluginFunctions.txt)

It can then be accessed either as part of pprofunctions.txt or on its own as a file menu (e.g. using a hot key associated with

*keys {filemenu <*path to*>\pproFunctions.txt})

dialog.dll is about 150k in size. If you want, you can use upx (http://upx.sourceforge.net/) to compress it down to a third of that. Read about the pros and cons at http://en.wikipedia.org/wiki/UPX.

5.0 Uninstall

Remove all files listed in the above section ("3.0 File list") from wherever they went.

6.0 Acknowledgements

Thanks to Bruce Switzer for help sorting out modal/non-modal dialog interactions with PowerPro, and suggestions for features. In fact this plugin was his idea in the first place.

I nicked some code for some dialog features from Chris Mallett's autoHotKey source code (www.autohotkey.com).

I nicked descriptions of styles from either the appropriate MSDN page or Chris Mallett's autoHotKey chm file.

Got code to load a gif or jpeg from Michael Chourdakis (http://www.codeguru.com/cpp/g-m/bitmap/article.php/c4935/)

test.swf nicked from http://www.arseiam.com/math_frames.htm.

Code to resize controls when dialog resized cannibalised from DlgResizeHelper

(http://www.codeguru.com/cpp/w-d/dislog/resizabledialogs/article.php/c1913/) by Stephan Keil.

Stole a good chunk of code for owner-draw buttons from the CbuttonST project (http://www.codeproject.com/buttonctrl/cbuttonst.asp) by Davide Calabro, and from "Owner-draw icon buttons in plain C" (http://www.codeproject.com/useritems/odib.asp) by Bruno Challier.

Thanks to Sheri Pierce for suggesting and testing of the regex dialog scripts. And she did a pdf of the 1.07 version of this document (at http://tech.groups.yahoo.com/group/power-pro/files/Plug-ins_and_add-ons). She also spotted memory leaks in GDI and other resources prior to the 1.19 version.

And to Detlef Leidinger for suggesting and testing the define_set service, and for suggesting "+" modified dimensions for dialog dimensions.

And to Andreas Mokros for suggestions on control resizing options.

7.0 Testing and Sample Scripts

To exercise the plugin:

Put *.powerpro, *.ico and *.txt files in distribution in the same folder. The main demo scripts are dialogPluginDemoN.powerpro, N varying from 1 to 6.

Then:

Either:

- run it with a command (menu item?) in PowerPro like

  *Script RunFile dialogPluginDemoN *or*
  .dialogPluginDemoN

Or:

- Put dialogPluginDemoN.powerpro anywhere and double click on it (as long as PowerPro is already running). This didn't initially work on my machine until I manually set up an association with **.**powerpro, but if your Powerpro installation went correctly it should work for you.

dialogPluginDemo*.powerpro scripts output to the Debug window.

All dialogPluginDemo*.powerpro scripts make extensive use of the PowerPro cb() function which Bruce added in version 4.4.09, so if you're running a version prior to that, you have some editing to do.

7.1 Sample Scripts Included with the Plugin Distribution

All sample scripts assume index base is zero, call set_base to make that so, and resotre index base when they exit if required.

dialogPluginDemo1.powerpro illustrates edit controls, buttons, statics, list controls, combo controls, images in statics, 3-state check controls, group controls, scrollbars, radio buttons, specified notifications, call backs, tooltips, fonts, the font common dialog. and right-click menus (see the "Text To Debug Win" button). It also runs **dialogSubordinate.powerpro** as a modal sub dialog. And illustrates enforcment of a single instance of a dialog from a script.

dialogPluginDemoFromConfigFile.powerpro creates a dialog from the dialog definition file dialogPluginDemo.txt. If you edit line 3 of the script, you can get it to use dialogPluginDemo.ini instead.

dialogPluginDemo2.powerpro illustrates use of system icons and the icons built into the plugin; and the progress, spinner, slider date-time, status bar and month calendar controls, resizing dialogs and the controls within them, and use of set_position.

dialogPluginDemo3.powerpro illustrates activeX controls and font specifications. Make sure resources\test.swf is in same folder as the script. It assumes the existence of Office Web Components (aka MS Office) and Shockwave/ Flash activeX components. If you don't have them, the dialog may refuse to load (I hope it will just load with blank controls, but can't test). If you're lacking a component, comment out the relevant bits of the script. **dialogPluginDemo3.powerpro** uses the reg plugin to try to work out which version of Office you have; if you'd prefer not to use the reg plugin, edit the script to set the variable progID to the correct value.

dialogPluginDemo4.powerpro illustrates tab controls, tree view controls and the use of the make_ctrl_handle service. Make sure resources\test.swf is in same folder as the script.

dialogPluginDemo5.powerpro illustrates tree view, list view and statusbar controls and the use of the make_ctrl_handle service.

dialogPluginDemo6.powerpro illustrates the list view control used as a way of displaying contents of a database. Requires the odbc and/or sqlite plugins (found at http://tech.groups.yahoo.com/group/power-pro/files/Plug-ins_and_add-ons/0_Interfaces/Database_Interfaces/). The resources\atlas.* data sources must be in the same folder as this script. atlas.db is a sqlite database, the other atlas.* files are used as ODBC sources. **dialogPluginDemo6.powerpro** also requires the dll plugin and illustrates use of dll.dereference to get data related to a notification, although if you don’t want to use the dll plugin, you can just edit out any calls to dll services.

dialogPluginDemo7.powerpro illustrates positioning of controls and buttons with images, using various position modifiers on dimensions; and use of an Animation control.

dialogPluginDemo8.powerpro illustrates the dialog style "draggable", the button image-related style "imgfill" and use of a <*script_to_call*> for a tooltip

controlFontChanger.powerpro illustrates manipulation of properties of dialogs other than those generated by the dialog plugin.

Run **controlFontChangerTest.powerpro** to show a Powerpro inputDialog, then run **controlFontChanger.powerpro** to change font size of the edit box in that input dialog.

notificationDialog.powerpro demonstrates use of a dummy dialog to pop up a balloon notification tooltip from the taskbar.

browse_for_file.powerpro demonstrates the use of the browse_for_file service.

regexDialog.powerpro illustrates a use of the rich edit control (in the results edit box, in "match" or "match all" modes. You'll need the regex plugin to make it work, and the dll plugin to make up some structs for some of the send_message calls in the script. You can get the regex plugin (from files of the form regex*.zip) at http://tech.groups.yahoo.com/group/power-pro/files/Plug-ins_and_add-ons/.

If you want to save or load format settings to an ini file you'll need the ini plugin (found at http://tech.groups.yahoo.com/group/power-pro/files/Plug-ins_and_add-ons/0_Registry and Ini Access/) If you don't want to use the ini plugin, you can set a variable (st_sIniFile) in the configuration section of the script to null to prevent ini file use.

Also illustrates use of browse_for_file service.

For some reason text formatting doesn't always work for me in a rich edit control; for which case I've added a switch (st_bHighlightingWorks) that alters behaviour of **regexDialog.powerpro** to just select matches, not highlight them.

The **regexDialog.powerpro** script must be kept in the same folder as regexDialog.ico and regexDialog.txt..

regexDialogScintilla.powerpro illustrates a scintilla control in same manner as regexDialog.powerpro illustrates the use of rich edit controls; but you get a more reliable demo of character formatting and you get visible whitespace if you want it. It must be kept in the same folder as regexDialog.ico and regexDialogScintilla.txt. It also requires the regex plugin, and, If you want to save or load format settings to an ini file, the ini plugin (found at http://tech.groups.yahoo.com/group/power-pro/files/Plug-ins_and_add-ons/0_Registry and Ini Access/) If you don't want to use the ini plugin, you can set a variable (st_sIniFile) in the configuration section of the script to null to prevent ini file use.

Also illustrates use of browse_for_file service.

If you want to try out the version in this distro, follow the instructions in the readme.

Differences between this version and Sheri's:

- I never got the positioning code to work right even in Sheri's original, so I've just commented it out: this version just opens in centre screen foreground.
- You can drag the dialog by any part of it's background you can read (so there's no separate "drag" button).
- The dialog isn't reloaded every time you enter a search word, just the combo box
- If you type a search word, the combo box will be reloaded automatically when you stop typing for a two seconds (you can change that delay by altering the static variable st_nTypingDelay). Setting st_nTypingDelay to zero will suppress that behaviour….
- …and there are a bunch of other parameters you can change that configure the script, all right at beginning of the start() function.
- There are shortcut keys for the search (S) and paste (P) buttons, and escape has the same effect as hitting the cancel button.
- Tooltip on Paste button changes depending on whether help file or other file is loaded.

Options are changed in a subsidiary dialog, instead of using hidden controls in the main dialog. You can choose to change or edit the source file from that same subsidiary dialog, instead of from a button on the main dialog.

dialogPluginDemoNonNative.powerpro tests use of dialog services to access and modify controls in dialogs other than those made by the dialog plugin.

You can run dialogPluginDemo*.powerpro scripts multiple times, creating multiple dialogs. Due to over-simplicity in script that runs when an event fires, you'll only get blinking or progress bar movement in one of the multiple dialogs.

None of the scripts use the evaluate-expression operator "&" so is not dependent on your choice for it. It uses the ?c…c syntax to avoid problems with your declared escape character, so you should have no problems whether that's ' or \.

If you need to debug dialog scripts, you can run exec.scriptdebug(1) if the dialogs are not foreground; that is the dialog.run service should not have a <*show_type*> parameter of "foreground".

7.2 Other Scripts

In http://groups.yahoo.com/group/power-pro/files/Scripts/ :

Detlef Leidinger's FARMSIMF is a nice use of dialog and regex plugins.

sgp contributed MyConfigDialog, which used the dialog and vars plugin.

Sheri Pierce contributed CodeAccelerator, which uses dialog and reg plugins.

I've put in periodic_*.zip, an ini-driven set of scripts to run events every certain number of days, but only if user to wants them to go (if not they're re-presented the next day).

8.0 List of Services And General Notes on Usage

Ensure dialog.dll is in your PowerPro installation directory, or in the plugins subfolder thereof.

There are numerous services in this plugin, listed below. All services that take a dialog handle as a first argument

dialog.service(<*dialog_handle*>,….)

also allow the syntax

***<dialog_handle>*.service(….)**

In addition *aliases*, listed below in italics, only work with the latter syntax:

***<dialog_handle>*.alias(….)**

The services are:

service	description	section
define	define a dialog's properties	10.1
define_control, *control*	add a control definition to a dialog	10.2
make_ctrl_handle, *ctrl_handle*	return a handle to a specific control to be used in <*handle_to_control*>.service syntax	10.4
create	creates a dialog as an invisible window, not yet executing	10.5
run	run a dialog	10.6
show	show (or hide) a dialog or control	10.7
enable	enable or disable a control	10.8
set_focus, *focus*	set focus to a control	10.9
get_value, *get*	get text or other property of a control or dialog	10.10
set_value, *set, modify, add*	set a control’s text or other value, or the dialog’s caption	10.11
clear, *remove*	remove items from a control (list items, tabs, list view columns, etc	10.12
set_tooltip, *tooltip*	set a control's tooltip	10.13
set_range, *range*	set range, page size for scrollbar, spinner, progress, or slider controls	10.14
set_colour, *colour*	set control's foreground, background colours	10.15
rgb	get RBG COLORREF from red, green, blue values	10.16
set_image, *image, set_images, images*	set a static control's image	10.17
set_response, *response*	change command, args, events for control or dialog	10.18
change_style, *style*	change styles for a control or the dialog	10.19
set_icon, *set_icons*, *icon*, *icons, add_icon, add_icons*	change the dialog icon	10.20
set_font, *font*	change dialog's or control's font	10.21
set_position, *position*	change position or dimewnsions of dialog or control	10.22
get_last_clicked	used in scripts responding to right mouse click	10.23
send_message, *message, send*	send a message to a control or dialog	10.24
browse_for_file	bring up file open/file save common dialogs	10.25
choose_font	bring up font common dialog	10.26
destroy_window	kill off a dialog's window	10.27
destroy	kill off a dialog and it's data structure	10.28
destroyall	terminate all dialogs and their data structures	10.29
export	export a created dialog to a text file: NOT YET IMPLEMENTED	10.30
version	returns plugin version number as four digit number, last two to be taken as right of decimal.	10.31
help	opens help file for plugin, if it can be found	10.32
returns_values returns_status returns_nothing	determine what if anything all other services return via the calling form returnval = dialog.service	10.33
error_dialog_on error_dialog_off	turn PowerPro script error dialog on and off	10.34

service	description	section
set_base	sets base when	10.35
config	set location of configuration ini file	10.36
unload	remove plugin from memory	

These are described below, in Section 10 and its subsections.

There's also a service, **editor_support**, which is only there to allow interaction with the dialog editor. Not generally for public consumption.

9.0 Writing scripts using the dialog plugin

Call the appropriate plugin service as follows:

retval = **dialog.<service>(**arguments if any**)**

Results, if any, are generally available as the returned value from the above expression, though there are alternatives**.**

After you are finished using the **dialog plugin** in your script, you can if you wish unload it with

dialog.unload

It's probably best done if you do not foresee the plugin being used again for a while.

Services that return data, by default, comes back as the result of the service expression (returnval in the above line), and, by default again, set a result string in variable **dialog_status**.

If you wish different behaviour, call either returns_status() or returns_nothing().

If you call returns_status(), all subsequent service calls will return a result string as the result of the call. **dialog_status** won't be altered, and data will be returned via a variable (**dialog_result**).

Use returns_values() to return to default behaviour.

If you unload the plugin, it's behaviour returns to the default in any subsequent call, i.e. further **dialog** service calls will return values.

There are other services which affect the behaviour of the **dialog plugin** (see Sections
10.33 - 10.36). You can also customise the behaviour of plugin services by providing a configuration ini file (see Section 9.1).

If you do an *Exec ChangeConfiguration, and the new and old pcf files are in different folders, you should unload this plugin before using any of its services with the new configuration. (only necessary if you use relative paths to specify the location of ini files).

9.1 The Configuration ini File

On the first call to any **dialog** service, the plugin checks for a file called dialog.ini in the folder pointed to by the Powerpro variable pprofolder (usually the folder in which the currently active pcf file is found). It then looks for a [dialogConfig] section in that ini file. If that's not found, it looks for the same section in the file plugins.ini in the same folder. If either are found, it looks for the following values in the section:

key	possible values	default value	can modify using service	meaning
helpFileLocation	path	null	none	set to location of chm or rtf help file
defaultFieldSeparator	single char	|	none	separator used to delimit fields within lines in a dialog definition file.
defaultEvaluationMarker	single char	#	none	used to mark a field for evaluation in a dialog definition file
			set_base	set base of
raiseErrors	0 1 y n t f *	1	error_dialog_on error_dialog_off	determines if errors in syntax format, service arguments etc cause powerpro to raise the script error dialog.
whatToReturn	s d n †	d	returns_values returns_status returns_nothing	for services that return results, determines whether what is returned by a service is the value, the status (whether there was an error or not), or nothing.

* Only the first non-whitespace character of the key's value is checked, no matter how many there are. So values "yes", "no", "true" or "false" will work. As usual the ini value is case-insensitive, so "Y" and "FALSE" are also valid.

† Only the first non-whitespace character of the key's value is checked, so "status", "data" and "none" will work.

If no dialog.ini or plugin.ini files are found with a [dialogConfig] section, the dialog plugin will initially just use the compiled-in, default values (specified in the third column above) to configure itself.

If an ini file with a [dialogConfig] section is processed, and there's a possible value that could be in ini but isn't, the **dialog plugin** reverts to the default value for that value.

Once an ini file with a [dialogConfig] section is processed, its values stay in force until the **dialog plugin** unloaded or the config service is run.

If the first service called is one which itself changes the configuration (e.g. error_dialog_on), the configuration ini file will be found and evaluated *before* the service is applied.

If you want to change your initial configuration, you can use the ini plugin to do it. E.g.:

ini.set(pprofolder ++ "plugins.ini", "dialogConfig", "raiseErrors", "1")

After making your changes, either:

- unload the **dialog plugin**. The next time you use one of its services, the new configuration will kick in. Or
- run dialog.config**(<path to configuration ini file>)**.

If you do an *Exec ChangeConfiguration; and if the new and old pcf files are in different folders; and if there's a dialog.ini or plugins.ini file in the new pprofolder, the configuration it specifies won’t take effect until you take one of the steps above..

9.2 How to Create a Dialog

9.2.1 Don't Bother: Use PowerPro Functionality

Many dialogs are available as simple PowerPro functions, which you can just call from a script. See the "Input dialogs" section of PowerPro help. In particular the the messagebox and the various input*** functions may be all you need.

Dialogs are available

You trigger from e.g. a button click (see for instance the ".." button near the "Pick a file" edit box in the dialog created by the script dialogPluginDemo1.powerpro).

9.2.2 Making the Dialog

There are two approaches. You can write a script, in which you use dialog.define to declare the dialog's properties, and successive calls to dialog.define_control to declare properties of controls. That looks like:

```
local hDlg = dialog.define(0, 0, 100, 100, "Hi there", "minbox",….)
dialog.define_control(hDlg, 10, 20, 20, 12, "button", "bt1", …)
dialog.define_control(hDlg, 50, 20, 20, 12, "button",  "bt1",…)
….
dialog.run(hDlg)
```

Or, more elegantly, you can use the **object.service** syntax, with or without service aliases:

```
local hDlg = dialog.define(0, 0, 100, 100, "Hi there", "minbox",….)
hDlg.define_control(10, 20, 20, 12, "button", "bt1", …)
;control is an alias for define_control
hDlg.control (hDlg, 50, 20, 20, 12, "button",  "bt1",…)
….
hDlg.run
```

Or you can create a *<dialog_definition_file>*, which defines all the properties of the dialog and its controls, which you then invoke from a script using the create or run services. That script looks like:

```
dialog.run(<dialog_definition_file>)
```

and the dialog definition file like:

```
 0, 0, 100, 100, Hi there, minbox,…
10, 20, 20, 12, button, bt1,.....
50, 20, 20, 12, button,  bt1,....
```

The good news: your script is a lot less complicated. Bad news: your *<dialog_definition_file>* is pretty complicated. Personally, I find it easier to keep track of what I'm doing by building a script with dialog.define and dialog.define_control, but YMMV. Also, although I try to issue error messages tagged with the offending line number when I detect an error in a dialog definition file, I thing error reporting is probably better in a script where you build the dialog control-by-control.

Speaking of error handling: I recommend that you initially run your dialog-creating scripts with the default error handling behaviour, in which PowerPro throws up an error dialog if there's a syntax error in your script or dialog definition file. In other words, don't invoke dialog.error_dialog_off() before creating your dialog. It will be easier to detect problems, and given the number of parameters and fields you have to play with, believe me, you'll have problems.

9.2.3 Modeless Versus Modal Dialogs

Once you've defined your dialog and it's controls. You run it. You can do that as either a modeless or modal dialog. Difference is, a when you execute dialog.run and specify modeless, your script keeps right on running; whereas if you specify the *<modal_mode>* parameter as anything greater than 0, it's modal, and you don't return from your dialog.run statement under the user dismisses it.

Which means, since there are quite a few ways you might want to customise your dialog before the user sees it, you might be particularly interested in the create service, which allows the dialog to come into existence as a window, but not actually start executing as a dialog.

You can launch a modal dialog from another dialog, the launching dialog will be disabled until the user dismisses the child dialog.

If you launch a modal dialog from another modal dialog, it is *essential* to pass the correct *<dialog_handle>* as the owner for the second dialog. Bad Things will happen otherwise.

You can create multiple simultaneous instances of the same *modeless* dialog by running the same script more than once.

I've not worked out how to use an event in conjunction with multiple modeless dialogs (to e.g. get a blinking control).

9.2.4 Configuring Controls

Many services that change control properties require that the control exist before you can use them (the constraint is documented for each service). So you may want to create a dialog, either from a *<dialog_definition_file>* or a previously returned handle to a dialog by dialog.define

```
local hDlg = dialog.define(0, 0, 100, 100, "Hi there", "minbox",….)
hDlg.define_control(" 75   0   20   20", "static",    "stPic1", "",  "grayframe")
hDlg.define_control("  0 p10  90  20", "slider",   "trk1",    "",   "3d top
autoticks", ;;
              cb("@onSlide"), "", "",        "green", "silver")
hDlg.create(0)

hDlg.set_image("stPic1", "exclamation")

hDlg.send_message("trk1", "cleartics", 1, 0)
hDlg.set_range("trk1", 1, 100)
hDlg.set_value("trk1", 30)….
….
hDlg.run("foreground")
```

9.2.5 Getting the Dialog to Do Stuff

Everything interesting in a dialog has to do with how it reacts to user actions. For both the controls and the dialog as a whole, you can attach (via a service parameter or dialog definition file field) the name of a script to run when a user action occurs; an argument to pass to that script; and you can alter the events that cause the script to run. Further, you can alter that behaviour at any time for the dialog or any control by invoking dialog.set_response (although if you want alternative behaviours when e.g. a user clicks a button, it's probably easier to do conditional code in the scripts that run.).

My preferred style is to start a .PowerPro file with the code to create a dialog and start it running, finishing that code section with a quit statement; then to follow that block of code with various functions that would deal with various user actions. Each *<script_to_call>* would then take the form:

```
cb("@onAction ") ;; cb() is the powerpro callback function
```

which would invoke

```
Function onAction(sUserArg, dlgHan, iCtrlNo, msg1, msg2)
local dialog_status
….
quit
```

Once a dialog is created, there are other services you can use to change a dialog's or control's properties on the fly, in response to user actions. You can change control colours, styles and text; change which control has focus; change a scrollbar's properties and position; change a dialog's caption , icon or styles; set an image in a static; show or hide a control; show, hide or otherwise change the show status of the dialog; or enable or disable a control. There's a list of all those services, and more, here.

In most of those services, you need to identify which control you want to operate on (For instance, on a button press you might want to populate a list box, or show a previously-hidden check box). You can refer to controls either by number or name. The latter is one you provide as an parameter to dialog.define_control, the former is handed out as the return value from each dialog.define_control. I find using names somewhat more flexible; in particular if you use numbers in reacting scripts to a dialog created from a dialog_definition_file, you'll have to figure out what number to use by counting control-description lines in the definition file, and adding a magic number to get the control id. Not very safe if you change your dialog around.

If you need operate many times on the same control, it can be more convenient to generate a *<handle_to_control>*, which can be used with the **object.service** syntax.

```
local tabCtrl =  hDlg.make_ctrl_handle("tabctrl1") ;; tabctrl1 in control id
tabCtrl.set_images("exclamation", ?"plugins\dialog.dll", 3)
tabCtrl.set(1, "choices 1",   1, "eb1", "ax3swf", 1)  ;; set is alias for set_value
tabCtrl.set(2, "choices 2",   2, "stFrame", "stHereis",    "trk1",  "lb1")
```

Services that don't return a result useful to users (e.g. the get_* services) usually (there are exceptions) return a handle to a control *and* which operate on a control can be chained onto:

services onto which other service calls can be chained		**services onto which other service calls can *not* be chained**	
change_style	set_image	browse_for_file	get_value
clear	set_position	choose_font	make_ctrl_handle
enable	set_range	create	rgb
set_focus	set_response	define	run
set_colour	set_tooltip	define_control	send_message
set_font	set_value (usually)	destroy	set_icon
	show	destroy_all	
		destroy_window	

You can chain such services using the dot syntax: e.g. if iBoxNo is a list or combo box id. E.g.

```
local hCb = dlgH.ctrl_handle(iBoxNo)
hCb.set("hello", "add").set("hello kitty", "add").set("hello fido", "add")
hCb.set("hello mouse", "add").set("hello erk", "add").set("hello tooth", "add")
```

(pinched from the populateBox function in **dialogPluginDemo1.powerpro**). Same syntax is also used in **regexDialog.powerpro)**.

page 24 of 260

9.2.6 Finishing The Dialog

Dialogs will have one or more buttons (with *<id>*s "ok" and/or "cancel") which cause a dialog to stop processing messages. It is your responsibility in the *<script_to_call>*s associated with the those controls to destroy the dialog.

There are three relevant services: dialog.show, dialog.destroy_window and dialog.destroy.

dialog.show with the "hide" <*show_type*> parameter causes a window to disappear from the screen, but you can still extract data from its controls.

destroy_window causes the dialog window to be destroyed, but the plugin data structure and its associated *<dialog_handle>* still exists. It's safe to call destroy_window in any *<script_to_call>* triggered by any control, but it's irreversible: once the window's gone, it's gone.

dialog.destroy on kills off the plugin data structure and its associated *<dialog_handle>* associated with a dialog, (as well as destroying the dialog window) when it's timeto completely eliminate the dialog. I find it safe to call in a *<script_to_call>* associated with e.g. a button for a modeless dialog, but, for a modal dialog, only after the call to dialog.run that started that modal dialog has returned.

If you want the <escape> key to close a dialog, you should have a control with the *<id>* of "cancel" or "escapable" (normally a button). The former assumes you will be destroying the dialog in your *<script_to_call>*, the latter assumes you will not (and would therefore typically be hiding it instead).

If you want the alt <F4> key, or clicking on the X-icon in the dialog caption bar, or on the "close" option in the control menu to close a dialog, you should either have a control with the *<id>* of "cancel" and a defined <*script_to_call*>, or use set_response with a <*sys_command*> of "close". If you have both, the former applies.

The *<script_to_call>* associated with a button with an *<id>* of "ok" will fire when you hit the [enter] key.

There's also the *<action_on_close>* parameter: see Section 11.5.3.

page 25 of 260

9.2.7 Use PowerPro Functionality

You can trigger useful subdialogs (from e.g. a button click) created an managed by built-in PowerPro functions. See the "Input dialogs" section of PowerPro help. In particular the messagebox, pickstring, inputcolor, inputdate and inputdatetime functions might be useful.

See for instance the ".." button near the "Pick a file" edit box in the dialog created by the script dialogPluginDemo1.powerpro.

If you want to use dialogs thrown up by Powerpro using functions like pickstring and inputcolor, you might want to make them act like modal dialogs with respect to your plugin-generated dialog. Best way to do that is to disable the latter completely, run your powerpro function, then reenable:

```
;first parameter is target, the pseudo control id for dialog as a whole
myDlg.enable(0 ,0)
local sStr = pickstring(sLines, "Pick one now!")
myDlg.enable(0 ,1)
```

9.2.8 Dealing with Index Base in Distributed Dialogs

If you plan to distribute your dialog to other users, remember that they may have an **indexBase** key in their configuration ini file (dialog.ini or plugins.ini) that may not be consistent with the base you've used in your script (only relevant if your dialog uses (the clear or get_value (alias *get*) or set_value (alias *set*)services on listviews, combo boxs, list boxs or tab controls.)

To get around this problem, it might be an idea to

- call set_base at the beginning of your script;
- save the **indexBase** key value (see the get_base() function in most sample scripts);
- …and restore that saved key value when your dialog exits.

The second step requires the ini plugin.

9.2.9 The Skeleton and Utility Dialogs

skeletonDialog.powerpro will run a simple dialog with "OK" and "Cancel" buttons. A simple edit of the script allows you top create the same dialog from the dialog_definition_files skeletonDialog.txt and skeletonDialog.ini. Might be a place to start from.

dialogViewer.powerpro is a simple dialog that can be used to display text in an edit box. It will run on it's own, or you can run it modal over notepad using **dialogViewerTest.powerpro**

9.2.10 Multiple Simultaneous Dialogs From The Same Script

Generally no problem, unless you use static variables for anything (e.g. to hold a handle to an event, or flags to control behaviour over time). Handles to events and suchlike must instead by held in (normally hidden) controls, which will be unique for each instance of a control. See for instance dialogPluginDemo2.powerpro.

9.2.11 Enforcing a Single Instance of a Dialog from a Script

Best bet is probably to:

- store the handle to the dialog returned by dialog.define in a static. Best not to set that static until just before you're ready to run the dialog; otherwise your script may terminate on some error, and you'll be left with a static pointing to a unworkable dialog. See sample script dialogPluginDemo1.powerpro.
- set the static to something else (null string?) when the dialog is destroyed
- test for a non-null static variable at the beginning of your script, and if found show the dialog in question

If you want your single instance of your dialog to respond to the escape key (e.g. by going invisible), you'll need to provide a control (button?) with an <*id*> of “escapable”.

page 27 of 260

9.3 How to Use the Dialog Plugin to Manipulate Non-Plugin Dialogs and Their Controls

9.3.1 Don't Bother: Use the win plugin

First, get your handle. You need a handle to the dialog or the specific control you want to manipulate. Use **win.handle**(cl) if you want to get the handle of a dialog. You may need a combination of **win.handle**(cl,"text"), **win.childhandlelist** and **win.handlelist** for controls. See **controlFontChanger.powerpro** for an example.

To set the text of a control or dialog: win.settext(cl, "newtext") sends a WM_SETTEXT which may set the text of the window matching cl to newtext. **To hide a control or dialog:** use **win.hide**(cl), cl as above.

To show a hidden control or dialog: use **win.showna**(han) for a control, **win.showna**(han) or **win.show**(han) for a dialog..

To enable/disable a control or dialog: use **win.enable**(han).

And so on. You should be able to use other win plugin services to get and set positions, set tooltips, cause a control or dialog to flash, set transparency and soon.

9.3.2 Fonts

dialog.set_font works with a first argument of a <*window_handle*>, which you can obtain as above (handle(cl) for main windows (dialogs) or handle(cl,"text") for controls).

dialog.get_value(<*window_handle*>*,* "font") gets the font of any window in a format compatible with set_font.

9.3.3 Using Other Plugin Services

Some services that take a dialog handle and a control id as first arguments can take instead a window handle to a control, and that control can live in some dialog or window not created by the dialog plugin. The plugin analyses the window handle you pass in, and determines from the window's class and styles what type of control it is, as follows:

class…	…with style…	= control type	class…	= control type
Button	none of the following:	**button** (II.1)	msctls_trackbar32	**slider** (II.10)
Button	BS_GROUPBOX	**group** (II.2)	msctls_progress32	**progress** (II.11)
Button	BS_AUTOCHECKBOX	**checkbox** (II.3)	SysDateTimePick32	**datetime** (II.12)
Button	BS_AUTO3STATE	**3state** (II.3)	SysMonthCal32	**monthcal** (II.13)
Button	BS_AUTORADIOBUTTON	**radiobutton** (II.3)	AtlAxWin	**activeX** (II.14)
Static	N/A	**static** (II.4)	msctls_statusbar32	**statusbar** (II.15)
Edit	N/A	**editbox** (II.5)	SysTabControl32	**tab control** (II.16)
ListBox	N/A	**listbox** (II.6)	SysTreeView32	**treeview** (II.17)
ComboBox	N/A	**combobox** (II.7)	RichEdit20A	**richedit** (II.18)
ScrollBar	N/A	**scrollbar** (II.8)	Scintilla	**scintilla** (II.19)
msctls_updown32	N/A	**spinner** (II.9)	SysListView32	**listview** (II.20)

It then allows the appropriate arguments for that service on that type of control. (If you pass a handle that isn't to a control, or is to a control of a class not recognised by the plugin, you'll get an error message. For instance many controls that look like Listviews (real class: SysListView32) are in fact of classes like ATL:BrowserListView or some Borland class. You may with to use a tool like winSpector, Spy++ or AU3_Spy that comes with autoIt to determine a control's class).

I've added the test script dialogPluginDemoNonNative.powerpro to demonstrate usage of some of the services below:

get_value (alias *get*), enable and show: should cause no problem for all standard (classic controls), and for listViews and statusbars. So you can grab all or part of a listview's contents, hide or disable a button, probably get the selected node in a listview (though haven't tried that last). I can add support for other common controls if required.

set_position: dunno. In theory you can change a control's position and size, but will that annoy the dialog's owning process? User beware.

send_message: Might as well use win.sendmessage, unless you want to use any of the named messages defined for each type of control set_value: User beware. Changing the label on a button or butting text into an edit box seems to work without horrible consequences (but of course you could do the same win win.settext. Changing the contents of a listview or treeview strikes me as very dicey: suddenly the user sees one set of data but the owning application may think the underlying data is something different. On the other hand it's probably perfectly safe to change selection or focus in a remote listview control.

set_font: I've tried it for edit boxes owned by Powerpro, might work for controls belonging to other processes.

change_style: I tried it on notepad's editbox and it got angry. Notepad??? Angry?? Probably a bad idea.

set_colour: may work, haven't tried yet.

choose_font: all the argument does is determine the font used to inisialise the font choice common dialog, so not sure why you'd use it, but you can.

10.0 The Services

Details of specific services follow.

Some portions of some service arguments that recur in various services are further described in subsections of Sections 11.0.

A few of the arguments to dialog services are pretty complicated. I've used the following pseudo-BNF to document them.

<*name*> : a named argument or part thereof

<*name*> := … :what <*name*> is made up of

[] : an optional part

<*this*> | <*that*> : alternatives

10.1 define

dialog.define(<*X*>, <*Y*>, <*width*>, <*height*>*,* <*caption*> [, <*styles*>
[, <*script_to_call*>[, <*command_arg*> [*,* <*events*>
[, <*action_on_close*> [*,* <*icon_path*> [*,* <*icon_number*>
[, <*font_spec*> [, <*right_click_command*>]]]]]]]]**)** or

dialog.define(<*dimensions*>*,* <*caption*> [, <*styles*>
[, <*script_to_call*>[, <*command_arg*> [*,* <*events*>
[, <*action_on_close*> [*,* <*icon_path*> [*,* <*icon_number*>
[, <*font_spec*> [, <*right_click_command*>]]]]]]]]**)** or

Used in all sample scripts except those that create dialogs from dialog_definition_files, i.e. all but dialogPluginDemoFromConfigFile.powerpro, regexDialog.powerpro and regexDialogScintilla.powerpro

dialog.define defines a dialog's properties and returns a handle to that dialog which can be used in other services.

If successful it returns a <*dialog_handle*>. See Section 11.1 "Handles to Dialogs and the <*dialog_handle*> parameter". However, other service calls *cannot* be chained to it (because services that return chainable handles require that that handle be automatically deleted, and you probably don't want handled returned by define to disappear instantly…).

The arguments:

<*X*> Required (unless subsumed into <*dimensions*>): Horizontal position of upper left hand corner of dialog, in dialog units. If you specify the "centre" or "centremouse" styles, this parameter will be ignored.
If they're not ignored, relative to the upper left hand corner of screen, unless you're creating or running the dialog as modal with a parent, in which case it'll be relative to the upper left hand corner of that parent's window. May have a sizing modifier.

<*Y*> Required (unless subsumed into <*dimensions*>): Vertical position of upper left hand corner of dialog, in dialog units.
If you specify the "centre" or "centremouse" styles, this parameter will be ignored.
If they're not ignored, relative to the upper left hand corner of screen, unless you're creating or running the dialog as modal with a parent, in which case it'll be relative to the upper left hand corner of that parent's window. May have a sizing modifier.

<*width*> Required (unless subsumed into <*dimensions*>): Width of dialog, in dialog units. May have a sizing modifier.

If prefixed by "+", dialog width will be maximum sum of <*X*> and <*width*> for any control, plus the quantity after the "+"

page 32 of 260

<height> Required (unless subsumed into *<dimensions>*): Height of dialog, in dialog units. May have a sizing modifier.

If prefixed by "+", dialog height will be maximum sum of *<Y>* and *<height>* for any control, plus the quantity after the "+".

<caption> Required, but can be the null string: the dialog caption that will appear on its title bar. No length limit.

<*styles*> Optional: Keywords or letters (see this table), or numerical values modifying the dialog's styles. For further details in Section 11.4 "Styles: the <*styles*> parameter".

Dialog Styles			
this style name	**or this letter**	**#define'd symbol**	**means**
sysmenu	s	WS_SYSMENU	add sysmenu box to dialog caption bar
minbox	n	WS_MAXIMIZEBOX	add maximize box to dialog caption bar
maxbox	x	WS_MINIMIZEBOX	add minimize box to dialog caption bar
centre	c	DS_CENTER	centre dialog on screen
centremous e	m	DS_CENTERMOUSE	centre dialog on mouse location
noborder		WS_BORDER	prevents border being applied
nodlgframe		~WS_DLGFRAME	prevents WS_DLGFRAME style being applied, which means no double border
thickframe	t	WS_THICKFRAME	creates a dialog with a thick frame that can be used to size it
3d	3	extended style WS_EX_CLIENTEDGE	makes 3d effect
modalframe	-	extended style WS_EX_DLGMODALFRAME	
hidden	h	~WS_VISIBLE	prevents WS_VISIBLE being applied, hiding dialog
tool toolwin	-	extended styles WS_EX_WINDOWEDGE | WS_EX_TOOLWINDOW	creates tool window; title bar shorter than normal, title drawn in smaller font;. doesn't appear in task bar ALT+TAB
topmost	-	extended style WS_EX_TOPMOST	dialog placed above all nontopmost windows and stay s above them even when the dialog is deactivated
draggable	-	-	can be dragged by mouse in client area of dialog (as well as in title bar)

<*script_to_call*> Optional. A PowerPro command to execute if an interesting event is fired by a control and that control doesn't handle it itself. For further details in Section 11.5 "Script Calls and Their Arguments".

<*command_arg*> Optional. An argument passed to the above command if it's invoked. If present <*script_to_call*> must not be the null string. For further details in Section 11.5 "Script Calls and Their Arguments".

<*events*> Optional. A list of notifications accompanying a WM COMMAND message that may cause <*script_to_call*> to be executed. See Section 11.6 "Specifying Which Messages Are Responded To".

<*action_on_close*> Optional: a PowerPro command that will run if you close a dialog by any means but clicking a button. Further details here.

<*icon_path*> and <*icon_number*> Optional: Specify an icon for the dialog. For details see Section 11.8 "Icons: the <*icon_path*>*,* <*path_to_image*> and <*icon_number*> parameters".

<*font_spec*> Optional: The dialog's font. If specified, all controls within the dialog use this font (otherwise they use the system font). (Windows always uses the system font for the title of the dialog box; you can't override that). See Section 11.9 "Fonts: the <*font*> and <*font_spec*> parameters".

<*right_click_command*>: Optional: If present, normally a command that causes a PowerPro menu to appear, thus simulating a context menu, e.g. using menu.show or cl.ShowMenu. The script will be run if dialog user right clicks on any part of a dialog *not* containing a control. See Section 11.11:
"<right_click_command>: Context Menus"

10.2 define_control

dialog.define_control(<*dialog_handle*>, <*X*>, <*Y*>, <*width*>, <*height*>,
<*control_type*> [,<*name*> [,<*text_or_var*> [, <*styles*>
[, <*script_to_call*> [, <*command_arg*> [, <*events*>
[, <*foreground*>, [<*background*> [, <*id*>
[, <*font_spec*> [, <*tooltip*> [, <*tooltip_style*>
[,<*right_click_command*>]]]]]]]]]]]]**)** or

<*dialog_handle*>.define_control(<*X*>, <*Y*>, <*width*>, <*height*>,
<*control_type*> [,<*name*> [,<*text_or_var*> [, <*styles*>
[, <*script_to_call*> [, <*command_arg*> [, <*events*>
[, <*foreground*>, [<*background*> [, <*id*>
[, <*font_spec*> [, <*tooltip*> [, <*tooltip_style*>
[,<*right_click_command*>]]]]]]]]]]]]**)** or

dialog.define_control(<*dialog_handle*>, <*dimensions*>,
<*control_type*> [,<*name*> [,<*text_or_var*> [, <*styles*>
[, <*script_to_call*> [, <*command_arg*> [, <*events*>
[, <*foreground*>, [<*background*> [, <*id*>
[, <*font_spec*> [, <*tooltip*> [, <*tooltip_style*>
[,<*right_click_command*>]]]]]]]]]]]]**)** or

<*dialog_handle*>.define_control(<*dimensions*>,
<*control_type*> [,<*name*> [,<*text_or_var*> [, <*styles*>
[, <*script_to_call*> [, <*command_arg*> [, <*events*>
[, <*foreground*>, [<*background*> [, <*id*>
[, <*font_spec*> [, <*tooltip*> [, <*tooltip_style*>
[,<*right_click_command*>]]]]]]]]]]]]**)**

alias: control

Used in all sample scripts except those that create dialogs from dialog_definition_files, i.e. all but dialogPluginDemoFromConfigFile.powerpro, regexDialog.powerpro and regexDialogScintilla.powerpro

dialog.define_control() adds knowledge of a control and it's properties to an existing dialog. It returns a handle to a control, which can be used in other services to indicate which control you want to operate on. Other service calls *cannot* be chained to define_control (because services that return chainable handles require that that handle be automatically deleted, and you probably don't want handles returned by define_control to disappear instantly).

Normally the drill is to invoke define_control (a lot) before a dialog is created or run. However, you can call it after either created or run if you want to. Most likely reason to do that would be that you create a dialog from a <*dialog_definition_file*>, then add to it using **dialog.define_control.**

Once a control has been defined, you can't undefined or delete it from the dialog of which it is part. On the other hand you can hide controls at any time using the dialog.show service, so they can seem to "disappear" as far as the dialog user is concerned.

The arguments:

<*dialog_handle*> Required: A handle returned by dialog.define. See Section 11.1 "Handles to Dialogs and the <*dialog_handle*> parameter".

<*X*> Required (unless subsumed into <*dimensions*>): Horizontal position of upper left hand corner of control, relative to upper-left hand corner of dialog, in dialog units. May have a sizing modifier. May be relative to the previous control.

<*Y*> Required (unless subsumed into <*dimensions*>): Vertical position of upper left hand corner of control, relative to upper-left hand corner of dialog, in dialog units. May have a sizing modifier. May be relative to the previous control.

<*width*> Required (unless subsumed into <*dimensions*>): Width of control, in dialog units. May have a sizing modifier. May be relative to the previous control.

<*height*> Required (unless subsumed into <*dimensions*>): Height of control, in dialog units. May have a sizing modifier. May be relative to the previous control.

<control_type> Required: A letter or word specifying the control type:

page 40 of 260

control types					
this control name	**or this letter**	**details in Appendix**	**default WM_COMMAND notifications handled**	**other notifications that might be useful**	**styles: in addition to those applying to all controls**
button	b	II.1	BN_CLICKED	BN_DBLCLK	see this table
group	g	II.2	none		see this table
checkbox	c	II.3	BN_CLICKED		see this table
3state	3	II.3	BN_CLICKED		see this table
radiobutton	r	II.3	BN_CLICKED		see this table
static	t	II.4	none	STN_CLICKED "notify"	see this table
editbox	e	II.5	none	EN_CHANGE	see this table
listbox	l	II.6	none	LBN_SELCHANGE	see this table
combobox	o	II.7	none	CBN_SELCHANGE CBN_EDITCHANGE	see this table
scrollbar	s	II.8	none; WM_HSCROLL, WM_VSCROLL always handled		see this table
spinner	u	II.9	none; WM_HSCROLL, WM_VSCROLL always handled		see this table
slider	r	II.10	none; WM_HSCROLL, WM_VSCROLL always handled		see this table
progress	p	II.11	none		see this table
datetime	d	II.12	none		see this table
monthcal	m	II.13	none		see this table
activeX	x	II.14	none		none
statusbar	-	II.15	none		see this table
tabcontrol	-	II.16	none		see this table
treeview	-	II.17	TVN_SELCHANGED		see this table
richedit	-	II.18	none	EN_CHANGE	see this table
scintilla	-	II.19	none		no special
listview	-	II.20	none		see this table
animation	-	II.21	none		see this table

<*name*> Optional: The name by which the control can be referred to as its *control id* (see Section 11.3 "Control Ids"). If you're going to use it as *control id* in other service calls, better make sure it's non-null. Names must

- be unique (case sensitive comparison used) within a dialog:
- be of 2-63 characters in length
- must begin with an alphabetic character
- must not be the same (case insensitive) as any of the <*show_type*>s allowed for the show command; the <*foreground*> or <*mouse_state*> parameters of set_colour; the possible styles that can apply to a dialog with change_style; and the properties that can be applied to dialogs in a call to get_value (alias *get*). Here's an alphabetical list of all those reserved words:

aqua black blue centre centremou se default focus	focused font foreground fuchsia green grey hover	hovering hwnd lime maroon maxbox maximize minbox	minimize navy noborder nodlgframe normal olive owner	pressed prevctrl purple red restore silver sysmenu	teal thickframe traydel trayicon traymin white yellow

Given all these restrictions, you could name controls, with some sort of prefix containing no vowels, indicating the type of control (bt for button, cb for combo, etc); followed by a few more meaningful characters, again ideally omitting vowels from common words to minimise the possibility of collision with the reserved words listed above.

You'll get an error message if you break the rules.

If omitted, a control's <*name*> is the null string and can't be use in subsequent scripting to refer to the control. You still have the control's handle, returned by define_control and many other services.

<*text_or_var*> Optional: If present, the plugin checks to see if the string provided is the name of an *already-declared* global or static variable. If it is, the value of that variable is taken as the text of interest. The static must be declared in the script that creates the control in question.

The text, whether literal or in a variable, becomes the displayed text for buttons, check box controls, radio buttons, groups and edit controls; an ampersand included in the text will appear underlined and act as a shortcut key for the control. For list controls and combo box controls, it's used to populate the lists; separate items with newlines ("\n").

page 42 of 260

There's no length limit on <*text_or_var*>.

If <*text_or_var*> is the name of a global or static variable, and there's a control (normally a button) created with an *<id>* parameter of "ok" (i.e., a default button), when that default button is pressed (or the user hits the return key, same thing), any global or static variables associated with any control via the <*text_or_var*> parameter will be updated with the control's current value.

<*styles*> Optional: Keywords or letters, or numerical values modifying the control's styles. Some keywords apply to all controls: see this table. Other keywords work only for a specify type of control: see the appropriate table indicated above. For further details see Section 11.4 "Styles: the <*styles*> parameter".

<*script_to_call*> Optional: A PowerPro command to execute if an interesting event is fired by the control. See Section 11.5 "Script Calls and Their Arguments".

<*command_arg*> Optional: An argument passed to the above command if it's invoked. If present <*script_to_call*> must not be the null string. See Section 11.5 "Script Calls and Their Arguments".

<*events*> Optional: A list of notifications accompanying a WM COMMAND or WM_NOTIFY message that may cause <*script_to_call*> to be executed. See Section 11.6 "Specifying Which Messages Are Responded To".

<*foreground*> Optional: The control text colour. See Section 11.7 "Colours". If "default" or the empty string, becomes default text colour for the type of control. Has no effect on tab control, animation or spinner, scrollbar or activeX controls. For date picker, month-calendar, treeview and listview controls, this parameter can be interpreted as the part of the control you wish to colour.

<*background*> Optional: The control background colour. See Section 11.7 "Colours". If "default" or the empty string, becomes default background colour for the type of control. Has no effect on tab control, animation, spinner or activeX controls. For date picker, month-calendar, treeview and listview controls, this parameter can become the one colour you wish to set part of the control to.

<id> Optional: Can be "cancel", "ok" or "escapable", case-insensitive.

- "ok": control is the default (usually a button); the control's <script_to_call> is executed if the user exits by hitting the <enter> key. It's assumed the <script_to_call> will destroy the dialog with something like

  dlgHan.destroy()

  If you're going to associate a global or static variable with a control via the <text_or_var> parameter, you must have a control with an <id> of "ok", i.e. a default button.

  There must be at most one control (normally a button) with an <id> of "ok".

- "cancel": the control's <script_to_call> is executed if the dialog is exited by user hitting the <escape> key; or, if you haven't specified a <script_to_call> to perform by a call to set_response for the "close" <sys_command>, if the user hits the alt-F4 key, or selects "close" on the system menu, or the "X" at the far right of the title bar.

  It's assumed the <script_to_call> will destroy the dialog with something like

  dlgHan.destroy()

  There must be at most one control (normally a button) with an <id> of "cancel". If there's one of them there can't be one with…

- "escapable": the control's <*script_to_call*> is executed if the dialog is exited by any of the methods listed above.

  There must be at most one control (normally a button) with an <*id*> of "escapable" *or* with the <*id*> of "cancel"; one precludes the other. Normally it would be hidden.

  It's assumed the <*script_to_call*> will *not* destroy the dialog.

  If you have a button with <*id*> of "escapable", you'll probably also want to set_response for the "close" <*sys_command*> (otherwise stuff like <*alt-F4*> would not terminate the dialog.

<*font_spec*> Optional: The control's font. See Section 11.9 "Fonts: the <*font*> and <*font_spec*> parameters"

<*tooltip*> Optional: Text of tooltip for the control. If absent, no tooltip. Not valid for static controls. “\n” or “\r\n” can be used to force <*tooltip*> line breaks (or "/" with the "slashisnl" style).

<*tooltip_style*> Optional: Only has an effect if <*tooltip*> is present and non-null.

<*tooltip_style*> =: [<*width*>] [<*tooltip_styles*>]

<*width*>, if specified, implies a multi-line tooltip of <*width*> dialog units; <*tooltip*> will be word wrapped to fit within that <*width*>. You can alternatively include the width:nn pseudostyle anywhere in <*tooltip_styles*> (see end of table below).

<*tooltip_styles*> may contain any of the following names or letters, whitespace-delimited:

ToolTip Styles			
this style name	**or this letter**	**#define'd symbol**	**means**
always	a	TTS_ALWAYSTIP	With style, tooltip appears when the cursor is on a tool, regardless of whether the dialog is active or inactive. Without style, the toottip appears only when the dialog is active.
balloon	b	TTS_BALLOON	tooltip has the appearance of a cartoon "balloon," with rounded corners and a stem pointing to the item
centre	c	TTF_CENTERTIP	only valid for balloon style; changes where the arrow goes
slashisnl	-	-	interpret "/" as a newline
showafter:nn	-	-	nn (any number of digits) is length of time in msec that the cursor must remain stationary within the bounding rectangle of a tool before the ToolTip window is displayed. –1 to revert to default (system double click time)
stayopen:nn	-	-	nn (any number of digits) is length of time in msec before the ToolTip window is hidden if the cursor remains stationary in the tool's bounding rectangle after the ToolTip window has appeared. –1 to revert to default (forever)
width:nn	-	-	sets maximum width of tab. Use as alternative to prefixing <*tooltip_styles*> with <*width*>

<right_click_command>: Optional: If present, normally a command that causes a PowerPro menu to appear, using e.g. menu.show or cl.ShowMenu, thus simulating a context menu. The script will be run if dialog user right clicks on the control. See Section 11.11: "*<right_click_command>*: Context Menus"

10.3 define_set

dialog.define_set(<*dialog_handle*> [*, <name>* [*, <control_types>*
[*, <included_controls (and'ed)>* [*, <included_controls (or'ed)>* [*, <excluded_controls>*]]]]**)**

***<dialog_handle>*.define_set(**[*<name>* [, *<control_types>*
[*, <included_controls (and'ed)>* [*, <included_controls (or'ed)>* [*, <excluded_controls>*]]]]**)**

aliases: none

Used in sample scripts dialogPluginDemo1.powerpro, and, to no great purpose, regexDialog.powerpro.

Creates a set of controls, the handle or name of which can then itself be used as an arguments or object of some other services.

Returns a handle to the set, which can be used as if it was a handle to a control.

A set should be created once and once only in a dialog script. It can be accessed from multiple functions in the script by name or statically stored handle.

<name> Optional: a name which can be used as the *<ctrl_id>* in calls to other services.

At least one of *<control_types>*, *<included_controls>* and *<excluded_controls>* must be a non-null string.

<control_types> Optional: a white-space separated list of control type names. May be the null string. All controls of the specified types will be included in the set.

<included_controls (and'ed)> Optional: A white-space separated list of control names which must be included the set; a control is only included in the final set if it is selected both by this list of control names *and* by the list of *<control_types>.* Each name may include the wildcards "?", "*". If a name doesn't include wildcards and is not a valid control name, you'll get an error message.

You may not include the name of a set in *<included_controls (and'ed)>*; sets, in other words, can't include other sets (though the same control can belong to many sets).

If you include the same control twice (most probably because of overlapping wildcard expressions), it will of course be included in the set only once.

Either *<control_types>* is the null string, *<included_controls (and'ed)>* has the same effect as *<included_controls (or'ed)>*.

If *<control_types>, <included_controls (and'ed)>* and *<included_controls (or'ed)>* are all null strings, it's an error.

<included_controls (or'ed)> Optional: A white-space separated list of control names which must be included the set; a control is included in the final set regardless of what has already been selected by previous parameters*.* Each name may include the wildcards "?", "*". If a name doesn't include wildcards and is not a valid control name, you'll get an error message.

In particular you can specify *<included_controls (or'ed)>* of "*", meaning all controls; you would probably want to specify some *<excluded_controls>.*

<excluded_controls> Optional: A white-space separated list of control names which must be excluded from the in the set. Each name may include the wildcards "?", "*". If a name doesn't include wildcards and is not a valid control name, you'll get an error message.

Only the following services work with a set name or a handle to a set:

enable, show: The most likely suspects, so you can hide and show, or enable and disable, sets of controls in response to user actions.

set_colour: Also may be quite handy, but perhaps not for the more complex controls that need amusing special parameters, like the date pickers and month-calendars.

set_value: Most likely to be useful with no *<property>* parameter and something like the null string as an argument, for e.g. clearing a bunch of edit boxes or statics. Not recommended for the more elaborate controls that need funny parameters, like listviews or treeviews.

set_position: Only make sense if used with relative dimensions; other wise all controls in the set will end up in the same place

change_style: If you use named styles, they'll have to be once that are legal for all the types of controls included in the set.

set_font: Sets font for controls.

clear: Only applies to controls with multiple bits; probably only useful if you want to use it without arguments, which for some types of controls (e.g. combo box controls or list controls) means "empty it out".

send_message: Same story: a named message will have to be legal for all the types of control in the set. It may be that you have to use a numeric message code (possibly obtained from dialog-related defines.txt

10.4 make_ctrl_handle

dialog.make_ctrl_handle(*<dialog_handle> , <ctrl_id>***)**
<dialog_handle>*.make_ctrl_handle (<ctrl_id>***)**

alias: ctrl_handle

Used in sample scripts dialogPluginDemo1.powerpro, dialogPluginDemo4.powerpro, dialogPluginDemo5.powerpro, dialogPluginDemo6.powerpro, and all the regex scripts.

dialog.make_ctrl_handle returns a *<handle_to_control>* (a string beginning "I\x05", followed by a hex number), which can *only* be used to refer to a specific control in a dialog using the **object.service** syntax:

***<handle_to_control>*.service(**….**)**

Such a *<handle_to_control>* would only be useful in situations where you need to execute numerous services on the same control. This might occur for instance for tab controls, and treeview controls, both of which usually require numerous calls to set_value to configure.

Other service calls *cannot* be chained to **make_ctrl_handle**.

Unlike handles to dialogs, *<handle_to_control>*s stored in local variables are automatically deleted when the local goes out of scope. They are not automatically destroyed when the dialog which contains the control in question is destroyed. In fact, they don't need to be destroyed at all, as no special memory is associated with them, so *<handle_to_control>*s stored in static or global variables don't need to be deleted using the destroy service (although it does no harm to do so), and *<handle_to_control>*s passed as arguments don't really need to be localcopy'd (although it does no harm to do so).

page 51 of 260

10.5 create

> **dialog.create(**<*dialog_handle*> [,<*modal_mode*> [, <*owner*>]]**)**
> ***<dialog_handle>*.create(**[<*modal_mode*> [, <*owner*>]]**)**
> **dialog.create(**<*dialog_definition_file*> [,<*modal_mode*> [, <*owner*>]]**)**

Used in all sample scripts.

dialog.create causes either:

> first form: a dialog that's been defined and presumably populated with controls come into existence.
>
> second form: creates a dialog from a *<dialog_definition_file>*, returning a handle to the newly created dialog. This form returns the handle to the newly created dialog.

In neither case does the dialog show up on screen or start executing as a window.

dialog.create is useful if you want to fiddle with properties of controls before the dialog comes into existence; you may want to disable them, or hide them, for instance. It's particularly useful for modal dialogs, which would be impossible to initialise once running.

Other service calls *cannot* be chained to **create**.

The arguments:

<*dialog_handle*> either this or <*dialog_definition_file*> required: A handle returned by dialog.define. See Section 11.1 "Handles to Dialogs and the <*dialog_handle*> parameter".

<*dialog_definition_file*> either this or <*dialog_handle*> required: The path to a file that defines a dialog and its controls. See Section 11.2 "The <*dialog_definition_file*> Parameter and the Format of a Definition File".

<*modal_mode*> Optional:

- **0**: create a modeless dialog: the default option if <*modal_mode*> omitted
- **1**: create a modal dialog that stops the creating script from running until the dialog completes.
- **2**: create a modal dialog that stops all PowerPro activity until the dialog completes: bars won't responds, the debug window won't display, timers probably stop.

<*owner*> Optional: a *<dialog_handle>* for the a dialog created with this plugin; or a handle to any window. Only relevant for a modal dialog. If you provide an <*owner*> *<dialog_handle>*, the calling (owner) dialog or window will be disabled

until the dialog it creates is closed. If you are displaying a modal dialog from another modal dialog, it is *essential* to pass the correct *<dialog_handle>* as the owner for the second dialog. Bad Things will happen otherwise.

10.6 run

> **dialog.run(**<*dialog_handle*> [*, <show_type*> [, <*modal_mode*> [*, <owner*>]]]**)**
>
> ***<dialog_handle>*.run(**[<*show_type*> [, <*modal_mode*> [*, <owner*>]]]**)**
>
> **dialog.run(**<*dialog_definition_file*> [*, <show_type*> [, <*modal_mode*> [*, <owner*>]]]**)**

Used in all sample scripts.

dialog.run causes either:

first form: a dialog that's been defined and presumably populated with controls to actually run. It will also run a dialog that's already been created.

second form: creates a dialog from a *<dialog_definition_file>* and runs it, returning a handle to the newly created dialog. This form returns the handle to the running dialog if its non-modal.

Other service calls *cannot* be chained to **run**.

Both forms return the value supplied to dialog.destroy_window if running a modal dialog.

The arguments:

<*dialog_handle*> (either this or <*dialog_definition_file*> is required): A handle returned by dialog.define. See Section 11.1 "Handles to Dialogs and the <*dialog_handle*> parameter".

<*dialog_definition_file*> (either this or <*dialog_handle*> is required): The path to a file that defines a dialog and its controls. See Section 11.2 "The <*dialog_definition_file*> Parameter and the Format of a Definition File".

<*show_type*> Optional: if present and non-null, may be any of the values appropriate as arguments of the show service appropriate for a dialog.

<*modal_mode*> Optional: Same meaning as for the create service (0 or absent for modeless, 1 for modal, etc). If present, *and* you're running a previously created dialog; *and* this value's not the same as the <*modal_mode*> used in that invocation of the create service, you'll get an error message. I'd strongly suggest that if you're invoking dialog.run after a dialog.create, you omit this parameter. It's only there for situations where you're running dialog.run without previous invoking create.

<*owner*> Optional: a *<dialog_handle>* a *<dialog_handle>* for the a dialog created with this plugin; or a handle to any window. Only relevant for a modal

dialog, which will cause the owner dialog or window to be disabled until it's closed. If present, *and* you're running a previously created dialog; *and* this value's not the same as the <*owner*> used in that invocation of that create service, you'll get an error message. I'd strongly suggest that if you're invoking dialog.run after a dialog.create, you omit this parameter. It's only there for situations where you're running dialog.run without previous invoking create.

If you are displaying a modal dialog from another modal dialog, it is *essential* to pass the correct *<dialog_handle>* as the owner for the second dialog. Bad Things will happen otherwise.

10.7 show

dialog.show(*<target>* [*, <show_type>*]**)**
***<dialog_handle>*.show(**[*<target>*] [*, <show_type>*]**)**
<ctrl_handle>*.show([<show_type>*]**)**

dialog.show(*<dialog_handle>*, "traymin" | "trayicon", [, *<script_to_call>* [, *<command_arg>* [, *<mouse_action>*]]]**)**

***<dialog_handle>*.show(**"traymin" | "trayicon", [, *<script_to_call>* [, *<command_arg>* [, *<mouse_action>* [,*<icon_path>* [*, <icon_number>*]]]]]**)**

Used in all sample scripts.

dialog.show() changes it's running display's state; or shows or hides a control.

You must only invoke show *after* a dialog is created or run.

If applied to a control (not to the dialog as a whole), you can chain other service calls to **show** calls.

*<target>***:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". If using *<dialog_handle>, <ctrl_id>, <ctrl_id>* may be 0 or absent, indicating you wish to change the show status of the dialog itself. If *<ctrl_id>* is a control tab, all controls associated with the currently selected tab will be hidden if you're hiding the control, made visible otherwise.

If *<show_type>* is absent, it's taken to be SW_SHOW if *<target>* is a control, or SW_NORMAL if *<target>* is a dialog handle.

For dialog and controls, *<show_type>* may be any of the following:

***<show_type>* values for dialogs and controls**		
this name	**or this letter**	**#define'd symbol**
show	s	SW_SHOW
1	1	SW_SHOW
hide	h	SW_HIDE
0	0	SW_HIDE

If your <*target*> is a dialog, *not* a control, <*show_type*> may also be:

<*show_type*> values for dialogs only		
this name	**or this letter**	**#define'd symbol**
restore	r	SW_RESTORE
minimize	i	SW_MINIMIZE
maximize	x	SW_MAXIMIZE
normal	n	SW_NORMAL
foreground	f	calls SetForegroundWindow
trayicon	-	show dialog icon in tray
traymin	-	show icon in tray and hide dialog
traydel	-	remove tray icon

You can use set_tooltip to set a tooltip on the tray icon.

There are other SW_ messages you can use as <*show_type*>: see **dialog-related_defines.txt**. If you want to use them, copy the variable declaration/assignment you want from there to your script, and use that variable as <*show_type*>.

Be careful: there's ambiguity of the **show** service has two arguments: In that case, if the second argument is 1 or 0, that's taken as showing or hiding the dialog; if it's a <*show_type*> name in the above table (e.g. "hide" or "show", but there's a control with that *control id*, it's taken as showing that control, not as applying that <*show_type*> to the dialog as a whole.

<*script_to_call*> Optional, only applies for <*show_type*>s: A PowerPro command to execute if an interesting event is fired by the control. See Section 11.5 "Script Calls and Their Arguments". If absent or the null string, clicking on the tray icon for the dialog will cause the dialog to show in the foreground..

<*command_arg*> Optional: An argument passed to the above command if it's invoked. If present, the <*script_to_call*> must not be the null string. See Section 11.5 "Script Calls and Their Arguments".

<icon_path>, *<icon_number>* Optional: Icon you want to use show in the tray. For details see Section 11.8 "Icons: the *<icon_path>, <path_to_image>* and *<icon_number>* parameters". If you omit *<icon_path>* and *<icon_number>*, the icon associated with the dialog via the equivalent arguments of dialog.define, or your most recent call to set_icon.

10.8 enable

dialog.enable(<*target*> [, <*enable_or_not*>]**)**

Used in sample scripts dialogPluginDemo1.powerpro, dialogPluginDemoFromConfigFile.powerpro and all the regex scripts.

dialog.enable() enables or disables a control.

You must only invoke enable *after* a dialog is created or run.

If applied to a control (not to the dialog as a whole), you can chain other service to **enable** calls.

<*target*>**:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". If using <*dialog_handle*>*,* <*ctrl_id*>*,* <*ctrl_id*> may be 0 or absent, indicating you wish to enable or disable the dialog itself. If <*ctrl_id*> is a control tab, all controls associated with *all* tabs will be enabled or disabled as well.

<*enable_or_not*> Optional: 0 to disable, 1 to enable, if omitted taken as 1 (enable).

10.9 set_focus

dialog.set_focus(<*target*>**)**

alias: focus

Used in sample scripts dialogPluginDemo1.powerpro, regexDialog.powerpro and regexDialogScintilla.powerpro.

dialog.set_focus() causes focus to switch to a particular control.

You must only invoke set_focus *after* a dialog is created or run.

If applied to a control (not to the dialog as a whole), you can chain other service calls to **set_focus** calls.

<*target*>**:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". If using <*dialog_handle*>*,* <*ctrl_id*>*,* <*ctrl_id*> may be 0 or absent, indicating you wish to set focus to the dialog itself. Doesn't work with a <*window_ handle_ to_ remote_ control*> <*target*>.

10.10 get_value

dialog.get_value(<*target*> [, <*property*>]**)**
dialog.get_value(<*dialog_handle*> [, "font" | "hwnd" | "prevctrl" | partno]**)**
***<dialog_handle>*.get(**["font" | "hwnd" | "prevctrl" | partno]**)**
***<dialog_handle>*.get_value(**<*target*> [, <*property*>]**)**
***<dialog_handle>*.get(**<*target*> [, <*property*>]**)**
x = ***<dialog_handle>*[**<*target*> [, <*property*>]**]**
dialog.get_value(<*window_handle*> [, "font"]**)**

alias: get

Used in the regex sample scripts.

dialog.get_value() gets the some value from a control or the dialog. What that means depends on the type of control:

page 60 of 260

if invoked on:	you get:
the dialog	it's caption
button	label on button
check box radio button three-state	0 if unchecked, 1 if checked/selected, 2 if indeterminate. If <*property*> begins with "text", gets the control's text instead
static	text in static
editbox scintilla	get text in box or it's font; scintilla has small variation in get_value with a <*property*> of "font". A <*property*> of "select" will get selected text; "selectlen" will return how long it is.
richedit	fetch either plain or rtf text in box. details
combo box	a little complicated. details
list box	a little complicated. details
date-time	date selected, in format yyyymmdd; if nothing selected (0nly possible if control has "shownone" style), 0
month-calendar	date selected, in format yyyymmdd; if control has the "multiselect" style, you get two dates back, separated by a space, represented the selected range
scrollbar spinner progress slider	the control position
statusbar	text in part *partno*
treeview	complicated. details
listview	complicated. details

You *can't* chain service calls to **get_value**.

<*target*>**:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". If using <*dialog_handle*>*, <ctrl_id*>*,* <*ctrl_id*> may be 0 or absent, indicating you wish to get the dialog's title. If

<ctrl_id> is a control tab, all controls associated with *all* tabs will be enabled or disabled as well.

Be careful; if you have a control named "font" and use it as *<ctrl_id>*, it will be interpreted as querying the font of the dialog as a whole, not getting the value of the "font" control.

You can only invoke get_value taking a *<dialog_handle>* and a *<target>* *after* a dialog is created or run.

<property> Optional:

value	**applies to**			**returns**
	dialog	**control**	**hwnd**	
tool	FALSE	TRUE	FALSE	gets tooltip text
font	TRUE	TRUE	TRUE	returns a font description, in the format used by set_font: see Section 11.9 "Fonts: the *<font>* and *<font_spec>* parameters". If *<target>* is an hwnd and a handle to a control, works; if a handle to most other kind of windows, probably doesn't (because most windows procedures don't respond to the WM_GETFONT message).
fonth fonthandle	TRUE	TRUE	TRUE	returns the HFONT associated with the *<target>*; for hwnd *<target>* see above
hwnd	TRUE	TRUE	FALSE	returns the window handle for control; 0 if dialog not created yet
id	FALSE	TRUE	FALSE	returns control number
type	FALSE	TRUE	FALSE	returns the type of control
name	FALSE	TRUE	FALSE	returns name, if any, assigned to control in call to define_control
prevctrl	TRUE	FALSE	FALSE	returns a handle to the control that was last used (had keystroke entry) before one whose <*script_to_call*> is currently executing, or the null string if no control has focus. statics, scrollbars, progress bars, and status bars don't count as "previously used".
owner	TRUE	TRUE	TRUE	returns a handle to the owning dialog. No allowed if applied to a non-modal dialog not launched by another dialog

Other values are valid for specific control types; see descriptions of set_value for each control type in appendices.

The ***<dialog_handle>*[***<ctrl_id>, <property>***]** syntax works nicely; so does ***<dialog_handle>*[***<property>***]** and ***<control_handle>*[***<property>***]**.

10.11 set_value

dialog.set_value(<*target*>*, <text>*
[*, <pos>* | *<property>*| *<date2>* | *<tab control stuff>*] **)**
dialog.set_value(<*target*>*, <pos>*, <<*text*>) (for statusbar parts)
dialog.set_value(<*dialog_handle*>*, <text>***)**
***<dialog_handle>*.set_value(**<*ctrl_id*>*, <text>*
[*, <pos>* | *<property>*| *<date2>* | *<tab control stuff>*] **)**
***<dialog_handle>*.set_value(**<*pos*>, <<*text*>) (for statusbar parts)
***<dialog_handle>*.set_value(**<*text*>**)**

***<dialog_handle>*[**<*ctrl_id*>**]** = <*text*>
dialog.set_value(<*window_handle_to_control*>*, ….***)**

aliases: set, modify, add

Used in all (?) sample scripts. dialogPluginDemo1.powerpro uses the ***<dialog_handle>*.set_value(**<*text*>**)** form.
dialogPluginDemoNonNative.powerpro uses set services to modify controls in dialogs not produced by the dialog plugin.

dialog.set_value() sets the a value of a control or the caption of a dialog, as follows:

if invoked on:	**you set:**
the dialog	It's caption. *<pos>*/*<property>* is ignored.
button	Label on button. *<pos>*/*<property>* is ignored.
check box radio button	Sets the control's state: 0 to set to unchecked, 1 to checked/selected, 2 to indeterminate. If *<property>* begins with "text", gets the control's text instead. You can also change the state of a check box or radio button with the setcheck message.
static	Text in static. *<pos>*/*<property>* is ignored.
editbox scintilla	get text in box.
richedit	set either plain or rtf text. details
combo box	If *<pos>* is absent, sets what's in the edit box. If *<pos>* is "add", *adds* item(s) to list (at end unless listbox has "sort" style, otherwise in appropriate sort order). If *<pos>* is numeric (lowest legal value 0 or 1), item(s) are *inserted* at *<pos>* (if *<pos>* is –1, item(s) added at end; the "sort" style never applies). You can add or insert multiple items at once by separating them with newlines ("\n"). Returns the index (lowest legal value 0 or 1) of the position at which the last item was inserted or added., or –1 if there was an error.
list box	If *<pos>* is absent or "add", *adds* item(s) to list (at end unless listbox has "sort" style, otherwise in appropriate sort order). If *<pos>* is numeric (lowest legal value 0 or 1), item(s) are *inserted* at *<pos>* (if *<pos>* is –1, item(s) added at end; the "sort" style never applies). You can add or insert multiple items at once by separating them with newlines ("\n"). Returns the index (lowest legal value 0 or 1) of the position at which the last item was inserted or added., or –1 if there was an error.
date-time	The selected date; you must provide *<text>* in format yyyymmdd; if *<text>* is the null string, clears selection. *<pos>*/*<property>* is ignored.
month-calendar	if control has "multiselect" style: you must provide *<text>* as first date, *<date2>* as second, both in format yyyymmdd if not: the selected date; you must provide *<text>* in format yyyymmdd. *<pos>*/*<property>* is ignored.
scrollbar, spinner progress slider	The control position. *<pos>*/*<property>* is ignored.
statusbar	Text of all or part of a statusbar. see details.
tab control	Complicated; see details.

if invoked on:	**you set:**
treeview	Complicated; see details.
listview	Oh, so complicated. see details

depending on last call to set_base, or, failing that, on value of the configuration ini file (dialog.ini or plugins,.ini) key indexBase; or, failing either, 1

If applied to a control (not to the dialog as a whole), you can usually chain other service calls to **set_value** calls; the exception is when used with the "add" subcommand on TreeView controls.

*<target>***:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". If using the *<dialog_handle>, <ctrl_id>* option*, <ctrl_id>* may be 0 or absent, indicating you wish to set the title of the dialog itself.

You must only invoke set_value taking a *<dialog_handle>* and a *<ctrl_id> after* a dialog is created or run.

<pos> | *<property>*| *<date2>* Optional: as specified for a specific control, above

Note the ***<dialog_handle>*****[***<target>***]** = *<text>* syntax doesn't support the *<pos>* | *<property>*| *<date2>* parameter.

10.12 clear

> **dialog.clear(***<target>* [*, <item>*]**)**
> ***<dialog_handle>*****.clear(***<ctrl_id>* [*, <item>*]**)**
> **dialog.clear(***<hfont>, "hfont"***)**

alias: remove

Used in sample scripts dialogPluginDemo5.powerpro and dialogPluginDemo6.powerpro.

dialog.clear() removes tabs from tab controls; nodes from tree view controls; rows or columns from list views; and lines of text from combo box controls or list controls. Go to appropriate link to he control you want to use for further details.

*<target>***:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". *<ctrl_id>* must be an id of one of the above control types. Doesn't work with a *<window_ handle_ to_ remote_ control>* *<target>*.

If applied to a control, you can chain other service calls to **clear** calls.

If first argument is an HFONT returned by set_font, deletes the font resource associated with it.

If you try to delete an item that doesn't exist, you'll get an error message.

10.13 set_tooltip

alias: tooltip

Used in sample script dialogPluginDemo1.powerpro.

There are three kinds of tooltips;

- ones that appear when you hover a mouse over a control
- ones that appear when you hover a mouse over a tray icon associated with a dialog
- ones that appear in bubble form from a tray icon associated with a dialog when you request them.

10.13.1 set_tooltip for controls

For a control tooltip:

dialog.set_tooltip(<*ctrl_id*> [*,* <*tooltip_text*> [**,** <*tooltip_style*> [**,** <*font*>
[**,** <*script_to_call*> [*,* <*command_arg*> [,
<*foreground_colour*> [, <*background_colour*> [*,*<*delay*> [*,*
<*title*> [*,* <*icon*>]]]]]]]]]**)**

***<dialog_handle>*.set_tooltip(**[<*ctrl_id*>][*,*<*tooltip_text*>
[*,* <*tooltip_style*> [**,** <*font*> [**,** <*script_to_call*>
[*,* <*command_arg*> [, <*foreground_colour*>
[, <*background_colour*> [*,*<*delay*> [*,*<*title*>
[*,*<*icon*>]]]]]]]]]]**)**

ctrl_handle.set_tooltip([<*tooltip_text*> [**,** <*tooltip_style*> [**,** <*font*>
[**,** <*script_to_call*> [*,* <*command_arg*>
[, <*foreground_colour* > [, <*background_colour* >
[*,* <*delay*> [*,* <*title*> [*,* <*icon*>]]]]]]]]]**)**

NB Tooltips don't work in XP if the powerpro manifest file (powerpro.exe.manifest) isn't present. Will try and fix.

<*target*>**:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows".

If <*target*> is the dialog as a whole, the tooltip applies to the dialog as a whole; for now, in that case, only <*tooltip_text*> is processed.

Doesn't work with a <*target*> of type <*window_ handle_ to_ remote_ control*> .

If all arguments after <*ctrl_id*> are absent (or no arguments provided for **ctrl_handle.set_tooltip** form) the tooltip is removed.

<*tooltip_text*> Optional**:** Tooltip's text. Sets or changes text if not the null string. "\n" or "\r\n" can be used to force <*tooltip*> line breaks (or "/" with the "slashisnl" style)

<*tooltip_style*> Optional**:** Determines behaviour and appearance of tooltip. May contain any of the names or letters, whitespace-delimited specified in the *table of tooltip_styles*.

If <*tooltip_text*> is the null string but there are <*tooltip_style*>s, tooltip text will remain the same but tooltip styles will change.

<*font*> Optional**:** Only applies for control tooltips. Sets tooltip's font (which you can also do using the set_font service with the <*tooltip*> argument appropiately set).

<*script_to_call*> Optional**:** Only applies for control tooltips. A PowerPro command to execute just before a tooltip displays. See Section 11.5 "Script Calls and Their Arguments".

<command_arg> Optional**:** Only applies for control tooltips. An argument passed to the above command if it's invoked. If present, *<script_to_call>* must not be the null string. See Section 11.5 "Script Calls and Their Arguments"

<foreground_colour>, *<background_colour>* Optional**:** Only applies for control tooltips. See Section 11.7 "Colours". If XP visual styles are in force (because there's a powerpro.manifest file in your PowerPro installation folder) these parameters don't work.

<delay> Optional**:** in milliseconds, sets initial (i.e. same as "showafter" pseudo-style), and autohide (i.e. "stayopen" pseudo-style, set to ten times the initial time) durations of tooltip. If you specify –1, all delays are returned to defaults (double click time for initial, infinite for autohide). If you specify delays "stayopen" or "showafter" pseudo-styles as well as a *<delay>*, the *<delay>* applies first, then overridden with the specific "stayopen" or "showafter" values.

<title> Optional**:** Only for balloon tooltips. The title above the main text.

<icon> Optional**:** Only for balloon tooltips. See below.

NB: A bug, Only if (a) you activate XP Visual Styles for powerpro by having the powerpro.exe.manifest in your PowerPro installation folder (the file as installed has ".unused" appended; to activate it you have to remove the appendage), and (b) you execute **set_tooltip** after you invoke created or run, the tooltip may leave a shadow on screen until it's clicked. I'm working on it. Workaround: call set_tooltip before create, or don't use the manifest.

You can invoke **set_tooltip** for a control at any time.

10.13.2 set_tooltip for dialogs

dialog.set_tooltip(*<dialog_handle>* [*, <tooltip_text>* [**,** *<title>,* [*<icon>* *,* [*<delay>*]]]]**)**

***<dialog_handle>*.set_tooltip(**[*<tooltip_text>,* [*<title>,* [*<icon>* *,* [*<delay>*]]]]**)**

If all arguments after *<dialog_handle>* are absent (or no arguments provided for ***<dialog_handle>*** form) the tooltip is removed.

You can create two kinds of tooltip for the tray icon using.

- If you want **a standard tooltip that shows whenever you hover your mouse over the tray icon**, use only *<tooltip_text>*.
- If you want to display **a bubble notification tooltip originating from the tray icon**, you must include at least *<title>* parameter, even if it's the null string.

You have both types of tooltip for the same dialog.

<tooltip_text> Optional**:** Tooltip's text. Sets or changes text if not the null string.

If *<tooltip_text>* is the last argument supplied, assumption is you want to have that text as an ordinary rectangular tooltip that appears when you hover over a tray icon. If you haven't yet created a tray icon with call to the show service using the traymin or trayicon *<show_type>*s, a tray icon will be created using the dialog's icon.

<title> Optional**:** Only applies to balloon tooltips. Must be no longer than 99 characters; The title above tooltip text.

<icon> Optional**:** Only applies to balloon tooltips.

<icon> **:=** *<icon_type>* | (*<icon_path>* [, *<icon_number>*])

<icon_type> can be one of "info", "warning" or "error".

<icon_path> and *<icon_number>* Optional: Specify an icon to use in the balloon. Only works for XP SP2 or later.. For details see Section 11.8 "Icons: the *<icon_path>, <path_to_image>* and *<icon_number>* parameters".

<delay> Optional**:** Only applies to balloon tooltips. Delay (in msec) after which the bubble tip will be closed.

Other service calls can be chained to **set_tooltip** calls.

10.14 set_range

dialog.set_range(*<target>, <min>, <max>* [*,<other_thing>*]**)**
ctrl_handle.
<dialog_handle>.

alias: range

Used in sample scripts dialogPluginDemo1.powerpro, dialogPluginDemo2.powerpro, dialogPluginDemo4.powerpro, dialogPluginDemoFromConfigFile.powerpro

dialog.set_range() sets properties of a scrollbar, spinner, progress, or slider control .

You must only invoke set_range *after* a dialog is created or run.

Other service calls can be chained to **set_range** calls.

*<target>***:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". If using the *<dialog_handle>, <ctrl_id>* option*, <ctrl_id>* must be of a ontrol of one of the above types.

<min> Required: Value associated with the left or bottom end of a control. Defaults to 0. If *<min>* is specified, *<max>* must be also.

<max> Required: Value associated with the right or top end of a control. Defaults to 99.

<other_thing> Optional:

- For a scrollbar or slider *<page_size>:*The quantity determining the thickness of the scrollbar or slider thumb, classically the size of a page in a document. Defaults to 1. Should be in the range 0 to *<max>* - *<min>* +1.
- For a spinner: ignored
- For a progress bar: *<step>* The amount the bar moves when told to step. Defaults to 10 if omitted.

10.15 set_colour

dialog.set_colour(<*target*> [, "tooltip" | "tool" | "tip"][, <*foreground*> [, <*background*>]]**)**

***<dialog_handle>*.set_colour(**[<*ctrl_id*>] [, "tooltip" | "tool" | "tip"][, <*foreground*> [, <*background*>]]**)**

ctrl_handle.set_colour([, "tooltip" | "tool" | "tip"][, <*foreground*> [, <*background*>]]**)**

dialog.set_colour(<*target*>, <*mouse_state*> [, <*colour*>]]**)**
***<dialog_handle>*.set_colour(**[<*ctrl_id*>], <*mouse_state*> [, <*colour*>]]**)**
ctrl_handle.set_colour(<*mouse_state*> [, <*colour*>]]**)**

alias: colour, set_colours

Used in sample scripts dialogPluginDemo1.powerpro, dialogPluginDemo2.powerpro, dialogPluginDemo4.powerpro, dialogPluginDemo5.powerpro, dialogPluginDemo6.powerpro, dialogPluginDemo7.powerpro and dialogPluginDemoFromConfigFile.powerpro. dialogPluginDemo2.powerpro and dialogPluginDemo7.powerpro illustrate use of mouse state keywords

Use **dialog.set_colour()** to change foreground and/or background colours of a control or dialog.

You can invoke set_colour before or after a dialog is created or run, , though I'm not sure why you'd want to do the former.

Other service calls can be chained to **set_colour** if it applies to a control (as opposed to the dialog as a whole).

<*target*>**:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". If using <*dialog_handle*>, <*ctrl_id*>, <*ctrl_id*> may be 0 or absent, indicating you wish to change the colours of the dialog itself.

If <*target*> specifies a tab control, animation or spinner or activeX control, **set_colour** has no effect. In the latter case, the COM control you've embedded may have a method for dealing with colour.

If <*target*> specifies a static, and a <*mouse_state*> is specified, the control has to have the "notify" (SS_NOTIFY) style, so that's automatically applied if you specify a <*mouse_state*>. Example in dialogPluginDemo7.powerpro.

"tooltip" | "tool" | "tip" Optional: only applies if <*target*> is a control; in which case you'll be altering the colour of a tooltip, which must already exist (i.e. have been created via a call to define_control or set_tooltip). But if XP visual styles are in force (because there's a powerpro.manifest file in your PowerPro installation folder) you can’t change a tooltip’s colours, so these options won’t work.

<foreground> Optional: The text colour. If absent or the null string, colour reverts to standard system colour. If "default" or the empty string, becomes default text colour for the type of control. See Section 11.7 “Colours”. Ignored if target is 0, i.e. the dialog itself. For date picker, month-calendar, treeview and listview controls, this parameter can be interpreted as the part of the control you wish to colour. Has no effect on scrollbar controls.

<background> Optional: The background colour. If absent or the null string, colour reverts to standard system colour. If "default" or the empty string, becomes default background colour for the type of control.. See Section 11.7 “Colours”. For date picker, month-calendar, treeview and listview controls, this parameter can become the one colour you wish to set part of the control to.

<*mouse_state*> Optional: Sets a colour for a control dependent on what the mouse is doing with it. Must contain one of:

Must contain one of:		
this keyword	**or this letter**	**applies when control**
pressed	p	has mouse press down
focus focused	c	has focus
hover hovering	h	has mouse over

and may contain one of		
this keyword	**or this letter**	**means**
back	b	background colour
fore	f	foreground colour

If a colour is defined when a mouse is hovering over a control, it takes precedence over one defined for the control having focus.

If neither "fore" nor "back" are specified, "back" is assumed.

Not all <*mouse_state*>s work on all controls:

scrollbar controls used as pseudo spinners won't show any effects, because there is no "foreground" and the background is usually hidden

datetime controls don't seem to respond to anything but "press" events; might be better with the "updown" style.

combobox controls react oddly to focus events

statusbars don't do foreground colours; tab controls and spinners don't do colours at all; can't get scintilla controls to do control-wide colouring; activex is a container, container must respond to mouse movement; groups don't receive mouse events.

statics have to have the "notify" (SS_NOTIFY) style, to respond to mouse presses, so that's automatically applied if you specify a <*mouse_state*>.

10.16 rgb

dialog.rgb(*<red_value>, <green_value>, <blue_value>***)**

Used in sample scripts dialogPluginDemo1.powerpro, dialogPluginDemo2.powerpro, dialogPluginDemo4.powerpro, dialogPluginDemo5.powerpro, dialogPluginDemo6.powerpro and the regex scripts.

dialog.rgb() can be used to generate an a value for the *<foreground>* or *<background>* parameters. of set_colour or define_control. The arguments must be in the range 0-255.

You can invoke rgb at any time.

10.17 set_image

Use **dialog.set_image()** to set an image for a static control, button, or tab control:

for a static control:

dialog.set_image(<*target*> [, <*path_to_image*>
[*,* <*icon_number*>]]**)**

for a tab control:

dialog.set_image(<*target*> [, <*path_to_image1*>
[*,* <*icon_number*>]*…*[,<*path_to_imageN*>
[*,* <*icon_numberN*>]]]**)**

for a button:

dialog.set_image(<*target*> [, <*path_to_image*>
[*,* <*icon_number*>] [,*…*<*path_to_pressed_image*>
[*,* <*icon_number*>]]]**)**

aliases: set_images, images, image, images

Used in sample scripts dialogPluginDemo1.powerpro, dialogPluginDemo2.powerpro, dialogPluginDemo4.powerpro, dialogPluginDemo7.powerpro (settings images for buttons) and dialogPluginDemoFromConfigFile.powerpro.

You must only invoke set_image *after* a dialog is created or run. I've found some kind of glitch: if you invoke set_image when a dialog is running in foreground, the dialog loses focus. If appropriate, might be better for now to run set_image after a create and before a run.

Other service calls can be chained to **set_image** calls.

<*target*>**:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". You must use the <*dialog_handle*>*,* <*ctrl_id*>*,* form, and <*ctrl_id*> must be a *control id* (see Section 11.3 "Control Ids") for a static control or tab control.

<*path_to_image*> Required: May be a path to an file containing an icon (.dll, .ico, .icl, .scr, .cpl or .exe); or a .bmp, .gif, .wmf, .emf .jpeg/jpg or .cur file. If not an absolute path, taken to be relative to the location of the current configuration file (the folder returned by pprofolder). If the null string, treated as specifying PowerPro.icl in the same folder as PowerPro.exe. If none of the above and

one of the keywords listed in Section 11.8 “Icons: the *<icon_path>, <path_to_image>* and *<icon_number>* parameters”, a system icon will be used.

For all types of controls, if you invoke **set_image** with no arguments it will remove all images from the control in question.

<icon_number> Optional: The icon within the file *<path_to_image>* to use; base depends on last call to set_base, or, failing that, on value of the configuration ini file (dialog.ini or plugins,.ini) key indexBase; or, failing either, 1. Ignored if *<path_to_image>* is neither the null string nor a path to a .dll, .icl, or .exe. If omitted or the null string, taken to be lowest possible current base value.

static controls:

Images will be automatically resized to fit the static control.

There's a useful centreimage style for your target.

I haven’t figured out how to erase an image once it’s drawn in a static control.

tab controls:

set_images can be used to add images to the collection of images available to be placed on tabs within the tab control. In any one call to **set_images** you can specify as many *<path_to_image>*s (and intervening *<icon_number>*s) as you can fit into the 23 allowed parameters for a service call. You can call **set_images** for the same tab control as many times as you want, adding more and more images to the control's image set.

buttons:

set_images can be used to set images for a button, to the collection of images available to be placed on tabs within the button. In any one call to **set_image** you can specify one or two *<path_to_image>*s (and optional *<icon_number>*s); the second image, if present, applies when the button is pressed. If you omit the second image, the one specified image is used no mater the button’s state.

Calling **set_image** with a null string will stop an image being used in the relevant button state.

There are special "styles" that determine how image and text are displayed on a button. If the text and image can’t both be fit on a button given it's image styles, the image will be shrunk to fit. For now, I don’t attempt to resize the image when you resize the dialog.

10.18 set_response

dialog.set_response(<*target*> [, <*script_to_call*> [, <*command_arg*> [, <*events*> | <*mouse_event*> | <*sys_command*> | “tray” | “trayicon”]]]**)**

***dlg_handle*.set_response(**[<*ctrl_id*>,][<*script_to_call*> [, <*command_arg*> [, <*events*> | <*mouse_event*> | <*sys_command*> | “tray” | “trayicon”]]]**)**

***ctrl_handle*.set_response(**[<*script_to_call*> [, <*command_arg*> [, <*events*> | <*mouse_event*>]]]**)**

aliases: response

Used in sample scripts dialogPluginDemo1.powerpro, regexDialog.powerpro and regexDialogScintilla.powerpro

Use **dialog.set_response()** to change which command is executed when a control receives a notification; what argument to pass with that command when the notification occurs; and which notifications (sub-messages of the WM_COMMAND command) trigger that command.

You may invoke **set_response** even before a dialog is created or run.

Other service calls can be chained to **set_response.**

<*target*>**:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". You must use the the <*dialog_handle*>*,* <*ctrl_id*> form; <*ctrl_id*> may be absent or 0, indicating you wish to modify the responses of the dialog itself.

If the <*target*> is a scintilla, rich text edit or plain edit control, the bad news is that a “right” <*mouse_event*> will prevent the standard edit context menu (“Undo”, “Copy”, “Cut”,…) from appearing (*but* a “right” <*mouse_event*> with a keyboard modifier like “ctrl” won’t suppress it). The good news is that you can pinch a replacement menu that does much the same defined as part of the makeFormatMenu function of **regexDialog.powerpro** and **regexDialogScintilla.powerpro**.

If the <*target*> is a static, <*mouse_event*>s will be ignored unless the control has the "notify" (SS_NOTIFY) style, so that's automatically applied if you specify a <*mouse_event*>.

If <*ctrl_id*> specifies an activeX control, **set_response** will have no effect (because all events occurring within an activeX control happen within the COM event framework, in front of which dialog plugin goes all shy .

<*script_to_call*> Optional: A PowerPro command to execute if an interesting event is fired by the control. See Section 11.5 “Script Calls and Their

Arguments”. If absent or the null string, the control or dialog will cease to have a script to execute and will therefore never respond to notifications.

<*command_arg*> Optional: An argument passed to the above command if it’s invoked. If present, the <*script_to_call*> must not be the null string. See Section 11.5 “Script Calls and Their Arguments”. If absent or the null string, the control or dialog won't pass an argument to <*script_to_call*> .

<*events*> Optional: A list of notifications (in numerical form) accompanying a WM COMMAND or WM_NOTIFY message that may cause <*script_to_call*> to be executed. See Section 11.6 “Specifying Which Messages Are Responded To”. If absent or the null string: For a control: the control won’t have its own notification list, and will therefore begin to use the dialog's notification list, if there is one, or, if not, the default notification for the control class. For a dialog: removes the dialog's notification list.

<*mouse_event*> Optional: Instead of specifying notifications accompanying messages, you can specify any one of the following mouse button identifiers, and optionally, and separated from it by white space, a keyboard modifier:

which mouse button		**keyboard modifier (either)**		**keyboard modifier (left)**		**keyboard modifier (right)**	
name	**or letter**	**name**	**or letter**	**name**		**name**	
left	l	ctrl	c	lctrl leftctrl		rctrl rightctrl	
middle *	m	shift	s	lshift leftshift		rshift rightshift	
right	r	alt *	a	lalt leftalt *		ralt rightalt *	
mouse4 *							
mouse5 *							

* alt doesn't work for me; might be something odd about how alt keypress is dealt with in a dialog, as it's involved in shortcut key recognition

I wouldn't use "left" on it's own, without modifiers: that's the same as specifying the normal left-click action that most controls respond to. I'd use the numeric value of the usual notification (BN_CLICK, for instance, for buttons).

Two other keywords are possible:

enter	response taken when mouse enters control's window
exit	response taken when mouse leaves control's window

The most obvious use the with a <*mouse_event*> is to trigger context menus. However, those don't work for controls that have their own built-in context menus (as as as I can find out, that's editboxes and monthcal controls). For them, even a response to "right" with a keyboard modifier doesn't happen, even if the response doesn't involve a menu. But "middle" (with or without modifiers) works fine for such controls. I can probably bypass this limitation by subclassing the control types in question, but not sure it's that important.

You can define multiple <*mouse_event*>s for the same control or dialog, as well as a response to notification <*events*>.

If you call with a combination of <*target*> and <*mouse_event*> that's already been used in a previous call, <*script_to_call*> and <*command_arg*> replace the values specified in that previous call. If both parameters are the null string, the response <*target*> and <*mouse_event*> ceases.

For statics, if you want a static to respond to <*mouse_event*>s, it's got to have the notify (SS_NOTIFY) style, so that's automatically added to any static you apply a **set_response** with a <*mouse_event*> to.

page 81 of 260

<*sys_command*>: only valid if your <*target*> is a handle to a dialog, can be any of the following keywords, corresponding to user actions on the dialog:

keyword	WM_SYSCOMMAND wParam	remarks
close	SC_CLOSE	user hits alt-F4, or the "X" box in title bar, or "Close" option in system menu
size	SC_SIZE	user changes size of dialog using sys menu or border
move	SC_MOVE	user moves dialog using sys menu or title bar; used in dialogPluginDemo2.powerpro
minimize	SC_MINIMIZE	user minimises; used in dialogPluginDemo2.powerpro
maximize	SC_MAXIMIZE	user maximises
restore	SC_RESTORE	user restores
default	SC_DEFAULT	use double clicks on title bar

If you want most of these responses to be possible, you'll of course have to define your dialog with the appropriate styles so that it has e.g. a system menu ("sysmenu"), or a minimize box ("minbox"), or a resizeable.border ("thickframe").

If there's no "close" response set, if user closes by any of the methods listed, action associated with a control with "cancel" <*id*> will be done instead.

If you use set_position to change a dialog's size or position, that won't trigger any <*script_to_call*> associated with "move" or "size". Whew.

"*tray*" or "*trayicon*": only valid if your <*target*> is a handle to a dialog. Specifies what happens when a user clicks on a tray icon associated with the dialog. Calling **set_response** with "*tray*" or "*trayicon*" will create a tray icon if not already created with show service, using the dialog's icon. If you provide no response for a tray icon (either as arguments of the show service or by call to **set_response**) Clicking on a tray icon will unhide the dialog and set focus on it. If you provide a response, suggest most typical would be to open a menu, probably defined using the Powerpro cl services. See for instance the onTrayRight function of dialogPluginDemo1.powerpro.

10.19 change_style

dialog.change_style(<*target*>, <*styles*> [, <*bAdd*>]**)**

alias: style

Used in sample scripts dialogPluginDemo1.powerpro and dialogPluginDemoFromConfigFile.powerpro.

Use **dialog.change_style()** to change the styles applied to a control or the dialog.

You must only invoke change_style *after* a dialog is created or run.

Other service calls can be chained to **change_style** if it applies to a control (as opposed to the dialog as a whole).

<*target*>**:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". If you use the <*dialog_handle*>, <*ctrl_id*>, form, <*ctrl_id*> may be absent or 0, indicating you wish to modify the style of the dialog itself. The <*window_ handle_ to_ remote_ control*> is valid but not recommended, as it will change a control's behaviour without the knowledge of its owner.

<*styles*> Required: Keywords or letters, or numerical values modifying the control's or dialog's styles. Some keywords apply to all controls: see this table. Other keywords work only for a specify type of control: see the appropriate subsection of Appendix II. Keywords for dialogs are here. For Further details see Section 11.4 "Styles: the <styles> parameter".

<*bAdd*> Optional: If 1 (the default if omitted): the styles specified are added to the current styles of the control or dialog. If 0, the specified styles are removed the target.

10.20 set_icon

dialog.set_icon(<*dialog_handle*>, <*icon_path*>[, <*icon_number*>]]**)**

aliases: icon

Used in sample scripts dialogPluginDemo1.powerpro and dialogPluginDemoFromConfigFile.powerpro.

dialog.set_icon changes a dialog's icon.

You must only invoke set_icon *after* a dialog is created or run.

You *can't* chain other service calls to **set_icon** calls.

<*dialog_handle*> Required: A handle returned by dialog.define. See Section 11.1 "Handles to Dialogs and the <*dialog_handle*> parameter".

<icon_path> and *<icon_number>* select which icon you want to use. For details see Section 11.8 “Icons: the *<icon_path>, <path_to_image>* and *<icon_number>* parameters”.

10.21 set_font

dialog.set_font(<*target*>*,* <*font*> [, <*tooltip_flag*>]**)**

alias: font

Used in sample scripts dialogPluginDemo1.powerpro, dialogPluginDemo3.powerpro, controlFontChanger.powerpro, regexDialog.powerpro and regexDialogScintilla.powerpro

Use **dialog.set_font()** to set the font of a control, a dialog and some other windows.

You can invoke set_font at any time, even before a dialog is created or run.

Other service calls can be chained to **set_font** calls, providing it's invoked to alter the font of a control, not the of the dialog as a whole or of a window handle.

<*target*>**:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". If you use the <*dialog_handle*>*,* <*ctrl_id*>*,* form, <*ctrl_id*> may be absent or 0, indicating you wish the font applied to all the dialog's controls. Windows always uses the system font for the dialog box title; executing a **set_font** with a zero target does not change this.

If <*target*> specifies an activeX control, **set_font** will have no effect.

If you use the <*window_handle*> form of <*target*>, you can alter the font displayed by a control belonging to powerpro (but not – safely at least -- of any other process, not even to pproconf.exe). E.g. for a window launched by the Powerpro inputDialog command, e.g. by the sample script controlFontChangerTest.powerpro) controlFontChanger.powerpro includes a generalised version of this code:

```
local sList = win.childhandlelist(<dialog title>, "c=Edit")
local n = word(sList, 0)
for (local i = 1; i le n; i++)
  local hWin = word(sList, i)
  if (win.visible(hWin))
    dialog.set_font(hWin, "arial 12")
endfor
```

If you use the <*window_handle*> form of <*target*>, set_font returns the HFONT associated with the window after set_font runs.

If you specify a <*window_handle*> to a window other than a control, probably won't work; most windows produecures other than those of controls don't deal with the WM_SETFONT control (or equivalents for richedit and scintilla controls), so won't respond to set_font.

<*font*> Required: See Section 11.9 "Fonts: the <*font*> and <*font_spec*> parameters"

page 85 of 260

If you specify only some properties of a font (font size, for instance, or weight), **set_font** will endeavour to preserve all some font properties of whichever control or window you're applying **set_font** to, not including stuff like bold and italic (the aspects of a font covered by the <*options*> component of a <*font_spec*>.

<*tooltip_flag*> Optional: if present and beginning with "t", changes the font of associated tooltip. Error if no tooltip has been defined by either parameters of define_control or by a call to set_tooltip; or <*target*> is the dialog (0) or a static control.

10.22 set_position

dialog.set_position([<*target*>] *,* <*X*> [, <*Y*>, [<*width*>, [<*height*>]]]**)**

dialog.set_position(<*target*>, *<dimensions>***)**

alias: position

Used in sample scripts dialogPluginDemo1.powerpro and dialogPluginDemo2.powerpro.

Use **dialog.set_position()** to change the position or size of a control or the dialog.

You can invoke set_position at any time, even before a dialog is created or run.

Other service calls can be chained to **set_position** calls, providing it's invoked to alter the position or size of a control, not of the dialog.

<*target*>**:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". If you use the <*dialog_handle*>*,* <*ctrl_id*>*,* form, <*ctrl_id*> may be absent or 0, indicating you wish to position the dialog as a whole. May work with a <*window_ handle_ to_ remote_ control*>*.*

<*X*>, <*Y*>, <*width*>, <*height*> (or *<dimensions>* in their place): The new position and/or size required of the control or dialog. The new position/size may be relative to the old ones. You can't use a resizing marker on a dimension when calling **set_position.** <*Y*>, <*width*>, <*height*> are optional and taken as 0 if absent; but if <*target*> is absent or zero, implying you're resizing the dialog, they must all be present (or *<dimensions>* in their place).

If you reposition the dialog, any and all sizing modifiers applied to the dialog or controls will apply; depending on them controls may resize with the dialog.

A dialog doesn't need to have the "thickframe" style for set_position to work. The "thickframe" style is needed only if you want to resize by dragging.

set_position is illustrated in the sample script dialogPluginDemo2.powerpro.

10.23 get_last_clicked

dialog.get_last_clicked()

aliases: ***last_clicked, clicked***

Returns the handle of the last control or dialog which responded to a user action, either by a user-defined mouse click, a normal left click, or a keyboard shortcut.

10.24 send_message

dialog.send_message(*<target>, <message>* [*, <wParam>* [*,<lParam>*
[*,<lParam_type>* [, *<bRedraw>*]]]]**)**

aliases: ***message, send***

Used in sample scripts dialogPluginDemo1.powerpro, dialogPluginDemo2.powerpro, dialogPluginDemo4.powerpro, dialogPluginDemoFromConfigFile.powerpro and all the regex dialogs.

dialog.send_message() allows you to send a message (using SendMessage) to a control.. It returns the LRESULT returned by SendMessage as an integer. Let me know if you run into a SendMessage variant that returns something else that you want to get at.

You must only invoke send_message *after* a dialog is created or run.

You *can't* chain other service calls to **send_message** calls (because **send_message** often returns a result).

*<target>***:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". If you use the *<dialog_handle>, <ctrl_id>,* form, *<ctrl_id>* may be absent or 0, indicating you wish to send a message to the dialog itself.. I'd use win.sendmessage service in most cases if you want to use the *<window_ handle_ to_ remote_ control>* form; the only advantage of using **dialog.send_message** instead is that can use the message keywords appropriate for each type of control.

<message> Required: the message to send.

Almost every type of control has a large number of legal messages that can be sent to it. You'll need to control Microsoft documentation to find what messages are legal for which type of control, and how each one works.

page 88 of 260

I've provided **named messages** for common messages appropriate for some types of control (buttons, check buttons and radio buttons, list controls and combo box controls, spinners, edit controls, sliders, progress bars, date-time pickers, month-calendar controls, statusbars, tab controls, treeviews, listviews and rich edit controls). Those definitions specify whether <*wParam*> or <*lParam*> are meaningful for each named message, and whether <*lParam*> is to be treated as a string integer, or pointer to a array or struct.. If the tables indicate that <*wParam*> or <*lParam*> *aren't* meaningful, the appropriate parameter can be omitted (and will be forced to zero). If <*lParam*> is meaningful, it will be treated as follows:

expected type as specified in named message	will be treated
integer	always IN (to the control targeted by the sent message)
string	IN unless a variable name provided, in which case may be either IN to or OUT from the control targeted by the sent message. If it's in OUT buffer, it's your responsibility to ensure that the variable named is long enough to accept the data that will be put in it.
composite	IN or OUT; you must provide a handle to a struct or array of the appropriate type and size (as returned by the create_struct or create_array services of the dll plugin)

In almost all cases where is valid, it's a DWORD. I've found a few cases where a pointer to a struct or array is required. In those cases, just provide a handle to the appropriate composite (as returned by the create_struct or create_array services of the dll plugin).

If you want to send a message which is not named for a particular control you can; you'll need it's numeric value, most of which you'll find in dialog-related_defines.txt (or, for scintilla controls, where just about everything is driven by messages, see scintilla-related_defines.txt). If you find the one you want there, you can paste the relevant local declaration into your script, e.g.

```
local BM_SETCHECK = 0x00F1
```

And then use it in your send_message call. If you can't find a declaration for the message you want, you'll have to look up it's numeric value yourself in Microsoft documentation.

<*wParam*> Optional: The wParam argument for the message. Always interpreted as a number. Taken to be zero if absent.

<lParam> Optional: The lParam argument for the message. May either be interpreted as a string, number, array or struct, depending on the following parameter. Taken to be zero or the null string if absent.

<lParam_type> Optional: How to interpret the *<lParam>* parameter:

value (case insensitive)	**means**
1, or begins with "s"	string
0, or begins with "n"	number.
begins with "c"	handle of array or struct generated by dll plugin

If *<lParam_type>* is absent and , not specified differently for a named message, *<lParam>* is taken as a string, unless it conforms to the pattern for an array or struct handle, in which case it's taken as such.is a string) .

<bRedraw> Optional: If present and 1, control is redrawn after message is sent.

10.25 browse_for_file

dialog.browse_for_file([<*dialogHandle*>,] <*type_and_options*> [,
<*initialFolder*>
[, <*title*> [, <*defaultExtension*> [, <*filter*>]
[, <*defaultFileName*> [, <*varNameROstate*>]]]]]]**)**

aliases: ***none***

Used in the main regex sample scripts regexDialog.powerpro and regexDialogScintilla.powerpro, and in browse_for_file.powerpro.

Brings up the "File Open" or "File Save As" common dialogs; if user makes a choice, returns the full path to the file chosen. In either case, no file is either saved or opened: it's up to you to do that with the path returned by the service.

You *can't* chain other service calls to **browse_for_file** calls.

<*dialogHandle*> Optional: If present, the dialog pointed to by the handle becomes the owner of the "File Open" or "File Save As" common dialog.

<*type_and_options*> Required: <*type_and_options*> := <*type*> [<*options*>]

<*type*>**:** A string beginning with **"o", "open"**, **"s", "save"**, **"m"** or "**multiple**". The latter means browse for a file to open, allowing the selection of multiple files.

By default, without any <*options*>:

- If you browse for a file to open, the folder and file(s) you choose must exist.
- If you browse for a file to save and the file you choose exists, you will be prompted whether you wish to override the file chosen.

If you pick a file or files, the service returns:

- If you specify <*type*> "**m**" or "**multiple**", a string that starts with the path of the folder in which files were selected, followed by a newline character, followed by successive filenames chosen in that folder, separated by newline characters. Use the PowerPro line() function to disassemble. This applies even if you select just one file, so the result will always differ from what happens…
- If you specify any other <*type*>, a string with the full path to the one chosen file.

<*type*> can be followed by (but separated by whitespace from):

<*options*>: any combination of the following keywords, separated by whitespace:

keyword	**OPENFILENAME flag**	**means**
createprompt (Open dialog only)	OFN_CREATEPROMPT	If the user specifies a file that does not exist, this flag causes the dialog box to prompt the user for permission to create the file. If the user chooses to create the file, the dialog box closes and the function returns the specified name; otherwise, the dialog box remains open.
filecanbeabsent (Open dialog only)	~OFN_FILEMUSTEXIST	Without this keyword, the user can type only names of existing files in the File Name entry field. If this flag is specified and the user enters an invalid name, the dialog box procedure displays a warning in a message box. If this flag is specified, the OFN_PATHMUSTEXIST flag is also used.
pathcanbeabsent (Open dialog only)	~OFN_PATHMUSTEXIST	Without this keyword, the user can type only valid paths and filenames. If this flag is used and the user types an invalid path and filename in the File Name entry field, the dialog box function displays a warning in a message box.
hidereadonly (Open dialog only)	OFN_HIDEREADONLY	Hides the Read Only check box.
readonly (Open dialog only)	OFN_READONLY	Causes the Read Only check box to be checked initially when the dialog box is created. This flag indicates the state of the Read Only check box when the dialog box is closed.
allowoverwrite (Save dialog only)	~OFN_OVERWRITEPROMPT	Without this keyword, the Save As dialog generates a message box if the selected file already exists. The user must confirm whether to overwrite the file.
nochangedir	OFN_NOCHANGEDIR	Restores the current directory to its original value if the user changed the directory while searching for files.
nodereferencelinks	OFN_NODEREFERENCELINKS	Directs the dialog box to return the path and filename of the selected shortcut (.LNK) file. If this value is not given, the dialog box returns the path and filename of the file referenced by the shortcut

<*initialFolder*> Optional: The initial folder from which to browse. If absent or the null string, the system uses the current folder

<*title*> Optional: A title for the dialog. If absent or the null string, the system uses the default title (that is, “Save As” for <*type*> **s**; or “Open” for <*type*>s **o** or **m**).

<*defaultExtension*> Optional: the default extension. The service appends this extension to the chosen filename if the user fails to type an extension. This string can be any length, but only the first three characters are appended. The string should not contain a period (.). If this parameter is absent or the null string and the user fails to type an extension, no extension is appended to the chosen file.

<*filter*> Optional: A string containing pairs of filter strings; the separator between members or each pair and between pairs must be \n or \r. The first string in each pair is a display string that describes the filter (for example, “Text Files”), and the second string specifies the filter pattern (for example, “*.TXT”). To specify multiple filter patterns for a single display string, use a semicolon to separate the patterns (for example, “*.TXT;*.DOC;*.BAK”). A pattern string can be a combination of valid filename characters and the asterisk (*) wildcard character. Do not include spaces in the pattern string.

<*defaultFileName*> Optional: If present, this value appears in the dialog's "File Name" edit box.

<*varNameROstate*> Optional (only relevant for Open dialog): If present, and the user chooses a file, the value of the Open/Save Read Only check box (1 or 0) will be placed in the named variable.

10.26 choose_font

dialog.choose_font([<*target*> [,<*var_for_colour*>]]**)**
dialog.choose_font([*""* ,<*var_for_colour*>**)**
dialog.choose_font([<*font_spec*> [,<*var_for_colour*>]]**)**

aliases: *none*

Used in sample scripts dialogPluginDemo3.powerpro, controlFontChanger.powerpro and the main regex scripts regexDialog.powerpro and regexDialogScintilla.powerpro.

Brings up the font choice common dialog; if user makes a choice, returns a font description compatible with the <*font_spec*> parameter of set_font, define or define_control. See Section 11.9 "Fonts: the <*font*> and <*font_spec*> parameters". If no choice is made, or there's an error, returns an empty string.

You *can't* chain other service calls to **choose_font** calls.

<*target*> optional**:** See Section 11.1 "Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows". If you use the <*dialog_handle*>*,* <*ctrl_id*>*,* form, <*ctrl_id*> may be absent or 0, indicating you wish to initialise the font choice common dialog with the font of the dialog as a whole. <*target*> can also be a handle to any window.

If you supply <*target*>*,* **choose_font** will attempt to initialise the font choice common dialog from the font of the window or control you've specified.

*<font_spec>***:** If there's no control, dialog or window to provide a fonmt with which to initialise the font dialog, instead of a <*target*> you can supply a font specification

<*var_for_colour*>: optional: If present and the name of a variable *that has already been declared*, the RGB value for the colour chosen in the font choice common dialog will be placed in the variable named. The value found in the <*var_for_colour*> will be used to initialise the font colour in the font choice common dialog.

If you want to specify <*var_for_colour*> without specifying <*target*>, or *<font_spec>,* the first parameter must be a null string ("").

If careful when using a target of a <*dialog_handle*>, with no <*ctrl_id*> (in order to specify the dialog itself as the source of a font). You must not in that case choose a variable name that's also a valid <*ctrl_id*>, or it will be taken as a <*ctrl_id*>.

Only screen (not printer-only) fonts are displayed.

At the moment it's not possible to initialise the "script" (character set) in the font choice common dialog, but can add if required.

For an example of use, see the "Font" button in dialog generated by **dialogPluginDemo1.powerpro**.

10.27 destroy_window

dialog.destroy_window(<*dialog_handle*> [, <*value_to_return*>]**)**

aliases: ***none***

Used in the sample script dialogSubordinate.powerpro (which is used modal).

You must obviously only invoke destroy_window *after* a dialog is created or run.

dialog.destroy_window() tells the dialog plugin to delete the dialog window associated with a <*dialog_handle*>. If you've executed a destroy_window you can still execute dialog.run on the same handle later.

destroy_window() is safe to use from code that executes from modal dialog controls. <*value_to_return*> will be returned from the dialog.run call that started the modal dialog running.

You *can't* chain other service calls to **destroy_window** calls.

<*dialog_handle*> Required: A handle returned by dialog.define. See Section 11.1 "Handles to Dialogs and the <*dialog_handle*> parameter".

<*value_to_return*> Optional: Only used if the running dialog ism modal: a value to return as a result of the dialog.run(<*modal_dialog_handle*>,…) service call.

10.28 destroy

dialog.destroy([<*dialog_handle*> | <*handle_to_control*>]**)**

aliases: ***release***

Used in all sample scripts.

dialog.destroy() is used primarily to delete a dialog window and the data structure associated with a <*dialog_handle*>. Once you've destroyed a <*dialog_handle*>, you can't run it again later: the data structure's gone.

You must obviously only invoke destroy *after* a dialog is created or run.

destroy() is *not* safe to use from code that executes from modal dialog controls.

You *can't* chain other service calls to **destroy** calls.

<*dialog_handle*> Optional, but almost always present: A handle returned by dialog.define. See Section 11.1 "Handles to Dialogs and the <*dialog_handle*> parameter". If you omit it, you will destroy the last dialog created. destroy with no arguments is used by the dialog editor to terminate dialogs run without a harness script.

There's a limit on the number of active (i.e. not destroyed) dialog handles that can exist at once (64 in the current release), so you might want to destroy the ones you're finished with.

All running dialogs and their associated <*dialog_handle*>s will be automatically destroyed when you unload the plugin

Example:

```
local hDlg = dialog.define(0, 0, 100, 100, "Hi there", "minbox",….)
dialog.define_control(hDlg, 10, 20, 20, 12, "button",…)
dialog.define_control(hDlg, "button",…)
….
dialog.run(hDlg)
  ……
hDlg = dialog.destroy(hDlg)
```

Unlike some in some other plugins that emit handles, the dialog plugin doesn't automatically release <*dialog_handle*>s stored in local variables when a script finishes. That would be counterproductive, to say the least, for non-modal dialogs, where the lifetime of the dialog produced by the script will typically long exceed that of the script that created it.

If you do destroy a handle held in a local variable, best invalidate it by e.g. doing

```
my_dlg = dialog.destroy(my_dlg)
```

You can also use **dialog.destroy** to delete a <*handle_to_control*> returned by the dialog.make_ctrl_handle service.

10.29 destroy_all

dialog.destroy_all()

Not used in any sample scripts.

dialog.destroy_all() tells the **dialog** plugin to let go of all issued dialog **handles**.

A dialog.unload has the same effect.

10.30 export NOT YET IMPLEMENTED

dialog.export(<*target_file*>)

dialog.export allows you to create *<dialog_definition_file>*s from created dialogs.

The <*target_file*> extension determines the format the dialog is saved to. See *<dialog_definition_file>* for details.

If <*target_file*> already exists, it will be renamed to <*target_file*>.bak.

10.31 version

dialog.version returns the plugin version number as four digits, an assumed decimal point before the last two digits. So version 1.34 comes back "0134".

10.32 help

dialog.help opens the help file associated with the dialog plugin, if

- the key helpFileLocation in the config file dialog.ini/plugins.ini specifies a valid file path (rtf or chm or anything, really) location. If path isn't absolute, its taken as relative to the PowerPro installation folder. Or, if the key helpFileLocation isn't present,

- the file dialogPluginReadme.chm, dialog.chm, dialogPluginReadme.rtf or dialog.rtf is found in either the PowerPro installation folder or the plugins subfolder thereof (the files are searched for in the order given).

10.33 returns_values, returns_status, returns_nothing

These services determine what if anything services return as retval in:

```
returnval = dialog.service(….)
```

By default, data, if there is any to be returned, comes back as the result of the service expression (returnval in the above line). By default again, all services set a result string in variable **dialog_status**.

If you wish different behaviour, call either **returns_status()** or **returns_nothing()**.

If you call **returns_status()**, all subsequent service calls will return a result string as the result of the call. **dialog_status** wont be altered, and data will be returned via the variable **dialog_result**.

Many services return no data, only a result status, so they will *always* return that result status as the return value of the service, regardless of which of the returns_xxx services has been called.

Use **returns_values()** to return to default behaviour.

If you unload the plugin, its behaviour returns to the default in any subsequent call, i.e. further **dialog** service calls will return values.

page 98 of 260

10.34 error_dialog_on(), error_dialog_off()

Some **dialog** services can result in errors. For instance, you might try to restore from a non-existent ini file, or a variable might be defined in an ini file that wasn't declared, and should have been. In addition to such errors setting a PowerPro variable or returning a value with a status message the prefixed "ERROR:" (see previous section); they will also trigger the standard PowerPro script error dialog, allowing you to cancel all running scripts.

Under some conditions you might not want **dialog plugin** errors to be treated as scripting errors, and you would therefore not want to see the PowerPro script error dialog. If that's what you want (maybe because you're testing for the presence or absence of a section/key pair), invoke **dialog.error_dialog_off()**, or make sure the **raiseErrors** key in the configuration ini file is false or 0. Invoke **dialog.error_dialog_on()** to turn error dialogs back on after you turn them off.

If you unload the plugin, its behaviour returns to the default in any subsequent call, i.e. errors on further **dialog** service calls will cause the error dialog to pop up.

Invoking **dialog.error_dialog_off()** only affects error dialogs appearing when a **dialog** service call goes wrong. The normal Powerpro error dialog will appear if anything else goes wrong in Powerpro.

10.35 set_base, get_base

dialog.set_base(<*base*>)
dialog.get_base()

Used in all scripts to guarantee base used; illustration of use of base 0 in dialogPluginDemo4.powerpro.

set_base affects the interpretation of arguments that are an index integer, e.g. clear, and variants of get_value (alias *get*) and set_value (alias *set*) for listviews, combo boxs and list boxs, and for the latter, tab controls.

<base> can be 0 or 1. If not called, base for indexes is the value of the **indexBase** in the configuration ini file, or, if *that's* not present, 1.

get_base returns 0 or 1, the current effective base.

10.36 config

dialog.config(name_of_ini_file)

specifies a configuration ini file, with format, section and keys as described in Section 9.1

The ini file can either be given as an absolute path, or a path relative to the folder returned by the pprofolder variable (generally the folder containing the currently running powerpro configuration file).

Returns "OK" if file and found and there are no keys with illegal values, or a message beginning "ERROR:" if there is one.

If you unload and reload a plugin, its behaviour returns to the default or to that defined by a default config ini file (see Section 9.1)

11.0 Common Service Arguments

11.1 Specifying the Target of a Dialog service: Handles to Dialogs, Controls, and Windows

Since Powerpro can't itself store and remember dialog details, the dialog define service produces a *handle* to one. The dialog details are stored internally, within the plugin A handle is just a simple string beginning "d\x05" followed by a number from 3000 to 3255.

It's essential to know the handle returned by a call to dialog.define, because without it you can't add controls to it or further modify its properties.

Example:

```
local hDlg = dialog.define(0, 0, 100, 100, "Hi there", "minbox",….)
dialog.define_control(hDlg, 10, 20, 20, 12, "button",…)
dialog.define_control(hDlg, "button",…)
….
dialog.run(hDlg)
  ……
hDlg = dialog.destroy(hDlg)
```

The plugin can store (and provide handles for) up to 64 dialogs. Because there's a limit, you may wish to use the dialog.destroy service to let go of dialogs you no longer need.

Unlike some other plugins that emit handles, the dialog plugin doesn't automatically release <*dialog_handle*>s stored in local variables when a script finishes. That would be counterproductive, to say the least, for non-modal dialogs, where the lifetime of the dialog produced by the script will typically long exceed that of the script that created it.

Almost all other dialog plugin services take a <*target*> as a first argument or pair of arguments.

<*target*> := <*dialog_handle*>, <*ctrl_ids*> **|** <*window_ handle_ to_ remote_ control*> **|** <*handle_to_control*>

<*dialog_handle*> A handle returned by dialog.define, as described above.

<*ctrl_ids*> Required with a <*dialog_handle*>: see Section 11.3.

If you use <*dialog_handle*>, <*ctrl_id*> as a services <*target*> you can apply the service to the <*dialog_handle*> using the **handle.service** syntax: e.g. instead of

```
dialog.define_control(hDlg, "buttonname",…)
```

use

hDlg.define_control("buttonname",…)

<*window_ handle_ to_ remote_ control*>: instead of a <*dialog_handle*> and a <*ctrl_id*> many services can take a first argument of a (window) handle of a control, as long as the control's class and style is equivalent to one of the control types the dialog plugin knows about. Such a handle can be obtained from something like:

<*handle*> = word(win.childhandlelist(<*cl*>, "c=SysListView32"), 1)

You may have particular difficulty getting hold of a specific button, radio button, group or checkbox, as all these are of windows class "Button"; generally you'd have to get process each word in the list returned by

<*handle*> = win.childhandlelist(<*cl*>, "c=Button")

and use win.gettext to get each controls text. See **dialogPluginDemoNonNative.powerpro** for examples.

<*handle_to_control*> is returned by the dialog.make_ctrl_handle service, and as a result of most services when applied to a control that don't other data. These are interesting as locals in situations where you want to operate repeatedly on the same control in the same function, or, stored as a static, for controls that need to be reference in many different control handler scripts. Such handles, if stored in local variables, *are deleted when the local goes out of scope*.

If you call a service that returns a <*handle_to_control*> with multiple control ids, it will return the handle of the first control id encountered.

Scripting tip: *don't* unload the dialog plugin at the end of a script that either creates a dialog, or handles a response from it. It can take a few microseconds for a dialog to be taken down, and the plugin needs to be around while that happens.

If you do destroy a handle held in a local variable, best invalidate it by doing e.g.:

myDlgHan = dialog.destroy(myDlgHan)

Unloading the dialog plugin will cause all created dialogs to be destroyed.

You can chain other service calls that can a <*dialog_handle*> as a first argument after calls to dialog.define.

11.2 The *<dialog_definition_file>* Parameter and the Format of Definition Files

A *<dialog_definition_file>* defines a dialog's properties and contained controls. It can be activated using the create or run services.

A *<dialog_definition_file>* has two possible formats: one-line-per control or ini file.

11.2.1 The one-line-per control *<dialog_definition_file>*

Any *<dialog_definition_file>* with an extension other than ".ini" is taken to be one-line-per control.

A *<dialog_definition_file>* contains one line (usually the first one, but sometimes the second; see second bullet point below) to define the dialog's properties, followed by one line per control within the dialog.

The lines defining the dialog or controls are made up of fields, which correspond exactly to the parameters of dialog.define (to describe the dialog) and dialog.define_control) to describe a control). Fields are separated by the **field separator** character, which by default is the bar ("|"). You can specify another by:

- Providing another default character via the defaultFieldSeparator in the dialog.ini/plugins.ini configuration file.
- Providing a line preceding the line which defines the dialog in your *<dialog_definition_file>* containing just two characters which are non-numeric and not one of "+-pPmM", the first one of which will be taken as the default **fieldSeparator** character for the rest of the file, or;
- By ensuring that the first non-whitespace characters of any line in a *<dialog_definition_file>* that defines the dialog or a control are two characters which are non-numeric and not one of "+-pPmM", the first one of which will be taken as the **fieldSeparator** character for *that line*.

The one line to define the dialog has the following fields, corresponding to and described in the list of parameters for dialog.define:

<X>, *<Y>*, *<width>*, *<height>, <caption>* [, *<styles>* [, *<script_to_call>* [, *<command_arg>* [*, <events>* [, *<action_on_close>* [*, <icon_path>* [*, <icon_number>*[, *<font_spec>* [, *<right_click_command>*]]]]]]]] ***or***

<dimensions>, <caption> [, *<styles>* [, *<script_to_call>* [, *<command_arg>* [*, <events>* [, *<action_on_close>* [*, <icon_path>* [*, <icon_number>*[, *<font_spec>* [, *<right_click_command>*]]]]]]]]

Subsequent lines to define controls have the following fields, corresponding to and described in the list of parameters for dialog.define_control:

<X>, *<Y>*, *<width>*, *<height>*, *<control_type>* [*,<name>* [*,<text>* [, *<styles>*
[, *<script_to_call>* [, *<command_arg>* [*, <events>* [*, <foreground>,* [
<background>
[, *<id>* [, *<font_spec>* [, *<tooltip>* [, *<tooltip_style>*
[, *<right_click_command>*]]]]]]]]]]]] ***or***

<dimensions>, *<control_type>* [*,<name>* [*,<text>* [, *<styles>*
[, *<script_to_call>* [, *<command_arg>* [*, <events>* [*, <foreground>,* [
<background>
[, *<id>* [, *<font_spec>* [, *<tooltip>* [, *<tooltip_style>*
[, *<right_click_command>*]]]]]]]]]]]]

The only difference between a parameter and a field is that, unlike for dialog.define_control, there is no need to a *<dialog_handle>*, since the particular dialog controls are going to be added to is known.

One further issue: the fields within each line of the are normally taken as literal text. On occasion you may wish the dialog plugin services to interpret a field as a PowerPro expression. To cause that, make the first non-whitespace character of the field the evaluation marker, which is **#** by default.. You can change that to another default character via the **defaultEvaluationMarker** in the dialog.ini/plugins.ini configuration file. Or, you can change the expression marker character on a per-file or per-line basic using the techniques discussed above: the active expression marker character would follow the field separator character.

Another boring detail: if you want a tooltip with line breaks, you can't define it in a *<dialog_definition_file>*, since you can't include "\n" characters; you'll have to use set_tooltip instead. I can probably code for newlines if anyone really needs it.

The distribution includes dialogPluginDemo.txt as a sample one-line-per-control *<dialog_definition_file>*.

11.2.2 The ini file <*dialog_definition_file*>

Any <*dialog_definition_file*> with a ".ini" extension is taken to be just that -- an ini file.

The dialog itself is defined by the contents of the [DialogProperties] section, which has the following keys, corresponding to the equivalently-named parameters of the dialog.define service:

left	required unless dimensions is present
top	required unless dimensions is present
width	required unless dimensions is present
height	required unless dimensions is present
dimensions	if present, "left", "top", "width", "height" ignored
caption	required
styles	optional
scriptToCall	optional
commandArgument	optional
events	optional
actionOnClose	optional
iconPath	optional
iconNumber	optional
maxControls	optional
comment	optional
font	optional
commandMouseAction	optional
evaluationMarkerDefault	optional
evaluationMarker	optional
fieldDelimiterDefault	optional
fieldDelimiter	optional

The "evaluationMarkerDefault" key specifies a marker for expressions to be evaluated and overrides the default "#" for the entire ini file, unless in turn overridden overridden by:

The "evaluationMarker" key specifies a marker for expressions to be evaluated and overrides the default the "evaluationMarkerDefault" value if present ("#" otherwise), just for the [DialogProperties] section.

The "fieldDelimiterDefault" key specifies a field delimiter for the entire dialog description. It's only relevant if you switch back to a one-line-per control <*dialog_definition_file*> format.

The "fieldDelimiter" key specifies a field delimiter for the dialog description line. It's only relevant if you switch back to a one-line-per control <*dialog_definition_file*> format.

The "comment" key is only used to try and capture comments (lines beginning with a semicolon) just before the dialog definition line of a one-line-per control <*dialog_definition_file*>, so they can be restored if you convert back to it.

Each control must have a unique section name other than [dialogProperties]; if you create an ini file from a text file, section names will be assigned of the form [CONTROLnnnn] where nnnn is an integer starting at 1001.

Each such section has the following keys, corresponding to the equivalently-named parameters of the dialog.define_control service:

name	optional
controlType	required
left	required unless "dimensions" is present
top	required unless "dimensions" is present
width	required unless "dimensions" is present
height	required unless "dimensions" is present
dimensions	if present, "left", "top", "width", "height" ignored
textOrVar	optional
styles	optional
scriptToCall	optional
commandArgument	optional
events	optional
foreground	optional
background	optional
font	optional
id	optional
tooltip	optional (can’t include line breaks; use set_tooltip)
tooltipstyles	optional
commandMouseAction	optional
comment	optional
evaluationMarker	optional
fieldDelimiter	optional

The “evaluationMarker” key specifies a marker for expressions to be evaluated and overrides the default “#” if present. If not present but “evaluationMarker” is specified for the dialog, the dialog evaluation marker applies to all controls as well.

The “fieldDelimiter” key specifies a field delimiter for the line describing a control. It's only relevant if you switch back to a one-line-per control <*dialog_definition_file*> format.

The "comment" key is only used to try and capture comments (lines beginning with a semicolon) just before a control definition line in a one-line-per control <*dialog_definition_file*>, so they can be restored if you convert back to it.

The distribution includes dialogPluginDemo.ini as a sample ini-form <*dialog_definition_file*>.

11.2.3 Converting and creating <*dialog_definition_file*>s

You can use the export service to create a <dialog_definition_file>, or convert one to another format. To convert, just create a dialog from a file in one format, and export to the other.

Comments (lines beginning “;”) and inter-line spacing in ini files will be lost in conversion.

11.3 Control Ids

Anything that manipulates or queries a control or the dialog itself may need a *<ctrl_id>* parameter. This can be:

- for some services: 0, which signifies the service is to apply to the dialog itself, not one of it's controls. For some services this is the default, assumed if the *<ctrl_id>* parameter is absent. For such services (show, enable, focus, show, set_font, set_position, set_value, get_value (alias *get*), choose_font and set_response) you can omit the *<ctrl_id>* parameter even if further arguments follow, which means you can use a dialog handle like this: ***<dialog_handle>*.get** instead of having to do ***<dialog_handle>*.get(**0**)**).

- Otherwise, you must provide a control id, which is either:

- a number from 101 onward, or:

- the name of a control (i.e. the exact value you passed as the *<name>* parameter when you invoked dialog.define_control for the control in question). If you didn't give the control a non-null *<name>* parameter, you won't be able to find it by name.

- Some services (ones that change controls, but don't fetch their properties) allow multiple, white-space separated control names or numbers: enable, show, set_colour, set_value, set_position, change_style, set_font, clear, and send_message.

Using control names instead if numbers would probably be slightly slower, since I have to do a loop through all controls looking for a matching *<name>*. But it's not likely to be a noticeable delay unless you perform some service call requiring a name control id *very* frequently (in which case getting a *<handle_to_control>* using make_ctrl_handle). And names are probably a lot more convenient, especially if you've defined your dialog with a dialog configuration file.

11.4 Styles: the <*styles*> parameter

optional in: define
required by: define_control

The <*styles*> parameter consist of a series of one or more individual styles, separated by whitespace.

Each style is either a keyword or letter, or a hex or decimal constant.

Which style keywords and letters are valid depends on what you are applying styles to: the dialog itself, or a particular class of control. The valid possibilities are listed in tables in Sections 10.1 (for the define service) and 10.2 (for the define_control service).

Keywords and letters are translated into the appropriate constants, also listed in those tables. Some obscure styles might not be enshrined in keywords; for those you can use the style constants directly. I've copied some of those in the VC++ header files into the file **dialog-related_defines.txt** as local variable assignments like:

```
local SS_CENTER = 0x00000001
```

If you want to use any such variable declaration/assignments in your script, just copy the lines you need and paste 'em in.

If you're using constants like these in combination with other styles, you must make sure each style is delimited by white space:

```
local sStyles = "flat border " ++ SS_CENTER
dialog.define_control(hDlg, "static",  sStyles, 10,  280,  75,   30, sSt1Txt)
```

Styles Applying to all Controls		
this style name	**or this letter**	**to:**
3d	3	apply extended style WS_EX_CLIENTEDGE
modalframe	-	apply extended style WS_EX_DLGMODALFRAME
hidden	h	prevent application of style WS_VISIBLE
border	b	apply style WS_BORDER
notab	t	prevent application of style WS_TABSTOP: control won't be in tab sequence
group	g	apply style WS_GROUP defines the start of a group within a dialog box. All controls defined between two WS_GROUP styles are members of a group. Arrow keys can then be used to navigate between members of a group. All radio buttons in a group after the first should have the "notab" anti-style.

Numeric constants: If you're including a number in a string, observe the same rules PowerPro uses for numerics. Anything beginning 0x is treated as hex; anything else beginning 0 is octal; everything else is decimal.

11.5 Script Calls and Their Arguments: the *<script_to_call>* and *<command_arg>* parameters

optional in: define, define_control, set_response

PowerPro scripts to call can be specified for a control, a tooltip, or the dialog as a whole using the <*script_to_call*> *or* <*right_click_command*> parameters of define or define_control, respectively, or in calls to set_response They will most likely look like this:

```
scriptfileName
scriptfileName@label
```

You can use the PowerPro cb() or cbx() function e.g.

cb(<*full_path_to_scriptfile@label*>)
cb(<*full_path_to_scriptfile*>)

cbx("@*label*")

You can also indicate a command list within your pcf by prefixing <*script_to_call*> with "!":

```
!commandListName
!commandListName@label
```

Don’t attempt to specify arguments as part of <*script_to_call*>. This, for instance, won’t work as <*script_to_call*>:

```
scriptfileName(“urk”)
```

Don't prefix your script with a full stop.

You can use an absolute path to a script if you want.

The <*script_to_call*> you specify will fire if:

- the control with which it's associated generates an event (typically a WM_COMMAND, or WM_NOTIFY message with it's accompanying notification) for which the dialog plugin is monitoring.

  By default, only these events fire <*script_to_call*>:

  - buttons and the button variants (check box controls, radio buttons) respond to the BN_CLICK notification accompanying a WM_COMMAND message.

  - tab controls respond to the TCN_SELCHANGE notification sent with a WM_NOTIFY message.

  You can specify non-standard events for controls using the <*events*> parameter of define_control.

- …Or: a control or the dialog has an associated WM COMMAND, WM_NOTIFY or mouse event specified by a previous call to set_response.

- …Or tooltips call their <*script_to_call*> when they display.

The <*command_arg*> parameter is optional; it's value is passed to <*script_to_call*> when it's invoked: see the next section.

The length of <*script_to_call*> plus the length of <*command_arg*> must be less than 500, or you'll get an error message.

page 115 of 260

11.5.1 The Structure of Event-Handling Scripts

When a notification accompanying a WM COMMAND or WM_NOTIFY message, or a mouse event, causes a control to execute a <*script_to_call*>, the dialog plugin attaches six parameters to the call: so, a typical event-handling script might look like this:

Function onX(<*sUserArg*>, <*dlgHan*>, <*iCtrlNo*>, <*msg1*>, <*msg2*>, <*targHan*>)
local dialog_status
……
quit

The arguments are as follows:

<*sUserArg*>**:** the value you specified as the <*command_arg*> parameter in the same define_control service call in which you specified the <*script_to_call*> that triggered this script to run. If you specified no <command_arg> parameter, 0 is passed instead.

<*dlgHan*>**:** the handle (as generated by dialog.define) of the dialog that generated this call.

<*iCtrlNo*>**:** the number of the control that generated this call. Within each dialog control numbers start at 1 and are assigned in the order that controls are created with dialog.define_control. Also the LOWORD of the wParam passed to the dialog-handling procedure when the triggering event occurred.

<msg1>: If you receive a message (a WM_HSCROLL or WM_VSCROLL) because a scrollbar control has moved, the text “scroll”

If you receive a message because of *any other* notification accompanying a WM COMMAND or WM_NOTIFY, *<msg1>* contains the value of that notification , e.g. BN_CLICK. Usually the same as the HIWORD of the wParam passed to the dialog-handling procedure when the triggering event occurred.

If the message occurs because of a mouse event, *<msg1>* will be one of:

<msg1>	**in hex**	**means**
WM_RBUTTONUP	0x0205	right
WM_MBUTTONUP	0x0208	middle
WM_LBUTTONUP	0x0202	left
WM_XBUTTONUP	0x020C	mouse4
WM_XBUTTONUP + 0x10000	0x1020 C	mouse5

<msg2>: If you receive a message (a WM_HSCROLL or WM_VSCROLL) because a scrollbar control has moved, this will be the current position of the scrollbar.

If you receive a message because of *any other* notification accompanying a WM COMMAND or WM_NOTIFY, *<msg2>* contains the value of the lParam parameter sent to the message handler.

If the message occurs because of a mouse event, *<msg2>* will indicate whether any modifer keys were pressed when the mouse action occurred and will be zero or a combination of:

<msg2>	**in hex**	**means**
VK_CONTROL	0x11	ctrl
VK_SHIFT	0x10	shift
VK_MENU	0x12	alt

<targHan>: the handle of the control that triggered the message, or, if message originated from the dialog itself, it's handle. This is

redundant on some of previous parameters, but handles for controls evolved after stuff like control numbers.

page 118 of 260

11.5.2 Which Script?

You can specify a script to run for a dialog as a whole as a parameter of dialog.define, and one for each control as a parameter of dialog.define_control. Which one runs: depends.

A script runs when

- a control receives a notification or message and,
- that notification or message is set to be one that is to be noticed by the control itself (via the define_control or set_response services), by the dialog (via the define or set_response services), or by the default for the control's type (BN_CLICKED for buttons, check box controls and radio buttons); and
- a <*script_to_run*> has been set either for the control or the dialog as a whole.

If on any given notification to any given control there's either a script to run for the control or for the dialog, but not both, there's no problem: If the notification is specified as an event for either the dialog or the control or even both, then *only one* script will run. A script defined to run for the dialog as a whole will be the unambiguous fall-back script for controls that have no script of their own.

But what happens if both the control and the dialog have a defined script to run? Which fires?

notifications accompanying a WM COMMAND or WM_NOTIFY message

for:	**message(s) and script defined?**							
control	yes	yes	yes	yes	no	no	no	no
dialog	yes	yes	no	no	yes	yes	no	no
control type	yes	no	yes	no	yes	no	yes	no
which script runs: *	**c**	**c**	**c**	**c**	**d**	**d**	**c**	**n**

*__c__: control's **d**: dialog's **n**: none

Which, translated means: a script defined for a control always aces a script defined for the dialog; except if the notification is only an event at dialog level, or only at dialog level and as the default for the type of control/

page 119 of 260

11.5.3 <action_on_close>

There's an <*action_on_close*> parameter of the define service that specifies a script that's run when you close the dialog by any action at all; clicking a button that terminates; <alt>-F4;or from the control box; or by <escape>. <*action_on_close*> can be any PowerPro command, though I expect it will be the name of a script (or subscript) to call. No arguments are assumed, though, so you can do whatever command you want. You should not manipulate any controls or the dialog; it's all ready on the way to being taken apart by the time <*action_on_close*> is invoked. In particular don't invoke dialog.destroy; it's already happening.

Use <*action_on_close*> for doing stuff like terminating events, invoking PowerPro commands, etc.

An alternative to having an <*action_on_close*> is to make sure you have a control with an *<id>* of "cancel", even a hidden one. If you do and you don’t have an <*action_on_close*>, the *<script_to_call>* associated with that control will fire if you close your dialog by any means except <enter>. To cover that eventuality, which will only close the dialog if you've defined a control with an *<id>* of "ok", make sure you do your terminating code in the *<script_to_call>* associated with that control as well. See Section 11.5 “Script Calls and Their Arguments”.

11.6 Specifying Which Messages Are Responded To: the <*events*> parameter

optional in: define, define_control

Scripts are run (see previous section) when events occur.

By default:

- buttons and the button variants (check box controls, radio buttons) respond to the BN_CLICK notification accompanying a WM_COMMAND message.

- tab controls respond to the TCN_SELCHANGE notification sent with a WM_NOTIFY message (and deliver the numeric id of the tab changed (lowest legal value 0 or 1) as the <*msg2*> parameter sent to your <*script_to_run*>.

Using the <*events*> parameter of the define_control service, you can override this default for any control and specify which notifications accompanying the WM_COMMAND or WM_NOTIFY messages will cause it to trigger a call back to a PowerPro script.

There's a problem with WM_NOTIFY messages (which mostly? all? come from the common control group); the data that travels with them is not always in the same format, so it's not always possible to determine exactly which notifications trigger it, and from which control them come. My code assumes

- <*msg2*> delivers a pointer to a NMHDR structure (from which the notification can be deduced. Possible values include those beginning NM_ (for all controls), and those containing ??N_ (for various flavours of common controls, especially listviews and treeviews, neither implemented yet); see dialog-related_defines.txt for some predefined ones.

- The id of the originating control either gets delivered as the value of wParam, or as the idFrom member of the NMHDR structure delivered with <*msg2*>

If the WM_NOTIFY notification you wish to use doesn’t conform to the above requirements, it won’t work; let me know and I'll see if I can add support for the notification you want.

Using the <*events*> parameter of the define service, you can specify events which should be notified for all controls in the created dialog..

You can specify up to ten notifications for any control or dialog.

If you wish to prevent event monitoring for a control or dialog, just specify an <*events*> parameter of "none".

You specify a notification just by typing its hex or decimal value, if you know it. Or you can find it in **dialog-related_defines.txt**, and copying the local variable

page 121 of 260

assignments you need from there to your script. For instance, if you want to monitor a particular edit box for changes, copy

```
local EN_CHANGE = 0x0300
```

to your script and use it to establish monitoring of a particular edit box with:

```
dialog.define_control(hDlg, "editbox", "",   10,   10,  245,  40, "",  ;;+
                      ".myScript@ebChange", "", EN_CHANGE)
```

If you're using more than one notification code, you must make sure each is delimited by white space:

```
local EN_VSCROLL = 0x0602
local sNotifs = EN_CHANGE ++ " " ++ EN_VSCROLL

dialog.define_control(hDlg, "editbox", "",   10,   10,  245,  40, "",  ;;+
                      ".myScript@ebChange", "", sNotifs)
```

For an explanation of how events determine which of several scripts might get fires, see above. Section 11.5.2 "Which Script?".

11.7 Colours: the *<foreground>*, *<background>* parameters

optional in: define_control, set_colour

You can specify the default colour for the foreground or background with the null string, or by omitting the parameter, or by using the word "default".

Or: you can use any of these colour names (main HTML colour names, according to autoHotKey's Chris Mallett):

name	COLORREF	name	COLORREF	name	COLORREF	name	COLORREF
black	0x000000	maroon	0x000080	green	0x008000	navy	0x800000
silver	0xC0C0C0	red	0x0000FF	lime	0x00FF00	blue	0xFF0000
grey	0x808080	purple	0x800080	olive	0x008080	teal	0x808000
white	0Xffffff	fuchsia	0xFF00FF	yellow	0x00FFFF	aqua	0xFFFF00

Or you can provide a number representing a COLORREF. Easiest way to get a number is to use the dialog.rgb function.

11.8 Icons: the *<icon_path>*, *<path_to_image>*, and *<icon_number>* parameter

optional in: define (*<icon_path>*)
show (*<icon_path>*)

required in: set_image (*<path_to_image>*)

<icon_path>/*<path_to_image>* is either the path to a source of icons (dll, ico, icl, or exe file), or the name of a system icon.

If *<icon_path>*/*<path_to_image>* is meant as a relative path, first it tested to see if it's relative to the location of the current configuration file (the folder returned by pprofolder); if not, it's then tested to see if it's relative to the PowerPro installation folder. If omitted or the null string, it's treated as specifying PowerPro.icl in the same folder as PowerPro.exe. That means you can specify an icon in PowerPro.icl as either "" or "PowerPro.icl" (a path relative to the PowerPro installation folder).

If it's one of the following keywords, an icon embedded in the plugin dialog.dll will be used:

keyword	**number**	**icon selected:**
blue2Q	1	blue icon, 2 question marks
green2Q	2	green icon, 2 question marks
red2Q	3	red icon, 2 question marks
blueQ2CkBar	4	blue icon, question mark, 2 checks, bar
greenQ2CkBar	5	green icon, question mark, 2 checks, bar
redQ2CkBar	6	red icon, question mark, 2 checks, bar
blueQ2Ck	7	blue icon, question mark, 2 checks
greenQ2Ck	8	green icon, question mark, 2 checks
redQ2Ck	9	red icon, question mark, 2 checks
blue1Q	10	blue icon, 1 question mark
green1Q	11	green icon, 1 question mark
red1Q	12	red icon, 1 question mark

You can also pick these icons from the file ?"plugins\dialog.dll" (a path relative to the PowerPro installation folder) with the icon number indicated. And

```
env("SystemRoot") ++ ?"\system32\shell32.dll"
```

will get you the standard system icons.

The dialog plugin uses gdiplus if it's available (for e.g. resizing images).

If it's one of the following keywords, one of the standard system icons is used:

keyword	**icon selected:**
application	Default application icon
asterisk information	Asterisk (used in informative messages)
exclamation warning	Exclamation point (used in warning messages)
hand error	Hand-shaped icon (used in serious warning messages)
question	Question mark (used in prompting messages)
winlogo	Windows logo

<icon_number> Optional: The icon within the file *<icon_path>* to use. If omitted or the null string,, taken to be 0 if *<icon_path>* is specified; taken to be 4 (the icon for ScriptFiles) if no explicit *<icon_path>* given and PowerPro.icl is the assumed icon source.

11.9 Fonts: the *<font>* and *<font_spec>* parameters

required in: set_font
optional in: define, define_control

<font> := <font_spec> | *<font_handle>*

<font_spec> **:=** [*<stock_font>* | *<font_name>*][*<options>*][*<point_size>*]

All of *<font_spec>* is case insensitive.

If *<font_name>* contains white space, it must be enclosed in quotes.

If you provide neither *<stock_font>* or *<font_name>*, the stock font associated with DEFAULT_GUI_FONT will be used.

<options> if present are separated from *<font_name>* (if present) by whitespace and may include "K" for strikethrough, "U" for underline, "I" for italic, and any of the following to set weight:

set with	Weight	Define Symbol
T	100	FW_THIN
L-	200	FW_EXTRALIGHT FW_ULTRALIGHT
L	300	FW_LIGHT
nothing	400	FW_NORMAL FW_REGULAR
M	500	FW_MEDIUM
B-	600	FW_SEMIBOLD FW_DEMIBOLD
B	700	FW_BOLD
B+	800	FW_EXTRABOLD FW_ULTRABOLD
H	900	FW_HEAVY FW_BLACK

You can have *<options>* without a *<font_name>*, and vica versa; but if you do make sure a *<font_name>* can't be confused with *<options>*; if you have a name made entirely of the letters TUITLMB and less than six letters long, quote it.

page 126 of 260

<point_size> if present and preceded by anything is separated from what precedes by whitespace. May include a decimal fraction, eg. "10.5" is valid; the decimal separator will vary with your locale, e.g. it may be a comma in many eurolands.

<font_handle> : The only way you can get a font handle at the moment is to use a script with dll plugin calls. You can only specify a *<font_handle>* as an argument to set_font, and then only once the dialog or control has been created (since you can't get a valid font handle for a window unless it already exists).

<stock_font> may be one of the following letters or names:

name	**letter**	**specifies**	**equivalent #define**
gui	g	on a US English non-customized XP machine., maps to "MS Shell Dlg" which in turn maps to "Microsoft Sans Serif"	DEFAULT_GUI_FONT
system	s	The system font. This is a proportional font based on the Windows character set, and is used by the operating system to display window titles, menu names, and text in dialog boxes. The system font is always available. Other fonts are available only if they have been installed.	SYSTEM_FONT
fixed	f	A monospace font based on the Windows character set. A Courier font is typically used.	ANSI_FIXED_FONT
variable	v	A proportional font based on the Windows character set. MS Sans Serif is typically used.	ANSI_VAR_FONT
device	d	The preferred font for the given device. This is typically the System font for display devices; however, for some dot-matrix printers this is a font that is resident on the device. (Printing with this font is usually faster than printing with a downloaded, bitmapped font).	DEVICE_DEFAULT_FONT
oem	o	a monospace font based on an OEM character set. For IBM® computers and compatibles, the OEM font is based on the IBM PC character set.	OEM_FIXED_FONT
sysfixed	x	A monospace font compatible with the System font in Windows versions earlier than 3.0.	SYSTEM_FIXED_FONT

If anyone needs the character set option, can be added.

11.10 Control and Dialog Positions and Sizes

required in: define, define_control

Mandatory parameters of calls to define and define_control define sizes and positions of controls and dialogs.

11.10.1 *<dimensions>*

With the various pre- and post-fixes allowed in *<X>*, *<Y>*, *<width>* and *<height>* parameters, all of which require a quoted string instead of a plain integer in a parameter list, you can instead merge all four parameters into one string (the *<dimensions>* parameter), which replaces *<X>*, *<Y>*, *<width>* and *<height>* parameters in calls to define or define_control. Leave whitespace between the subfields, which are in the same order as the original parameters. Trailing subfields may be omitted and will be assumed to be 0.

11.10.2 Dialog Units

<X>, *<Y>*, *<width>* and *<height>* parameters *aren't in pixels*; they are in dialog units, which are based on the size of the font used by the system (and chosen by the user). If you have a large font selected, the dialog will be large, if you use a smaller font, the dialog will be that much smaller. This is important as it makes sure that all of the controls are the proper size to display their text in the current font.

These measurements are device independent, so an application can use a single template to create the same dialog box for all types of display devices. This ensures that a dialog box will have the same proportions and appearance on all screens despite differing resolutions and aspect ratios between screens.

One horizontal dialog unit is equal to one-fourth of the average character width for the system font.

One vertical dialog unit is equal to one-eighth of the average character height for the system font.

page 128 of 260

11.10.3 Resizing Dialogs

optional in: define, define_control

Used in sample scripts dialogPluginDemo2.powerpro, dialogPluginDemo7.powerpro, regexDialog.powerpro (via associated *dialog_definition_file*), and regexDialogScintilla.powerpro (via associated *dialog_definition_file*)

<*X*>, <*Y*>, <*width*> and <*height*> parameters in calls to both define and define_control (or their equivalent subfields in a <*dimensions*> parameter) can be given a suffixes. These determine what happens when a dialog is resized.

First: you can only resize a dialog by dragging if it has the "thickframe" style. If it has that style, you can make resizing easier if you want to by adding a status bar with the "sizegrip" style.

When a dialog is resized, by default all controls just stay where they are, unchanged. If you want to override that behaviour, use the following postfixes:

postfixes	mean	apply to	explanation
L or I	left or inner	X	anchor left; preserve distance between left edge of dialog and left edge of contro
R or O	right or outer	X	anchor right; preserve distance between right edge of dialog and right edge of control
T or I	top or inner	Y	anchor top; preserve distance between top edge of dialog and top edge of control
B or O	bottom or outer	Y	anchor bottom ; preserve distance between bottom edge of dialog and bottom edge of control
$	both	X Y	anchor both edges of the control; inconsistent with W or H or "!"
!	proportional	X Y W H	dimension grows in proportion to increasing dimension of dialog

If <*X*> and/or <*Y*> parameters in calls to define have a resizing postfix, *all* the dialog's controls will move as specified by those postfixes in response to the dialog's new dimensions. If <*width*> and/or <*height*> are postfixed, *all* controls will be resized.

Adding a resizing postfix to the <*X*>, <*Y*>, <*width*> and/or <*height*> parameters (or their equivalent subfields in a <*dimensions*> parameter) in calls to define_control that is *identical* to the equivalent postfix in the call to define which created the control turns off that postfix for that control.

Adding a resizing postfix to a parameter in calls to define_control that is *different* to the equivalent postfix in the call to define overrides that suffix for that control.

So for instance if the <*width*> parameter of a call to define is postfixed with "!", and the <*width*> parameter of a call to define_control for a control for the same dialog is also postfixed with "!", that particular control will not resize on the width dimension when the dialog is resized.

To summarise:

Does parameter has postfix?		**control changes with dialog?**
dialog	**control**	
no	no	no
no	yes	yes
yes	no	yes
yes	yes, identical	no
yes	yes, different	yes

The dialog editor remembers resizing postfixes when they occur in dialog_definition_files, and preserves them when saving.

11.10.4 Control Dimensions

optional in: define_control, set_position

<X>, *<Y>*, *<width>* and *<height>* parameters (or their equivalent subfields in a *<dimensions>* parameter) in calls to define_control (except for the first call to it for a dialog) may be relative to the previously defined control.

The same parameters of set_position may be relative to the *current* dimensions of the control or dialog.

Relative dimensions are expressed with the following prefixes::

- "+*n*", "P*n*" or "p*n*" indicates that the new dimension will be *n* units larger than the old or previous
- "-*n*", "M*n*" or "m*n*" indicates that the new dimension will be *n* units smaller than the old or previous
- "0" indicates that the new dimension will be the same as the old or previous
- "Z", on it's own, or followed by zero, that overrides the previous meaning of "0"; i.e. a zero value should be used for the dimension in question.

In addition any X or Y dimension in *any* call to define_control or set_position:

- ">*n*" or "_*n*" should be taken as a distance *n* units from the right or bottom edge of the dialog, instead of the usual top or left.

The dialog editor understands relative dimensions when reading dialog_definition_files, but doesn't preserve them when writing them, so it's best to add relative dimensions to a dialog_definition_files only when you're done using the editor on it.

Relative dimensions are illustrated in the sample script dialogPluginDemo2.powerpro.

page 131 of 260

11.11 <*right_click_command*>: Context Menus

If present, in a **define_control** or **define** service call, specifies a command to run when user right clicks (on control or space in dialog outside any control) Typically the command would be a call to a script which would cause a PowerPro menu to appear, thus simulating a context menu, e.g. using menu.show or cl.ShowMenu.

If you invoke a script, six arguments will be appended to the script call if you right click on the control or dialog: see Section 11.5.1 "The Structure of Event-Handling Scripts".

The first of those arguments is <*UserArg*>, data provided by you. There is no place for you to specify that data in a **define_control** or **define** service call, so if you want to specify some, use

set_response(<*target*> <*script_to_call*>, <*command_arg*>,"right").

If you specify a <*right_click_command*> for a scintilla, rich text edit or plain edit control, the bad news is that it will prevent the standard edit context menu ("Undo", "Copy", "Cut",…) from appearing. The good news is that you'll find a replacement menu that does much the same defined as part of the makeFormatMenu function of **regexDialog.powerpro** and **regexDialogScintilla.powerpro**.

If the value of a <*right_click_command*> is

```
cb("@rClick")
```

The following function would create a menu and show it

```
Function rClick(sUserArg, dlgHan, iCtrlNo, msg1, msg2>, targHan)
local cLst

 if (cl.Exists("dlgEditCnxt")) do
  cLst = cl.Get("dlgRclick")
else
  cLst = cl.create("dlgRclick")
endif

cLst.removeAll
local hitem = cLst.insert(0)
hitem.AddLeft(cb("@onAction1", targHan)).setLabel("Action1")
local hitem = cLst.insert(0)
hitem.AddLeft(cb("@onAction2", targHan)).setLabel("Action2")
cLst.ShowMenu("centreundermouse")
cLst.removeAll

quit
```

You would of course then need to provide

Function onAction1(targHan)

Function onAction1(targHan)

to response to user context menu choices.

page 133 of 260

12.0 dialog_ Variables

These are two variables set by various services: they will set one or the other, but not both, depending on whether returns_values, returns_status, or returns_nothing have been called, or, if none have been called, on the value **whatToReturn** in dialog.ini/plugins.ini.

dialog variable	**set by**
dialog_status or **dialog_result**	define define_control run get_value (alias *get*) send_message rgb
dialog_status	all

13.0 The Dialog Editor

It's called DialogEditor.exe. Creative, huh? It reads dialog_definition_files and displays the dialog layout they specify, which you can edit and save.

Please treat DialogEditor.exe as a beta. Make a backup of any dialog_definition_file you run through it.

The purpose of the dialog editor is to allow you to create and edit dialog_definition_files graphically, primarily so you have a WYSIWYG tool to visualise control layout. It only allows you to alter a few dialog and control properties in addition to their size and position.

13.1 Configuration

The dialog editor has an ini file, in the same folder as DialogEditor.exe, sensibly called DialogEditor.ini. If your configuration ini file (dialog.ini or plugins.ini) contains the keys **defaultFieldSeparator** or **defaultEvaluationMarker**, you must have the same keys with the same values in the [DialogEditor] section of DialogEditor.ini. If you don't do this, your dialog_definition_files won't make sense to the editor.

In the same ini file,. the key **defaultFileType** determines the default file type for File | Open, File | Save and File | Save As menu items. The key value can be of any length, but only the first character counts: i for ini (the default if the **defaultFileType** key is absent) , t for text files.

Incidentally DialogEditor.exe modifies the registry at

```
HKEY_CURRENT_USER\Software\PowerPro\Dialog Editor
```

to keep the MRU: a MFC rigidity that I can't be bothered to override.

13.2 Loading and saving dialogs

Dialogs can be loaded from or saved to text or ini files, as described above (see dialog_definition_files). (I may get around to loading and saving a dialog in the form of a powerpro script eventually).

If a file without an .ini extension is opened in the dialog editor, it will be treated as a text file; if the file has an .ini extension, that's how it will be parsed.

You can drop a file onto the editor to open the file.

Once a file is open, if it's modified externally (by e.g. a text editor) you'll be warened and asked if you want to reload.

On saving, the file format to save to is determined by extension: if you save to a file ending .ini, you get that; any other (or no) extension gives you a one-line-per-control text file.

You can open a file in one format and save it as another.

If you save an existing file, a *.bak backup will be created; any previously created *.bak file of the same name will be deleted.

The dialog editor understands relative dimensions when reading dialog_definition_files, but doesn't preserve them when writing them. And it remembers resizing postfixes when reading dialog_definition_files, and preserves them when saving.

13.3 Editing dialog properties

The editor doesn't display a titlebar for your dialog, but there is one: I just haven't worked out how to simulate it in the MFC document.

Only a few dialog parameters are editable (from the “Dialog | Properties" menu item, or the toolbar, or double-clicking on any part of the dialog area not occupied by a control)) – the primary being position, size and title. But any other parameters you've added manually to an ini or text file will be preserved when you subsequently save changes.

If a parameter contains an evaluation marker (normally “#”), it’s an expression, and the editor can’t access it’s true contents, so the relevant field(s) in the properties editing dialog will be disabled (e.g., if the <*style*> field contains an evaluation marker, all the style check box controls will be disabled.).

13.4 Editing controls

Add new controls from the "Controls" menu. Control types I haven't yet implemented in the dialog plugin are greyed out.

You can also add controls using the keyboard shortcuts indicated on the menu for the some control types, or the toolbar.

You can move a control once they're placed by dragging with the mouse, or you can nudge it around with the arrow keys.

You can resize a control, once selected, with the sizing handles.

From the "Drawing Aids" menu, you can show or hide the drawing grid and margin; whether controls snap to the grid or margin; and their sizes. If you change any of these settings, your changes will be saved to the DialogEditor.ini file and will come in to forcer next time you run the editor.

You can cut, copy, paste and delete controls from the "Edit" menu, by using the conventional accelerator keys, or via the toolbar.

You can undo actions from the menu and with <ctrl>Z.

You can select groups of controls either by rubberbanding around them with the mouse, or <ctrl>-left clicking on them. You can them move the group by dragging with the mouse.

If more than one control is selected, you can line them up in various way from the Layout menu.

if several are controls selected you can't resize them, nor can you edit their properties.

You can edit a few properties of each control (double click on the control; or context menu, "Properties"; or "Layout" menu, "Properties").

- You can change the control name (only useful if you want to refer to the control in your script).
- You can set the text of buttons, check box controls and radio buttons.
- You can add or remove styles which apply to all controls: "3d", "border", "notab", "group" and "hidden". Checking or unchecking "3d" or "border" will update the appearance of the control in the editor

But: if a parameter activated from a file contains an evaluation marker (normally "#"), it's an expression; the editor can't access it's true contents, so the relevant field in the properties editing dialog will be disabled.

If you save a dialog which was originally loaded from a file, any control parameters specified in that file that aren't visible in this editor will be preserved.

page 138 of 260

13.5 Running dialogs

A dialog can be run through the dialog plugin (from the "Dialog" menu) either

Running a dialog on it's own (keyboard shortcut: Shift-F5)**:**
The variables dialog_status and dialog_result will be declared as a *global* variables before the dialog is run.

Some simplifications have to be made:

- Any dialog or control parameters that contain an evaluation marker will be ignored.
- So will any variable names in the <*text_or_var*> parameter, since I can't know what context the dialog is mean to run in, and therefore can't determine what variables there might be and what values they might have.
- Any *<script_to_call>* you've defined will be ignored.

You can close the displayed dialog either:

- by choosing the same Dialog menu item (which changes text to "Terminate Dialog"), or:
- by hitting Shift-F5 while the editor has focus, or;
- by simply closing the displayeddialog via Ctrl-F4 or the X- caption bar icon.

Running a dialog through a script (keyboard shortcut: F5):
With dialog_definition_files of some complexity, you will almost certainly have included expressions with evaluation markers and references to *<scripts_to_call>* which are dependent on the dialog running from a particular script. If you chose this option, a dialog pops up and asks you to pick a script to use. If you've run the same dialog_definition_file through a script before, the editor will remember (via the dialogEditor.ini file) and propose the same script again.

If you have PowerPro 4.5.02 or later, the complete path to the script you've chosen will be passed on to PowerPro, using Bruce's new call function. If you're using an earlier version, only the script's name will be passed (using the ".scriptname" syntax), so it better be on your scriptpath. So, if you want to try opening dialogPluginDemo.txt or dialogPluginDemo.ini in the editor, and then running them through dialogPluginDemoFromConfigFile.powerpro, the latter must be on your scriptpath (as must all it's associated image and icon files) if you use a version of PowerPro earlier than 4.5.02.

The path to the dialog definition file will be passed as the first argument to the chosen script, so the script must be coded to deal with arg(1) as that path..

13.6 Editing dialog definition files as text

You can edit dialog and control properties directly by invoking the Edit Script menu option ("Dialog menu"; keyboard shortcut: Control-F5); this will freeze the dialog editor, bring the dialog definition file you're working on up in your text editor, and only return to the dialog editor when you close the dialog_definition_file in your text editor.

To make this work you have to provide a powerpro script to manage the transfer between dialog and text editors. Three samples are provided: dialogEditorToNotepad.powerpro, dialogEditorToWordpad.powerpro and dialogEditorToUltraEdit.powerpro (which probably works, with minor mods, with Crimson Editor – see comments within).

You must name the script which you use via the **textEditorScript** key in dialogEditor.ini; if you have PowerPro 4.5.02 or later the script can be anywhere and **textEditorScript** can specify an absolute path; if you don't give an absolute path, the script must be on your scriptpath. You can include or omit the .powerpro extension.

The script must be able to determine if a particular file is open in your text editor, and whether, when it's closed in that editor, it's been modified. It must call a dialog plugin service (editor_support) whose only purpose is to interact with the dialog editor. Hopefully the dialogEditor*.powerpro scripts are well-enough commented so you can roll your own from them. Notepad is dead easy; ultraEdit and other more sophisticated MDI interfaces are somewhat more difficult. Wordpad requires accessing the registry to get wordpad.exe's location, and massaging the result.

14.0 General Restrictions

Don't use any owner-drawn styles in a control.

15.0 Possible Enhancements

Tooltips don’t work in XP if the powerpro manifest file (powerpro.exe.manifest) isn’t present. Will try and fix.

Quite a lot of stuff probably missing for listviews; I'll add more in subsequent versions. In particular there's no get_value implemented yet; no named messages; tooltips probably need work.

Colours currently don't work for combo box controls or the thumb on sliders.. I can work around this by making them owner-drawn and may do.

Tooltips don’t work for treeViews; I’ll look into supporting them.

<ctrl><tab> doesn’t navigate properly in tab controls; I’ll try to fix.

I'll probably add an export service to dialog plugin, so you can create dialog definition files from scripts.

The resizing algorithm is inaccurate. I'll improve it.

16.0 Change History

version 1.19:

- Changed parameters to choose_font allowing a *<font_spec>* instead of a <*target*>, and variable name to be specified to set and return colour. choose_font now used in controlFontChanger.powerpro.

- set_font applied to <*window_handle*> now returns the HFONT associated with the window after set_font runs.

- Added a variant to clear service allowing deletion of font resource referred to by an HFONT

- Fixed an error in set_font which caused PowerPro to crash if applied to a window with the system font. Updated documentation for set_font to describe more accurately when it works and when it doesn't when applied to a <*window_handle*>.

- Fixed a bug in choose_font which caused to initialise the font picker dialog with garbage when <*target*> is a <*window_handle*>.

- Added to documentation for get_value(<*window_handle*>, "font") variant; usually doesn't work on anything but controls

- Added variant of get_value with <*property*> "fonth" or "fonthandle"

- set_response now has <*mouse_event*>s "enter" and "exit" (to specify a response to make when mouse enters a control's window, or leaves it).

- For statics, if you invoke set_response to set a response to a <*mouse_event*>s; or set_colour with a <*mouse_state*>*;* The "notify" (SS_NOTIFY) style will automatically be added to the control's styles. Example of static with both a set_colour and a set_response to make a static with a clickable url can be found in dialogPluginDemo7.powerpro.

- Added pdf version of documentation

- Fixed memory leaks in GDI and other resources (thanks to Sheri Pierce for locating them). There are more leaks, will try to plug them in the next version.

- Tooltips don't work in XP if the powerpro manifest file (powerpro.exe.manifest) isn't present. Will try and fix.

version 1.18:

- changed the first <*type*> parameter of browse_for_file service to <*type_and_options*>, and added numerous flag keywords. <*type_and_options*> *i*s now required, not optional.

- forgot to document trailing parameter to browse_for_file service call <defaultFileName>.

- added trailing parameter to browse_for_file service call *<varNameROstate>.*

- added code to return multiple file selections from browse_for_file service

demo file browse_for_file.powerpro.

version 1.17:

- fixed error in show service: when applied to a control, didn't return the control's handle (it returned the dialog's handle instead).

- clarified behaviour when you call a service that returns a *<handle_to_control>* with multiple control ids; service will return the handle of the first control id encountered.

version 1.16:

- fixed inconsistency in handling clicking on the X-icon in the dialog caption bar, or on the "close" option in the control menu to close a dialog. Now works as defined in docs.

version 1.15:

- Fixed error in docs and behaviour re control tooltip widths; you either prefix the *<tooltip_style>* with a width, or include width:nn.

- left and right arrow keys will now navigate between tabs of a tab control which has focus. ctrl-tab should rotate through tabs but doesn't yet

version 1.14:

- Updated docs re control tooltips: problem with no-shows when manifest in operation apparently gone

- Updated docs re set_tooltip for control tooltips: weird problem with shadow appearing appears if you call set_tooltip after create when manifest in operation. Workaround: call set_tooltip before create, or don't use the manifest.

- Added a variant of get_value (alias *get*) that will return the text in part of a statusbar. Also get_value now works with external statusbar controls (those on other windows besides those generated by this plugin).

- Added to set_value for statusbars allowing multiple parts to be set in one call.

version 1.13:

- Hopefully fixed bugs causing tooltips to disappear when a control was hovered over for a short period, or when a control was clicked.

- Fixed bug in set_value for listview with keyword "row".

- Tightened rules for set_value for listview; you have to set the row ("item") label before you can set values for "subitems" (columns)

- Fixed bug in set_value for treeview which meant insertion of a node under another wasn;t working.

- Added new script **notificationDialog.powerpro** demonstrating use of a dummy dialog to pop up a balloon notification tooltip from the taskbar.

- Control tooltips now default to opening after your system double-click time, then staying open forever as long as you hover over the control

- Added tooltip pseudo-styles "slashisnl", "showafter" and "stayopen" Added to writeup for the <*delay*> parameter of set_tooltip for controls to take the latter, and a problem in XP SP2, into account.

- Statics can now have tooltips (via set_tooltip for controls or define_control)

version 1.12:

- Another bugfix: define_set failed. A lot.

version 1.11:

- Oh dear. This version is just a bugfix. On XP dialog plugin version 1.10 tried to cope with Visual Styles and failed miserably, and managed to screw up your system-wide Visual Styles at the same time.

version 1.10:

- Tightened up rules for value of the <*show_type*> parameter of the show service, so that controls only accept "show", "hide" and their synonyms, not the other dialog-oriented possibilities.

- Again for the show service, added the <*show_type*>s "traymin", "trayicon" (and a bunch of arguments to follow) for dialogs, so dialogs can now display an icon in the system tray (not necessarily the one assigned to the dialog).

  Calling dlgHan.show("traymin") will hide the dialog as well as creating the tray icon.

  dlgHan.show("traydel") will kill the tray icon.

You can assign an action to a tray icon using the <*script_to_call*> argument of the show service; you can use cl services to make that action showing a menu.

- You can set the colours of a control's tooltip with set_tooltip and set_colour…
- …but Sheri Pierce notices that there's a problem using tooltips and XP visual styles
- You can use set_tooltip to set a tooltip on a tray icon created with the show service, and to display a balloon-style, notification tip from the tray icon on request.
- set_tooltip, if used to set a balloon tooltip's properties, can have a further parameters, <*title*> and <*icon*>.
- set_tooltip <*delay*> parameter can be used to set a tooltip's delays.
- You can use set_response to set or change actions associated with mouse clicks on the tray icon.
- For dialogs you can specify <*sys_command*>s for set_response, allowing you to respond to stuff like user closing, moving, resizing, minimising and maximize the dialog.
- the *<name>* parameter of define_control is subject to tighter rules; must be be of 2-63 characters in length, must begin with an alphabetic character, and must not be the same (case insensitive) as any of the *<show_type>*s allowed for the show command applied to a dialog
- There can now be only one control with <*id*> (set as a parameter of **define_control**) of "escapable" or "cancel"; one precludes the other. You can also get the same effect as "escapable" by creating a button with <*id*> of "cancel" and with the "nodestroy" style.
- I thought tooltip text could only have explicit line breaks if you specified <*width*> as part of *<tooltip_style>*. Turns out, not so. And since "\n" works fine as a line break in a tooltip, I've removed the rule that line breaks are marked by "/". That means if you want a tooltip with line breaks, you can't define it in a *<dialog_definition_file>*; you'll have to use set_tooltip instead.
- Clarified discussion of *<action_on_close>*.
- Added "prevctrl" property to get_value applied to dialogs, and "owner" applied to controls and dialogs created modal and with an <*owner*>.
- Updated dialogPluginFunctions.txt to match above changes
- Improved specificity of error messages to do with bad parameters.

- Finally got around to fleshing out handling script (“onLV”) for the listView in dialogPluginDemo5.powerpro; it now reports any time a new row is selected in the control.

- Fixed bug in get_value for listviews with keyword "selectedindex"

version 1.08:

- added discussion Multiple Simultaneous Dialogs From The Same Script and Enforcing a Single Instance of a Dialog from a Script

- added a new value for the <*id*> parameter of define_control, “escapable”, which will trigger when user hits <*escape*> but not <*alt-F4*> or any of the other ways of closing a dialog, and won’t presume user has destroyed the dialog; also clarified description of <*id*>. See dialogPluginDemo1.powerpro for example of usage.

- You can now specify <*script_to_call*> and <*command_arg*> arguments for a tooltip.

- dialogPluginDemo8.powerpro illustrates the dialog style "draggable", the button image-related style "imgfill" and use of a <*script_to_call*> for a tooltip.

version 1.07:

- removed aliases **set_icons, icons, add_icon, add_icons** for set_icon (which only applies to the dialog, not to controls, so there can only ever be one)

- finally fixed (I think) peculiar concentric rectangles appearing when tootips specified for controls

- Previously, set_tooltip only worked if a tooltip had been specified when a control was defined. Now that's no longer required: you can create a tooltip on the fly. Also, set_tooltip takes optional <*tooltip_style*> and <*font*> arguments. And you can use the same service to kill off a control's tooltip.

- Added dialogPluginFunctions.txt to be used as a file menu, perhaps merged with pprofunctions.txt.

- There were errors in dialogPluginDemo.ini and dialogPluginDemo.txt, so the sample script dialogPluginDemoFromConfigFile.powerpro didn’t work right

version 1.05:

- Added dialog styles "toolwin", "tool" and "topmost"

- Added "dialog style" "draggable"

- Added a button image-related style "imgfill"
- Added the “reset” and "find" messages for combo boxes

version 1.03:

- I left out the dialogPluginDemo.ico file in the last distro. Sorry.
- dialog.set_colour now works for scintilla controls, including keyword arguments allowing controls to have different colours depending on whether they’re pressed, have focus, or have the mouse over them.. Try the edit boxes in dialogPluginDemo2.powerpro and the edit boxes in regexDialogScintilla.powerpro.
- Added additional valid keywords for the <*mouse_event*> parameter of set_response allowing discrimination between left and right modifiers.

version 1.01:

- All sample scripts now use cbx() for callbacks instead of cb(), and therefore require at least PowerPro 4.8 RC3.
- Minor fixes to **regexDialog.powerpro** and **regex\regexDialogScintilla.powerpro** scripts, including inability to see selected text highlighted in the latter.
- Added keyword arguments to set_colour service, allowing controls to have different colours depending on whether they’re pressed, have focus, or have the mouse over them.
- Changed the names of supporting files (.powerpro, .txt, .ini, .ico) from **dialogPluginTest*** to **dialogPluginDemo***
- The colour parameters in define_control for a month-calendar and date-time controls now work.
- In both dialog.define_control and dialog.set_colour, the <*foreground*> for date-time, month-calendar, treeview and listview controls can be either a number (possible generated by dialog.rgb) or a colour name, in which case it will be interpreted as a foreground-ish colour; or it will be a keyword specifying the colour aspect of the control you wish to set.
- dialog.set_colour now works for status controls (still background colour only though, and still ugly).
- dialog.set_colour can now be invoked before a dialog is created or run, though I'm not sure why you'd want to

.version 0.97:

- Fixed bug meaning some get_value <*type*>s, in particular "rtf" for rich edit controls, didn't work

- Fixed "wrap/don't wrap" bit of context menu for **regexDialog.powerpro**

- In this documentation, changed get_value <*type*> to get_value <*property*>; same for set_value.

version 0.96:

- Fixed a bug that meant last argument sent to functions handling mouse actions was wrong (it was always the handle to the target dialog, not the handle to target control, if any)

- Fixed a bug that meant static variables preceded by the evaluation marker wouldn't be recognised.

- The "imgspacing" style for buttons can be negative; that will overlap graphic and text, which may work if the graphic has a substantial transparent background.

- Added internal fixes in way the window receiving a mouse action is identified; should be faster and more reliable, but I bet change won't be observable.

- get_last_clicked now returns the handle to any the last control or dialog which responded to a user action, not just funny mouse actions like right-click.

- scintilla, rich text edit and plain edit controls can now be assigned a <*right_click_command*> with define, define_control or set_response, overriding the built-in context menus (or, in the case of rich text edit controls, the default context menu the plugin supplies).

- I forgot to document the get_value <*property*>s "selectlen", and "selected" or "select" for scintilla, rich text edit and plain edit controls.

- Added the get_value <*property*> "type" for all controls

- Changed name of button style equivalent to BS_FLAT from "buttonflat" to "flat".

- I'm working on interpreting "flat" button style to mean the fancy kind of flat button that gets a border when you hover over it. Work in progress, not yet done – and I may add a new style ("cool" ?) that's different from plain old flat buttons.

version 0.94:

- fixed a bug that mean same dialog run twice wouldn't cancel. Sorry about that.

- dialog editor supports animation controls.
- Added "imgspacing" style for buttons

version 0.92:

- define <*height*> and <*width*> may be prefixed by "+"; in which case dialog dimension will be large enough to contain all controls (given their dimensions) plus the quantity after the "+"
- The <*max_controls*> of the define service and its equivalent in a dialog_definition_file is now irrelevant; there is no intrinsic limit on the number of controls in a dialog. I'll continue to parse for the parameter, but will ignore it, and it's been removed from the documentation.
- define_control now returns a handle to a control, not a control id.
- Added define_set service to allow sets of controls to be used as if they were themselves controls (for some services only).
- <*name*>s in calls to define_control must be less than 63 characters (used to be 50); you get an error it they're longer.
- *<ctrl_ids>* can now, for some services (ones that change controls, but don't fetch their properties), be made up of multiple, white-space separated control names or numbers.
- Added "select" (or anything beginning "select") as a <*property*> parameter for get_value for scintilla, rich text edit and plain edit controls
- Added functionality to set_response to allow specifying actions when mouse events (other than simple left-clicks) occur on controls or the dialog.
- The scripts that responde to such mouse events (or the one specified in the parameter (*<right_click_command>)* now take the same parameters, more or less, as any event-response script.
- For set_response, if using the <*dialog_handle*>*,* <*ctrl_id*> option*,* <*ctrl_id*> may be absent, indicating you wish to set a response for the dialog itself.
- Added service get_last_clicked
- Fixed get_value("font") directed at dialog as a whole with no font defined by user.
- get_value takes new <*property*>s "hwnd", "id" and "name"; the latter two only valid when directed at a control, the first valid for a control or the dialog as a whole. Using a <*property*> of "hwnd" is the same as calling

get_hwnd; using a <*property*> of "id" is the same as calling get_control_no.

- …therefore removed services get_hwnd and get_control_no.

- The ***<dialog_handle>*[**<*ctrl_id*>, <*property*>**]** syntax works nicely in place of get_value (in previous versions I said it didn't); so does ***control_handle>*[**<*property*>**]**.

- Added new dimension prefixes to allow delta as well as proportional resizing of controls when a dialog is resized. See Section 11.10.3, "Resizing dialogs", and the illustration in dialogPluginDemo7.powerpro.

- the X or Y arguments in *any* call to define_control or set_position can be prefixed by ">" or "_", meaning they should be taken as a distance from the right or bottom edge of the dialog, instead of the usual top or left.

- buttons can now have images (set with set_image)

- buttons with colours or images now display and behave properly when disabled, and when they have "left" or "right" styles.

- For all types of controls to which it applies, if you invoke set_image with no arguments it will remove all images from the control in question.

- Added animation controls that can play avi files (and not do anything else).

- Fixed some details of named messages for sliders, e.g. TBM_CLEARSEL and TBM_SETTIC

- The <*icon_number*> argument of the **set_image service** now is either 0 or 1 based depending on the **indexBase** key in the configuration ini file (dialog.ini or plugins.ini) or on the last call to set_base..

- The decimal separator in the <*point_size*> element of a *<font_spec>* will vary with your locale, e.g. it may be a comma in many eurolands (see Section 11.9 "Fonts: the <*font*> and <*font_spec*> parameters")

version 0.90:

- Fixed an error in get_value that, if applied to non-plugin controls, causing them to disappear.

- Added "all" and item index, and other options for get_value service applied to listboxes and combo boxes.

- Added a context menu to rich text edit controls.

- Added yet another parameter (*<right_click_command>*) / ini file key (commandMouseAction) to define_control and define services. You can

page 150 of 260

now specify a script to run you right click on a control or dialog (which would normally simulate a context menu using e.g. cl.ShowMenu). Try the **dialogPluginDemo1.powerpro**, "Text To Debug Win" button.

- Fixed minor problems in regex.powerpro sample script. I'm working on moving the menu triggered by the Options buttons into context menus for edit boxes in the regex sample scripts.

version 0.88:

- Probably fixed problem with dialogs containing activeX controls crashing PowerPro on exit.
- …and another, where dlgHan.get_hwnd() crashed
- For listViews, get_value**(**<*target*>*,* "selectedAllIndex"**)** is now **get_value(**<*target*>*,* "selectedIndex" [, <*howMany*>]**)** with <*howMany*> - 1
- Corrected documentation for listViews set_value**(**<*target*>, "item",
- Tightened up rules on effects of the configuration ini file key **indexBase,** and of the set_base service; documentation notes effects wherever relevant
- Added get_base service. All scripts that require set_base(1) on entry now revert to current base if required on exit.

version 0.86:

- Fixed error in get_value with "font" tag.
- Added a new key to the configuration ini file (dialog.ini or plugins.ini) **indexBase,** and set_base service, both of which effect various services that specify elements with an index (clear, and variants of get_value and set_value for listviews, combo boxs and list boxs, and for the latter, tab controls. All scripts updated by calling set_base(1) for safety.
- Scripts that use any of those controls now set_base to 1, and restore it if was previously 0.

version 0.84:

- Fixed bugs in which services prevents from running when using *<handle_to_control>*s with minimum number of arguments.

version 0.83:

- Dialog title now not forced into lower case.
- *<handle_to_control>*s no longer have memory associated with them, so they never need to be destroyed or localcopy'd.

- Fixed a bug in handling of chained service calls that sometimes caused PowerPro crashes.

- set_position rules tightened up; if <*target*> absent or zero, implying you're resizing the dialog, all dimension arguments must be present. And fixed bug in relative dimension processing in set_position.

version 0.81:

- Fixed set_position; now works…and if you're using separate arguments for each dimension, all but the first (X) are optional. Should work before and after create (that'w what docs always said, but you don't *really* believe what I say, do you??).

- Fixed set_colour; now works for buttons; still not working for statusbars.

- Fixed buttons with both font and colour; now does the font, instead of (sob) ignoring it.

- Added "Z" relative dimension modifier, making a dimension zero ("0" is taken as "just like previously defined control's dimension").

version 0.79:

- Some services that take a dialog handle and a control id as first arguments can take instead a window handle to a control, and that control can live in some dialog or window not created by the dialog plugin. For instance get_value, set_value, set_font, enable, set_position, and show now allow that possibility. New test script dialogPluginDemoNonNative.powerpro added to demo some of the possibilities.

- Added lots of variants to get_value and set_value applied to listview controls. You can among other things set and get listview's current selection and focus, and get the whole or part of a listview returned in a vector. Many of those variants work for listviews in other dialogs, which meams they cover much the same territory as autotIt's ControlListView or autoHotkey's ControlGet.

- added alias release for destroy

version 0.77:

- Added a new section: How to Use the Dialog Plugin to Manipulate Non-Plugin Dialogs and Their Controls.

- Changed the signature used by <*handle_to_control*>s; previous one conflicted with one used by the COM plugin.

- Fixed (at least partially) resizing algorithm; at least controls move around more or less correctly when you drag a dialog bigger and smaller. Illustrated in **dialogPluginDemo2.powerpro** and the **regex** dialog scripts (text files slightly modified, to take advantage of resizing).

version 0.74:

- Syntactic sugar, catching up with Bruce's cl services; you can now chain services which operate on controls and don't return values useful: e.g. those services named set_* and some others.

- The regex samples used to have a separate file for the viewer window; now there's just one powerpro script for each of the two regex sample varants.

- enable and focus can now be applied to a dialog as whole, as well as to a particular control.

- enable, focus, show, set_font, set_position, get_hwnd, set_value and get_value can now omit the *<ctrl_id>* argument, in which case the service applies to the dialog as a whole (which previously required instead a *<ctrl_id>* value of zero).

- Fixed bug in parsing ini *<dialog_definition_file>* file files. In particular, the bug ensured that any ini file generated by the dialog editor would crash Powerpro. Duh.

- Added get_value service for list view controls.

- Added test script **dialogPluginDemo6.powerpro** that demonstrates use of get_value for list view controls, and presents athe results of a database query in a list view.

- Added *<state>* parameter for set_value service of list view controls

- Improved regex dialogs

- SciLexer.dll is required for scintilla controls. In previous version it had to be in the PowerPro folder or on the path; now it can also be in the plugins folder.

- scintilla controls now accept all named styles and messages appropriate for rich edit controls. There's a "but" for messages; see scintilla section.

- I've begun adding space in named message tables for a brief description of each message's purpose. For the moment most of that new, last columns of each table is empty; I'll be filling them in eventually.

- *<dialog_handle>*s stored in locals were never meant to be automatically deleted when the local when out of scope. They were. Now they aren't

- The distribution zip file now has a folder structure; make sure you preserve that structure when unzipping it . (It only really matters for test scripts.)
- Fixed really obscure bug involving <*handle_to_control*>s returned by calls to dialog.make_ctrl_handle in one dialog disappearing when a second dialog was destroyed.
- Added browse_for_file service to bring up file open and file save common dialogs.
- If you use set_font specifying only some properties of a font (font size, for instance, or weight), plugin will endeavour to preserve some existing font properties.
- set_font accepts any <*window_handle*> as a first argument, so you can alter the font used by any (??) window or control.
- set_font now works with richedit and scintilla controls, with some restrictins on the latter.
- The choose_font service now takes parameters allowing you to specify a dialog, control or window whoe font should be used to initialise the font choice common dialog.
- get_value takes a <*property*> of "font"; that works with a handle to a window as well as a window to a control.
- Rich edit controls only send out notifications if a mask is set to tell it so. I've added code to automatically take care of that for a subset of possible notification codes.
- added "setcursor" named messages for combo boxes and list boxes
- Added controlFontChangerTest.powerpro and controlFontChanger.powerpro to illustrate manipulation of properties of controls in dialogs not generated by the dialog plugin.
- Finished off descriptions for tab control styles
- I've added a note in the description of each type on control indicating which sample script(s) it's used in.

version 0.72:

- In last version I said I got "setcharformat" and the like to work for rich edit controls. I lied. Sometimes it works, sometimes it doesn't. Seems to depend on what else is open. scintilla controls seem more reliable. Meanwhile I've added a switch (st_bHighlightingWorks) in that alters behaviour of **regexDialog.powerpro**, depending on what works for you.

- added regexDialogScintilla.powerpro and regexDialogScintilla.txt to demonstrate use of scintilla control.

- The <*script_to_call*> argument in define_control and the *<action_on_close>* parameter of the define service can refer to command lists withon your pcf. See Section 11.5 “Script Calls and Their Arguments”.

- Added choose_font service, which brings up the font common dialog. Its output can be used in the <*font_spec*> parameter of set_font, define or define_control.

- Added support for list view, rich edit and scintilla controls in the dialog editor.

- Added lots of named messages for list view controls

version 0.70:

- set_value for both rich edit controls and plain edit controls now checks to see if the maximum length of text for the control is exceeded, and if possible increased that limit. As a result the named messages "limittext" and "exlimittext" have been removed.

- added <*flags*> and <*codepage*> argument to set_value for rich edit controls. As a result the named message "settextex" has been removed.

- removed "usectf" style for rich edit controls

- got "setcharformat" and the like to work for rich edit controls .

- send_message now assumes the <*lParam*> parameter is of integer type if it isn’t told otherwise (i.e. by the <*lParam_type*> argument or the information carried by a named message). It also has a new, optional parameter to force a control to be redrawn after the message has been sent

- added basic support for scintilla controls

version 0.67:

- Added support for listview controls, illustrated in sample script dialogPluginDemo5.powerpro. listview documentation and the get_value method are incomplete, and I haven't defined any named messages yet.
- Added support for rich edit controls; but I haven't managed to get text or paragraph formatting to work yet; it I can't, I may switch to the scite
- Added named messages applicable to both standard and rich edit controls.
- added aliases add_icon and add_icons to service set_icon
- added the clear service (alias *remove*), which can be used to removed tabs from tab controls; nodes from tree view controls; rows or columns from list views; lines from combobox or list controls.
- …which means it was sensible to remove the forms of **set_value**
  dialog.set_value(<*dialog_handle*>*,* <*tab_ctrl_id*>, "delete", <*tab_no*>**)**
  dialog.set_value(<*dialog_handle*>*,* <*tab_ctrl_id*>, "deleteall"**)**
  dialog.set_value(<*dialog_handle*>*,* <*treeview_ctrl_id*>, <*node_id*>, "delete"**)**
  dialog.set_value(<*dialog_handle*>*,* <*treeview_ctrl_id*>, "deleteall"**)**
- fixed error message when you try to use a dialog_definition_ini file and the plugin can't find the ini plugin.
- if an <*lParam*> argument of a named message used in send_message is "string", you can now specify the name of a variable, whose allocatedmemory can then be used as a buffer to receive data from the target of the sent message.
- The <*owner*> parameter of the create and run services (used to create a modal dialog) can now be a handle to any window, as well as being a dialog_handle.
- When sending a message to a control, the <*wParam*>parameter can by a handle to a struct or array (as returned by the create_struct or create_array services of the dll plugin)
- Added **dialogViewer.powerpro,** a simple dialog that can be used to display text in an edit box. See 9.2.6.

version 0.61:

- Added aliases for some services
- Added dlgHandle[<*ctrl_id*>] as synonym for get_value and set_value

The above changes are illustrated in the sample script dialogPluginDemo1.powerpro.

- Added support for tab controls, including numerous variants set_value to configure them, and set_image to allow adding multiple images to the control's image list.. Illustrated in sample script dialogPluginDemo4.powerpro.
- dialog editor supports tab controls.
- Improved visuals for various controls in dialog editor, so you can now tell them apart more easily.
- Now handles (some) notifications that come in with WM_NOTIFY as well as with WM_COMMAND.
- Images loaded into a static control with set_image will now be automatically resized to fit the control.
- Added skeletonDialog.powerpro, skeletonDialog.txt, skeletonDialog.ini, to give you a very simple, two-button dialog to start from.
- Icon numbers for icons embedded in dialog.dll now 1-based instead of 0-based

version 0.57:

- Fixed bug that meant running same dialog repeatedly intermittently crashed PowerpPro
- You can now control what happens to controls when a dialog is resized.
- A control's position and size can be specified relative to a previously defined control
- added set_position service
- You can merge <*X*>, <*Y*>, <*width*> and <*height*> parameters in calls to define, define_control and set_position services into a single *<dimensions>* parameter.

The above changes are illustrated in the sample script dialogPluginDemo2.powerpro; in addition single *<dimensions>* parameters are illustrated in dialogPluginDemo.ini and dialogPluginDemo.txt *<dialog_definition_file>*s.

page 157 of 260

version 0.55:

- send_message parameter <*lParam_is_string*> renamed <*lParam_type*>
- send_message <*lParam*> can be a handle to a composite (array or struct) obtained from the dll plugin.
- …which means that I've been able to define named messages for date pickers and month-calendar controls that make use of such composites; so for instance you can now more easily restrict the date range of such controls, and deal with a set of selected dates in month-calendar controls
- get_value works for a month-calendar control with the "multiselect" style; it returns two dates, separated by a space.
- added support for statusbars. They can have tooltips, size grips, and multiple parts.
- set_colour now works for date picker and month-calendar controls

All the above changes are illustrated in the sample script dialogPluginDemo2.powerpro

version 0.53:

- Fixed bug in get_value for check box controls and radio buttons
- The get_value service has optional <*property*> parameter; for any control with a tooltip, <*property*> beginning "tool" causes service to return tooltip text. When used for check box controls, radio buttons or three-states, <*property*> beginning "text" gets control text instead of it's state.
- The set_value service now by default returns the state of a check box, radio button or three-states. The service's optional third parameter may be <*property*> for such controls; if it begins "text", service set's the control's text instead of it's state. Illustrated in dialogPluginDemo1.powerpro

version 0.51:

- added *<tooltip>* and *<tooltip_style>* parameters to define_control service and to control-defining entries in *<dialog_definition_file>*s. Example tooltips are displayed on top three buttons displayed by dialogPluginDemo1.powerpro.
- added set_tooltip service, which allows on-the-fly changes to existing tooltips
- added a *<tooltip>* parameter to set_font service, allowing fonts of tooltips to be changed.
- Fixed a problem where running same dialog repeatedly sometimes crashed PowerPro.
- When specifying a font: stock fonts can now be modified with point size and type attributes like bold and italic. You don't have to provide any font name, just point size and/or type attributes. See Section 11.9 "Fonts: the *<font>* and *<font_spec>* parameters".
- Fixed bug in dialog editor that caused style codes to be repeated in Style property output to ini or text file.

version 0.49:

- Fixed errors meaning some controls did not respond to services they were supposed to (e.g. statics didn't accept **set_image**)

version 0.47:

- Updated the dialog editor so it knows what to do with activeX controls
- Fixed errors in the dialog editor control property dialog
- **dialogPluginDemo3.powerpro** modified so that it may work with other versions of Office. It needs the reg plugin to try to do that.

version 0.45:

- added support for activeX controls: you can now insert a control which contains an activeX object, and then manipulate the control using the com plugin.

- added set_font service: you can set font of the dialog, or of a specific control.

- added the <*font*> parameter to dialog.define and dialog.define_control services
  You will have to modify calls to the latter if you used the <*max_controls*> parameter.

All the above illustrated in dialogPluginDemo3.powerpro.

- dialog_definition_files now include a font parameter for dialogs and controls.

- I think I fixed a bug that meant I sometimes got crashes after dialog.destroy, and which I prevented by putting waits before and after dialog.destroy calls. Hopefully those waits no longer needed.

version 0.43:

- Fixed <*script_to_call*> to work with new version of cb() introduced in PowerPro 4.5.04, and to generally use call(), which allows full paths to scripts.

version 0.41:

- There's now a dialog editor, which is mostly an aid in positioning controls. It will read and write dialog_definition_files.

- dialog_definition_files can be ini files or (as previously) one-line-per control text files. Both formats are supported by the dialog editor.

- set_value can now add and insert items into list controls and combo box controls; when used for that, returns the zero-based index of the last item added or inserted. Usage is illustrated in test script dialogPluginDemo1.powerpro (at label @populateBox).

- Fixed all scripts to use cb() function as modified in 4.4.13.

- The variable you can tie a control to can now be static in the script that created the dialog, as well as a global: illustrated in dialogPluginDemo1.powerpro. Usage is illustrated in test script dialogPluginDemo1.powerpro.

- Added 12 icons (provided by Alan Martin) into the plugin dll which you can use as the argument in the dialog define service. Where the

<icon_path> parameter is expected. Usage is illustrated in test script dialogPluginDemo2.powerpro.

version 0.39:

- Fixed handling of closing dialog by alt<F4> or X-icon on caption bar; now those actions cause *<script_to_call>* associated with the control with *<id>* of "cancel" to run.
- Fixed scripts to use cb() function introduced in PowerPro 4.4.10.
- You can tie a control to a global variable by naming it in the define_control that creates the control
- Clarified the documentation on group controls and what they contain
- Changed the way get_value works with radio buttons, check box controls and three-states.

version 0.37:

- removed the "default" style from button controls. Use define_control *<id>* parameter of "ok" instead.
- added to docs: set_image supports .wmf, .emf file types.
- set_colour can be used to set the dialog background colour
- you can set button foreground and background colours with set_colour or the define_control *<foreground>* and *<background>* parameters. dialogPluginDemo1.powerpro has an example.

version 0.35:

- added support for spinner, progress, slider, month-calendar and date-time controls
- standard system icons now usable in define *<icon_path>* and set_image *<path_to_image>* parameters.
- added the get_hwnd service to get the window handle associated with a control
- merged **get_scroll_pos** and get_text services into get_value
- changed the name of **set_scroll** service to set_range, as it applies to several spinner, progress, slider and scrollbars now
- changed the name of **set_text** service to set_value, as it applies to spinner, progress, slider and scrollbars now
- changed name of dialogPluginDemo.powerpro to dialogPluginDemo1.powerpro. dialogPluginDemo2.powerpro added with examples of new features in this version.

version 0.32:

- Now observes handle.service syntax if you want to use it.

version 0.30:

- First version..

page 163 of 260

Appendix I: The Windows Dialog API

To read up on dialog boxes generally:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/windowing/dialogboxes.asp

Here's a starting point on controls:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/shellcc/platform/commctls/wincontrols.asp

page 164 of 260

Appendix II: Details on Specific Controls

II.1: Button Controls

<*control_type*> **value:** button

Used in *all* sample scripts.

II.1.1 Styles

In addition to styles applying to all controls, there are styles that apply everything that Microsoft has decided are of the BUTTON class, which includes buttons, groups, check box controls and radio buttons). They follow. No other styles apply to yer actual *button* buttons.

There are a few other button-related styles, but probably none of interest. Avoid the BS_OWNERDRAW style, it needs more support. Also no support yet for BS_USERBUTTON, BS_ICON or BS_BITMAP.

General Button Styles (including Check Box Controls, Radio Buttons, Groups)			
this style name	**or this letter**	**#define'd symbol**	**means**
lefttext		BS_LEFTTEXT	When combined with a radio-button or check-box style, the text appears on the left side of the radio button or check box
left	l	BS_LEFT	Left-aligns the text in the button rectangle on the right side of the check box
right	r	BS_RIGHT	Right-aligns text in the button rectangle on the right side of the check box
centre	c	BS_CENTER	Centres the text horizontally in the button rectangle
top	t	BS_TOP	Places text at the top of the button rectangle
bottom	b	BS_BOTTOM	Places the text at the bottom of the button rectangle
vcentre	v	BS_VCENTER	Vertically centres text in the button rectangle
pushlike	p	BS_PUSHLIKE	Makes check box controls, three-state check box controls and radio buttons look and act like a push button. The button looks raised when it isn't pushed or checked, and sunken when it is pushed or checked.
multiline	m	BS_MULTILINE	Wraps the button text to multiple lines
notify	n	BS_NOTIFY	Causes radio button to send dialog BN_DBLCLK, BN_DISABLE, and BN_ENABLE notifications when the relevant event occurs.
flat	f	BS_FLAT	Specifies that the button is two-dimensional; it does not use the default shading to create a 3-D image.
flathover	h	-	flat style, no border under mouseover
nodestroy	-		only valid if control has *<id>* *of "cancel"*; same as specifying an *<id>* of "escapable"

The following styles apply only to buttons, and only to those with images (set with set_image): without any of the styles below, the image set will be up against the left edge of the focus rectangle, and the caption up against the right:

Styles Applying to Buttons with Images	
this style name	**means**
imgright	image to right of caption; if no caption, no effect
imgvert	image above caption; if no caption, no effect
imgover	image centred over caption; if no caption , no effect
imgfill	image fills button; if caption, no effect, *unless* the caption consists of just two characters in the form "&*L*", in which case caption will be hidden by full-button image, but willstill be there (and therefore the shortcut key *L* will work)
imgspacing:nn	spacing between image and text set to nn pixels; if no caption , no effect colon is mandatory. nn may be negative, in which case text overlaps boundary of image
imgpressedtb	image moves up and down when button pressed, instead of down and right

If there's no caption, the image will always be centred on the button.

Note that the image alignment styles apply orthogonally to the general text alignment styles ("left", "right", etc) listed in the previous table. So you can have, for instance, "imgright left": image to right of caption, but the whole lot up against right edge of the button.

page 167 of 260

II.1.2 Messages and Services

The following named messages are defined and can be used in dialog.send_message:

Named Messages for (General) Buttons (including Check Box Controls, Radio Buttons)				
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful
getstate	-	BM_GETSTATE	no	no
setstate	-	BM_SETSTATE	yes	no
setstyle	-	BM_SETSTYLE	yes	no

You can use set_value to change the text on a button.

II.1.3 Colours

I think colours work ok with set_colour and define_control now..

II.2: Groups

<*control_type*> **value:** group

Used in sample scripts:
dialogPluginDemo1.powerpro, dialogPluginDemo4.powerpro, dialogPluginDemo6.powerpro, regexDialog.powerpro (via associated *dialog_definition_file*), regexDialogScintilla.powerpro (via associated *dialog_definition_file*)

A group box cannot be selected, so it has no check state, focus state, or push state.

An application cannot send messages to a group box.

To paraphrase the MSDN docs: "The first control in a group combines the WS_GROUP and WS_TABSTOP styles so that the user can move from group to group by using the TAB key. If the group contains radio buttons, the application should apply the WS_TABSTOP style only to the first control in the group. Windows automatically moves the style when the user moves between controls in the group. This ensures that the input focus will always be on the most recently selected control when the user moves to the group using the TAB key." Which means, since the is applies to all controls by default, you should apply the "group" style to the first control in a group, and the "notab" style to the remaining ones.

II.2.1 Styles

The group box has only one style, defined by the constant BS_GROUPBOX.

In addition to styles applying to all controls, there are styles that apply everything that Microsoft has decided are of the BUTTON class, including groups. See above.

II.3: Radio Buttons, Check Box Controls, Three-States

<control_type> **values:** radiobutton, checkbox, 3state

Radio buttons used in:
dialogPluginDemo1.powerpro, dialogPluginDemo6.powerpro, regexDialog.powerpro (via associated *dialog_definition_file*), regexDialogScintilla.powerpro (via associated *dialog_definition_file*)

Check box control used in:
regexDialog.powerpro (via associated *dialog_definition_file*), regexDialogScintilla.powerpro (via associated *dialog_definition_file*)

Three-state used in: dialogPluginDemo1.powerpro script

Three-states are check box controls that allow a third, greyed-out state.

Normally radio buttons, and sometimes check box controls and three-states are often placed in groups. See Section II.2 above concerning the styles you should apply to controls in groups.

II.3.1 Styles

In addition to styles applying to all controls, there are styles that apply everything that Microsoft has decided are of the BUTTON class, including radio buttons and check box controls. See above.

There are a few other button-related styles, but probably none of interest. Avoid the BS_OWNERDRAW style, it needs more support. Also no support yet for BS_USERBUTTON, BS_ICON or BS_BITMAP.

II.3.2 Messages and Services

The following named messages are defined and can be used in dialog.send_message:
they either take or return 0 (for unchecked), 1 if (for checked/selected) and 2 (for indeterminate):

Named Messages for Check Box Controls, Radio Buttons

this message name	or this letter	#define'd symbol	wParam meaningful	lParam meaningful
getcheck	g	BM_GETCHECK	no	no
setcheck	s	BM_SETCHECK	yes	no

There are a few other messages that can be sent to any kind of button, including radio buttons and check box controls. See above. Use those messages to set and check radio buttons and check box controls, find out if they're checked or chosen, and get their state.

You can use set_value to change the text of any of these controls.

II.3.3 Colours

Text (foreground) and background colours work as advertised.

II.4: Static Controls

<*control_type*> **value:** static

Static controls are used in *all* sample scripts.

You can simulate a static with a clickable url by providing a set_response for a "left" <*mouse_event*> and a set_colour with a <*mouse_state*> to simulate a text or background change when your mouse is over the control. See dialogPluginDemo7.powerpro for an example.

II.4.1 Styles

In addition to styles applying to all controls, there are:

page 172 of 260

Styles for Statics			
this style name	**or this letter**	**#define'd symbol**	**means**
left	l	SS_LEFT	Designates a simple rectangle and displays the given text flush-left in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next flush-left line.
leftnowordwrap	l	SS_LEFTNOWORDWRAP	Designates a simple rectangle and displays the given text flush-left in the rectangle. Tabs are expanded, but words are not wrapped. Text that extends past the end of a line is clipped.
centreimage	-	SS_CENTERIMAGE	Use to centre an image. If the image is larger than the control, it will be clipped; if it is smaller, the empty space around the image will be filled by the colour of the pixel in the upper left corner of the image.
centre	c	SS_CENTER	Designates a simple rectangle and displays the given text centred in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next centred line.
right	r	SS_RIGHT	Designates a simple rectangle and displays the given text flush-right in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next flush-right line.
blackframe	b	SS_BLACKFRAME	Specifies a box with a frame drawn with the same colour as window frames. The default is black.
grayframe	g	SS_GRAYFRAME	Specifies a box with a frame drawn with the same colour as the screen background (desktop). The default is grey.
whiteframe	w	SS_WHITEFRAME	Specifies a box with a frame drawn with the same colour as the window background. The default is white.
simple		SS_SIMPLE	Designates a simple rectangle and displays a single line of text flush-left in the rectangle. The line of text cannot be shortened or altered in any way. (The control's parent window or dialog box must not process the WM_CTLCOLOR message.)
sunken	s	SS_SUNKEN	Draws a half-sunken border around the static
rightjust	r	SS_RIGHTJUST	Specifies that the lower right corner of a static control with the SS_BITMAP or SS_ICON style is to remain fixed when the control is resized. Only the top and left sides are adjusted to accommodate a new bitmap or icon.
notify	n	SS_NOTIFY	Causes static to send dialog STN_CLICKED, STN_DBLCLK, STN_DISABLE, and STN_ENABLE notifications when the relevant event occurs. Automatically inserted if you invoke set_response to set a response to a <*mouse_event*>s; or set_colour with a <*mouse_state*>.

page 173 of 260

There are other styles, allowing filled rectangles, other kinds of edged rectangles, and treatment of text with ellipsis. See dialog-related_defines.txt: Look for "SS_". Avoid the SS_OWNERDRAW style, it needs more support. And for now, no support for images in statics, so avoid SS_*IMAGE styles.

II.4.2 Messages and Services

As yet I've not provided any message names for statics.

You can use set_value to change the text in a static.

II.4.3 Colours

Text (foreground) and background colours work as advertised.

page 174 of 260

II.5: Edit Controls

<*control_type*> **value:** editbox

Used in sample scripts:
dialogPluginDemo1.powerpro, dialogPluginDemo2.powerpro, dialogPluginDemo4.powerpro, dialogPluginDemo6.powerpro, regexDialog.powerpro (via associated *dialog_definition_file*), regexDialogScintilla.powerpro (via associated *dialog_definition_file*)

Use get_value to get text, selected text or the font description for an edit control.

In addition to the usual values of get_value <*property*>s, you can also use the <property> "selectlen" to return the length of the current selection and "selected" or "select" to return the selection itself.

Use set_value or one of its aliases (e.g. *set, modify*) to change the text of an edit control.

The maximum length of a text string (either when defining the control or using set_value to change the displayed text) is 32,766 bytes for a single-line edit control, and 65,535 bytes for a multiline edit control. The **set_value** and **define_control** services will enforce those limits.

Edit controls can now be assigned a <*right_click_command*> with define, define_control or set_response, overriding the built-in context menu. (But a “right” <*mouse_event*> with a keyboard modifier like “ctrl” won’t override it; nor will any other mouse event, e.g. “middle”).

The dialog plugin supports two other kinds of edit controls (rich edit and scintilla). See section II.18.4 "Which Type of Edit Control?" comparing them..

II.5.1 Styles

In addition to styles applying to all controls, there are the following. note that some, in particular ES_MULTILINE. can’t be altered once an edit control is created.

Styles for Edit Controls			
this style name	**or this letter**	**#define'd symbol**	**means**
vscroll		WS_VSCROLL	apply vertical scrollbar
hscroll		WS_HSCROLL	apply horizontal scrollbar
multiline	m	ES_MULTILINE	see note
left	l	ES_LEFT	Aligns text flush left.
centre	c	ES_CENTER	Centres text in a multiline edit control.
right	r	ES_RIGHT	Aligns text flush right in a multiline edit control.
uppercase	u	ES_UPPERCASE	Converts all characters to uppercase as they are typed into the edit control.
lowercase	l	ES_LOWERCASE	Converts all characters to lowercase as they are typed into the edit control.
password	p	ES_PASSWORD	Displays all characters as an asterisk (*) as they are typed into the edit control. An application can use the SetPasswordChar member function to change the character that is displayed.
autovscroll		ES_AUTOVSCROLL	Automatically scrolls text up one page when the user presses ENTER on the last line
autohscroll		ES_AUTOHSCROLL	Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position 0.
nohidesel		ES_NOHIDESEL	Normally, an edit control hides the selection when the control loses the input focus and inverts the selection when the control receives the input focus. Specifying ES_NOHIDESEL deletes this default action.
oemconvert	o	ES_OEMCONVERT	see note
readonly		ES_READONLY	Prevents the user from entering or editing text in the edit control.
wantreturn		ES_WANTRETURN	see note

ES_MULTILINE: Designates a multiple-line edit control. (The default is single line.) If the ES_AUTOVSCROLL style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. If ES_AUTOVSCROLL is not given, the edit control shows as many lines as possible and beeps if ENTER is pressed when no more lines can be displayed. If the ES_AUTOHSCROLL style is specified, the multiple-line edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press ENTER. If ES_AUTOHSCROLL is not given, the control automatically wraps words to the beginning of the next line when necessary; a new line is also started if ENTER is pressed. The position of the word-wrap is determined by the window size. If the window size changes, the word-wrap position changes and the text is redisplayed. Multiple-line edit controls can have scroll bars. An edit control with scroll bars processes its own scroll-bar messages. Edit controls without scroll bars scroll as described above and process any scroll messages sent by the parent window.

Undocumented, but it seems you can’t add or remove the ES_MULTILINE style on the fly to an existing edit control. If you want the appearance of doing so, use two edit controls, one with MULTILINE, one without; then hide and show them as required.

ES_WANTRETURN: Specifies that a carriage return be inserted when the user presses the ENTER key while entering text into a multiple-line edit control in a dialog box. Without this style, pressing the ENTER key has the same effect as pressing the dialog boxes default pushbutton. This style has no effect on a single-line edit control.

ES_OEMCONVERT: Text entered in the edit control is converted from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the AnsiToOem Windows function to convert an ANSI string in the edit control to OEM characters. This style is most useful for edit controls that contain filenames.

I think that's all the styles there are.

II.5.2 Messages and Services

You can use set_value to change the text in an edit box.

Here's a list of messages that also work for rich edit controls. Following this table is a short list of messages that *only* apply to plain edit controls.

Messages for Edit and Rich edit controls					
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful	**description**
getsel		EM_GETSEL	see note	see note	retrieves the starting and ending character positions of the current selection
setsel		EM_SETSEL	int; start char	int; end char	selects a range of characters; If the start is 0 and the end is –1, all text is selected; if start is –1, any current selection is deselected.
scroll		EM_SCROLL	int; how	no	scrolls the text vertically in a multiline control; for wParam see SB_ consts among EM_ messages in dialog-related_defines.txt
scrollcaret		EM_SCROLLCARET	no	no	scrolls the caret into view
setmargins		EM_SETMARGINS	int; EC_ const	yes	sets the widths of the left and right margins; lParam: low-order word is width left margin, high-order word width of right
getmodify		EM_GETMODIFY	no	no	retrieves the state of an edit control's modification flag
setmodify		EM_SETMODIFY	0/1	no	sets or clears the modification flag; (1 = modified)
getlinecount		EM_GETLINECOUNT	no	no	retrieves the number of lines in a multiline edit control
lineindex		EM_LINEINDEX	yes	no	retrieves the character index of the first character of a specified line in a multiline edit control.
getthumb		EM_GETTHUMB	no	no	retrieves the position of the scroll box (thumb) in the vertical scroll bar of a multiline edit control
linelength		EM_LINELENGTH	yes	no	retrieves the length, in characters, of a line in an edit control

page 178 of 260

Messages for Edit and Rich edit controls					
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful	**description**
replacesel		EM_REPLACESEL	1/0; can be undone or not	string	replaces the current selection in an edit control with the specified text
getline		EM_GETLINE	yes (index)	string	lParam: first word of buffer set to buffer size
canundo		EM_CANUNDO	no	no	determines whether there are any actions in an edit control's undo queue
undo		EM_UNDO	no	no	undoes the last edit control operation in the control's undo queue
emptyundobuffer		EM_EMPTYUNDOBUFFER	no	no	resets the undo flag of an edit control
linefromchar		EM_LINEFROMCHAR	yes	no	retrieves the index of the line that contains the specified character index in a multiline edit control
getfirstvisibleline		EM_GETFIRSTVISIBLELINE	no	no	retrieves the zero-based index of the uppermost visible line in a multiline edit control
setpasswordchar		EM_SETPASSWORDCHAR	char (0 to rem)	no	sets or removes the password character
setreadonly		EM_SETREADONLY	yes	no	sets or removes the read-only style (ES_READONLY)
setlimittext		EM_SETLIMITTEXT	yes	no	sets the maximum amount of text that the user can type into control

getsel: wParam: pointer to a buffer that receives the starting position of the selection; can be NULL. lParam: pointer to a buffer that receives the position of the first nonselected character after the end of the selection; can be NULL. Returns is a zero-based value with the starting position of the selection in the low-order word and the position of the first character after the last selected character in the high-order word. If either of these values exceeds 65,535, the return value is –1. Best to use EM_EXSETSEL if you've got a rich edit control with more than 64k characters in it.

setmargins complicated and depends on wether plain or rich edit box, and which version of rich edit. See http://msdn.microsoft.com/library/en-us/shellcc/platform/commctls/editcontrols/editcontrolreference/editcontrolmessages/em_setmargins.asp

page 179 of 260

Messages for Edit Controls Only					
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful	
fmtlines		EM_FMTLINES	yes	no	Specifies whether soft line-break characters are to be inserted
setrect		EM_SETRECT	no	RECT*	set the formatting rectangle of a multiple-line edit control; causes text to be redrawn
setrectnp		EM_SETRECTNP	no	RECT*	as above, no redraw
posfromchar		EM_POSFROMCHAR	0-based char ndx	no	retrieves the client area coordinates of a specified character
charfrompos		EM_CHARFROMPOS	no	integer	lParam: X in low-order word, Y in high-order word
linescroll		EM_LINESCROLL	int; no chars to scroll horizontally	int; no chars to scroll vertically	scrolls the text in a multiline edit control
settabstops		EM_SETTABSTOPS	no tab stops	INT array	sets the tab stops in a multiline edit control

settabstops: complicated, see http://msdn.microsoft.com/library/en-us/shellcc/platform/commctls/editcontrols/editcontrolreference/editcontrolmessages/em_settabstops.asp

II.5.3 Colours

Text (foreground) and background colours work as advertised.

II.6: List Controls

<*control_type*> **value:** lixtbox

Used in sample scripts:
dialogPluginDemo1.powerpro, dialogPluginDemo4.powerpro, dialogPluginDemo6.powerpro

You can use set_value to insert or add items in a list box, and the following variants of get_value:

> **dialog.get_value(***<target>***)**
> **dialog.get_value(***<target>,* "selected"**)**
> **dialog.get_value(***<target>,* "all"**)**
> **dialog.get_value(***<target>,* "selectedIndex"**)**
> **dialog.get_value(***<target>,* "selectedCount"**)**

dialog.get_value(*<target>***)**
dialog.get_value(*<target>,* "selected"**)**

Gets selected items, separated by newlines (\n)

dialog.get_value(*<target>,* "all"**)**

Gets all items, separted by newlines (\n)

dialog.get_value(*<target>,* "selectedIndex"**)**

Gets the indices of selected items, separtaed by spaces

dialog.get_value(*<target>,* "selectedCount"**)**

Gets the number of items selected.

II.6.1 Styles

In addition to styles applying to all controls, there are:

Styles for List Controls			
this style name	**or this letter**	**#define'd symbol**	**means**
novscroll		WS_VSCROLL	suppress vertical scroll bar
hscroll		WS_HSCROLL	horizontal scroll bar
extendedsel	e	LBS_EXTENDEDSEL	The user can select multiple items using the SHIFT key and the mouse or special key combinations.
multicolumn	c	LBS_MULTICOLUMN	Specifies a multicolumn list box that is scrolled horizontally. The SetColumnWidth member function sets the width of the columns.
multiplesel	m	LBS_MULTIPLESEL	String selection is toggled each time the user clicks or double-clicks the string. Any number of strings can be selected.
noredraw		LBS_NOREDRAW	List box display is not updated when changes are made. This style can be changed at any time by sending a WM_SETREDRAW message.
notify	n	LBS_NOTIFY	Parent window receives an input message whenever the user clicks or double-clicks a string.
sort	o	LBS_SORT	Strings in the list box are sorted alphabetically.
standard		LBS_STANDARD	Strings in the list box are sorted alphabetically, and the parent window receives an input message whenever the user clicks or double-clicks a string. The list box contains borders on all sides.
usetabstops	u	LBS_USETABSTOPS	see note
disablenoscroll	d	LBS_DISABLENOSCROLL	The list box shows a disabled vertical scroll bar when the list box does not contain enough items to scroll. Without this style, the scroll bar is hidden when the list box does not contain enough items.
nointegralheight		LBS_NOINTEGRALHEIGHT	The size of the list box is exactly the size specified by the application when it created the list box. Usually, Windows sizes a list box so that the list box does not display partial items.
wantkeyboardinput		LBS_WANTKEYBOARDINPUT	The owner of the list box receives WM_VKEYTOITEM or WM_CHARTOITEM messages whenever the user presses a key while the list box has input focus. This allows an application to perform special processing on the keyboard input.

LBS_USETABSTOPS: Allows a list box to recognize and expand tab characters when drawing its strings. The default tab positions are 32 dialog units. (A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to one-fourth of the current dialog base width unit. The dialog base units are computed based on the height and width of the current system font. The GetDialogBaseUnits Windows function returns the current dialog base units in pixels.)

There are a few other styles. See dialog-related_defines.txt: Look for "LBS_". Avoid LBS_OWNERDRAWxxx styles, they need more support.

II.6.2 Messages and Services

The following named messages are defined for list controls and can be used in dialog.send_message:

Named Messages for List Controls				
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful
add	a	LB_ADDSTRING	no	string
delete	s	LB_DELETESTRING	yes	no
insert	i	LB_INSERTSTRING	yes	string
select	s	LB_SELECTSTRING	yes	string
setcursor	t	LB_SETCURSEL	yes (0-based idx)	

II.6.3 Colours

Text (foreground) and background colours work as advertised.

II.6.4 Removing Strings

Use the clear service to remove all strings or a single string from the control. If you want to delete a specific string, provide its index index (lowest legal value 0 or 1) as a second argument.

II.7: Combo Box Controls

<*control_type*> **value:** combobox

Used in sample script dialogPluginDemo1.powerpro

Note that the dimensions of a combobox control has to be large enough to show the list box, even if it's normally not visible (i.e. "dropdown" or "dropdownlist" styles)

You can use set_value to change the text in the edit box portion of a combo, or to insert or add items in a list box., and the following variants of get_value:

> **dialog.get_value(***<target>***)**
> **dialog.get_value(***<target>,* "selected"**)**
> **dialog.get_value(***<target>,* "all"**)**
> **dialog.get_value(***<target>,* "selectedIndex"**)**

dialog.get_value(*<target>***)**
dialog.get_value(*<target>,* "selected"**)**

Gets the selected item (the item in the edit box).

dialog.get_value(*<target>,* "all"**)**

Gets all items, separated by newlines (\n)

dialog.get_value(*<target>,* "selectedIndex"**)**

Gets the index the selected item

II.7.1 Styles

In addition to styles applying to all controls, there are:

Styles for Combo Box Controls			
this style name	**or this letter**	**#define'd symbol**	**means**
novscroll		WS_VSCROLL	suppress vertical scroll bar
autohscroll	d	CBS_AUTOHSCROLL	Automatically scrolls the text in the edit control to the right when the user types a character at the end of the line. If this style is not set, only text that fits within the rectangular boundary is allowed.
dropdown	d	CBS_DROPDOWN	Similar to CBS_SIMPLE, except that the list box is not displayed unless the user selects an icon next to the edit control.
dropdownlist	l	CBS_DROPDOWNLIST	Similar to CBS_DROPDOWN, except that the edit control is replaced by a static-text item that displays the current selection in the list box.
oemconvert	d	CBS_OEMCONVERT	see note
simple	s	CBS_SIMPLE	The list box is displayed at all times. The current selection in the list box is displayed in the edit control.
sort	o	CBS_SORT	Automatically sorts strings entered into the list box.
disablenoscroll		CBS_DISABLENOSCROLL	The list box shows a disabled vertical scroll bar when the list box does not contain enough items to scroll. Without this style, the scroll bar is hidden when the list box does not contain enough items.
nointegralheight		CBS_NOINTEGRALHEIGHT	Specifies that the size of the combo box is exactly the size specified by the application when it created the combo box. Normally, Windows sizes a combo box so that the combo box does not display partial items.

CBS_OEMCONVERT: Text entered in the combo box edit control is converted from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the AnsiToOem Windows function to convert an ANSI string in the combo box to OEM characters. This style is most useful for combo box controls that contain filenames and applies only to combo box controls created with the CBS_SIMPLE or CBS_DROPDOWN styles.

page 186 of 260

There are a few other styles. See dialog-related_defines.txt: Look for "CBS_". Avoid CBS_OWNERDRAWxxx styles, they need more support.

II.7.2 Messages and Services

The following named messages for combo box controls are defined and can be used in dialog.send_message:

Named Messages for Combo Box Controls				
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful
add	a	CB_ADDSTRING	no	string
delete	s	CB_DELETESTRING	yes	no
insert	i	CB_INSERTSTRING	yes	string
select	s	CB_SELECTSTRING	yes	string
find	f	CB_FINDSTRING	yes	string
reset	-	CB_RESETCONTENT	no	no
setcursor	t	CB_SETCURSEL	yes (0-based idx)	no

II.7.3 Colours

Text (foreground) colour doesn't seem to work. Might be style-dependent.

Background colour doesn't work for the CBS_DROPDOWN ("dropdown") or CBS_SIMPLE ("simple") but does for the edit portion only for CBS_DROPDOWNLIST ("dropdownlist"). Go figger. It might work for some other styles.

II.7.4 Removing Strings

Use the clear service to remove all strings or a single string from the control. If you want to delete a specific string, provide its index index (lowest legal value 0 or 1) as a second argument.

II.8: The Scrollbar Control

<*control_type*> **value:** scrollbar

Used in sample script dialogPluginDemo1.powerpro

In the main example script (dialogPluginDemo.powerpro) I use a scrollbar to fake a spinner: in the current version you're probably best using an up-down control for this purpose. You can use them as pseudo-sliders too, setting the page value to a small value using dialog.set_range.

For more about scrollbars see:

http://msdn.microsoft.com/library/en-us/shellcc/platform/commctls/scrollbars/aboutscrollbars.asp?frame=true

II.8.1 Styles

In addition to styles applying to all controls, there are:

Styles for Scrollbars			
this style name	**or this letter**	**#define'd symbol**	**means**
horizontal	h	SBS_HORZ	Designates a horizontal scroll bar
vertical	v	SBS_VERT	Designates a vertical scroll bar

Probably best to use exactly one of these styles. No idea what happens if your specify neither, or, god help you, both.

There are a bunch of other weird styles, all to do with sizing and placement of scrollbar elements. See the URL above and dialog-related_defines.txt. Look for "SBS_".

II.8.2 Messages and Services

As yet I've not provided any message names for scrollbars.

There are three services that apply to scroll bars: set_range, set_value, and get_value. You'll almost always have to call dialog.set_range after creating or running a dialog, to set your desired <*position*>, <*min*>, <*max*> and <*page_size*> (if you don't call it, or omit a parameter when you do, defaults are: <*min*> of 0, <*max*> of 99, and <*page_size*> of 1.)

If you provide a *<script_to_call>* for a scrollbar control, it will fire whenever the user moves the scrollbar, whether by mouse or keyboard (which, if you care, causes either a WM_VSCROLL or WM_HSCROLL message to be generated).

page 188 of 260

II.8.3 Colours

Text (foreground) colour has no meaning for scrollbars, so the *<foreground>* parameters of define_control and calls to set_colour are ignored.

II.9: The Spinner Control

<*control_type*> **value:** spinner

Used in sample script dialogPluginDemo2.powerpro

Unlike the scrollbar, a spinner can be associated with a buddy control (typically an edit or static) which automatically updates as the value of the spinner changes. You have to send the **setbuddyint** message using as an argument the window handle of the buddy control (obtained with dialog.get_hwnd). With appropriate styles you can force the spinner to position itself next to it's buddy. You don't have to monitor user changes to the buddy edit control; if it's value changes, the spinner will automatically be updated.

If you want your associated control to display anything other than an integer, the buddy mechanism won't work for you; you'll have to catch events in both the spinner and the associated control, rather as I've done for the scrollbar used as a spinner in the test script dialogPluginDemo1.powerpro.

page 190 of 260

II.9.1 Styles

In addition to styles applying to all controls, there are:

Styles for Spinners			
this style name	**or this letter**	**#define'd symbol**	**means**
wrap	w	UDS_WRAP	Causes the position to "wrap" if it is incremented or decremented beyond the ending or beginning of the range.
setbuddyint	b	UDS_SETBUDDYINT	Causes the control to set the text of the buddy window when the position changes. The text consists of the position formatted as a decimal or hexadecimal string.
alignright	r	UDS_ALIGNRIGHT	Positions the spin button control next to the right edge of the buddy window. The width of the buddy window is decreased to accommodate the width of the control.
alignleft	l	UDS_ALIGNLEFT	Positions the spin button control next to the left edge of the buddy window. The buddy window is moved to the right and its width decreased to accommodate the width of the control.
autobuddy	a	UDS_AUTOBUDDY	Automatically selects the previous window in the Z-order as the controls buddy window.
arrowkeys	k	UDS_ARROWKEYS	Causes the control to increment and decrement the position when the UP ARROW and DOWN ARROW keys are pressed.
horz	h	UDS_HORZ	Causes the controls arrows to point left and right instead of up and down
nothousands	n	UDS_NOTHOUSANDS	Does not insert a thousands separator between every three decimal digits.
hottrack		UDS_HOTTRACK	Causes the control to exhibit "hot tracking" behaviour. That is, it highlights the control's buttons as the mouse passes over them. This style requires win98 or win2000. If the system is running 95 or NT4, the flag is ignored. The flag is also ignored on XP when the desktop theme overrides it.

There are a bunch of other weird styles, all to do with sizing and placement of

page 191 of 260

spinner elements. See the URL above and dialog-related_defines.txt. Look for "UDS_".

II.9.2 Messages and Services

The following named messages are defined for spinners and can be used in dialog.send_message:

Named Messages for Services				
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful
setbase	b	UDM_SETBASE	yes	no
setbuddy	u	UDM_SETBUDDY	yes	no

The **setbase** message sets the radix base for an up-down control. The base value determines whether the buddy window displays numbers in decimal or hexadecimal digits. Hexadecimal numbers are always unsigned, and decimal numbers are signed.

The **setbuddy** message sets the buddy window wParam must be the window handle of the buddy window (obtained with dialog.get_hwnd)

There are three services that apply to spinners: set_range, set_value, and get_value. You'll almost always have to call dialog.set_range after creating or running a dialog, to set your desired <*min*> (must be more than -0x7fff) and <*max*> (must be less than 0x7fff) (if you don't call it, defaults are: <*min*> of 0 and <*max*> of 99). The fourth parameter of set_range is ignored.

If you provide a *<script_to_call>* for a spinner, it will fire whenever the user moves the spinner, whether by mouse or keyboard (which, if you care, causes either a WM_VSCROLL or WM_HSCROLL message to be generated).

II.9.3 Colours

Colours can't be set for a spinner, so the <*foreground*> and <*background*> parameters of define_control and calls to set_colour are ignored.

II.10: The Slider (Tracker Bar) Control

<*control_type*> **value:** slider

Used in sample scripts:
dialogPluginDemo2.powerpro, dialogPluginDemo4.powerpro

The slider or tracker bar is another specialised variant of a scrollbar, that gives you a thumb to move along a scale to indicate a quantity. Details at:

http://msdn.microsoft.com/library/en-us/shellcc/platform/commctls/trackbar/trackbar.asp?frame=true

II.10.1 Styles

In addition to styles applying to all controls, there are:

Styles for Sliders			
this style name	**or this letter**	**#define'd symbol**	**means**
vert	v	TBS_VERT	Orients the slider vertically. If you do not specify an orientation, the slider is oriented horizontally.
left	l	TBS_LEFT	Displays tick marks on the left of a vertical slider. Can be used with the TBS_RIGHT style to display tick marks on both sides of the slider control.
top	t	TBS_TOP	Displays tick marks on the top of a horizontal slider. Can be used with the TBS_BOTTOM style to display tick marks on both sides of the slider control.
both	b	TBS_BOTH	Displays tick marks on both sides of the slider in any orientation.
bottom	0	TBS_BOTTOM	Displays tick marks on the bottom of a horizontal slider. Can be used with the TBS_TOP style to display tick marks on both sides of the slider control.
right	0	TBS_RIGHT	Displays tick marks on the right of a vertical slider. Can be used with the TBS_LEFT style to display tick marks on both sides of the slider control.
autoticks	a	TBS_AUTOTICKS	Creates a slider that has a tick mark for each increment in its range of values. These tick marks are added automatically when an application calls the SetRange member function. You cannot use the SetTic and SetTicFreq member functions to specify the position of the tick marks if you use this style. Use the ClearTics member function instead.
enableselrange	e	TBS_ENABLESELRANGE	see note
fixedlength	f	TBS_FIXEDLENGTH	Allows the thumb's size to be changed
nothumb	h	TBS_NOTHUMB	No thumb!!
noticks	n	TBS_NOTICKS	Creates a slider that does not display tick marks.

TBS_ENABLESELRANGE: if set, you can specify a "selection range," which restricts the user to a specified portion of the total range. The logical units do not change, but only a subset of them are available for use. The slider highlights the available range and displays triangular tick marks at the start and end. Typically, an application handles the slider's notification messages and sets the slider's selection range according to the user's input.

When a slider control has this style, the tick marks at the starting and ending positions of a selection range are displayed as triangles (instead of vertical dashes) and the selection range is highlighted. For example, selection ranges might be useful in a simple scheduling application. The user could select a range of tick marks corresponding to hours in a day to identify a scheduled meeting time.

There are other styles. See dialog-related_defines.txt. Look for "TBS_".

II.10.2 Messages and Services

There are three services that apply to sliders: set_range, set_value, and get_value. You'll almost always have to call dialog.set_range after creating or running a dialog, to set your desired <*min*>*,* <*max*> and <*page_size*> (if you don't call it, or omit a parameter when you do, defaults are: <*min*> of 0, <*max*> of 99 and <*page_size*> of 1.)

If you provide a *<script_to_call>* for a slider, it will fire whenever the user moves the slider, whether by mouse or keyboard (which, if you care, causes either a WM_VSCROLL or WM_HSCROLL message to be generated).

The following named messages are defined for sliders and can be used in dialog.send_message:

Named Messages for Sliders					
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful	**means**
getpos	g	TBM_GETPOS	no	no	Returns current logical position of the slider, i.e. integer values in the slider's range of minimum to maximum slider positions.
settic	t	TBM_SETTIC	yes	no	sets a tick mark in a slider at the specified logical position
cleartics	c	TBM_CLEARTICS	0/1	no	removes the current tick marks from a slider. This message does not remove the first and last tick marks that are created automatically by the slider. Only param is redraw flag: If TRUE, the message redraws the slider after receiving the message; if FALSE, doesn't.
setselstart	s	TBM_SETSELSTART	yes	no	sets the starting logical position of the current selection range in a slider
setselend	e	TBM_SETSELEND	0/1	yes (pos)	sets the ending logical position of the current selection range in a trackbar; ignored if the slider doesn't have the TBS_ENABLESELRANGE style. wParam is redraw flag: If TRUE, the message redraws the slider after receiving the message; if FALSE, doesn't
getnumtics	n	TBM_GETNUMTICS	no	no	retrieves the number of tick marks in slider
getselstart	0	TBM_GETSELSTART	no	no	retrieves the starting position of the current selection range in a slider. A slider can have a selection range only if you specified the TBS_ENABLESELRANGE style when you created it.
getselend	0	TBM_GETSELEND	no	no	sets the ending logical position of the current selection range in a slider. These messages are ignored if the slider doesn't have the

Named Messages for Sliders					
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful	**means**
					TBS_ENABLESELRANGE style.
clearsel	c	TBM_CLEARSEL	0/1	no	clears the current selection range in a slider; wParam is redraw flag: If TRUE, the message redraws the slider after receiving the message; if FALSE, doesn't
setticfreq	f	TBM_SETTICFREQ	yes	no	The slider must have the TBS_AUTOTICKS style to use this message
setpagesize	p	TBM_SETPAGESIZE	no	integer	sets the number of logical positions the slider moves in response to keyboard input, such as the PAGE UP or PAGE DOWN
setlinesize	l	TBM_SETLINESIZE	no	integer	retrieves the number of logical positions the trackbar's slider moves in response to keyboard input from the arrow keys, such as the RIGHT ARROW or DOWN ARROW keys. The logical positions are the integer increments in the trackbar's range of minimum to maximum slider positions. The default setting for the line size is 1
setthumblength	b	TBM_SETTHUMBLENGTH	yes	no	message sets the length of the thumb; ignored if the slider doesn't have the TBS_FIXEDLENGTH style

There are other messages, mostly to do with retrieving slider settings. See dialog-related_defines.txt. Look for "TBM_".

page 199 of 260

II.10.3 Colours

Background colours work as advertised; foreground colours don't, so you can't colour the thumb thingie. I might get around to fixing that.

II.11: The Progress Control

<control_type> **value:** progress

Used in sample script dialogPluginDemo2.powerpro

II.11.1 Styles

In addition to styles applying to all controls, there are:

Styles for Progress Bars			
this style name	**or this letter**	**#define'd symbol**	**means**
smooth	s	PBS_SMOOTH	The progress bar displays progress status in a smooth scrolling bar instead of the default segmented bar. When this style is present, the control automatically reverts to the Classic Theme appearance on Windows XP or later. Windows 95 and NT4 r
vertical	v	PBS_VERTICAL	The progress bar displays progress status vertically, from bottom to top. Windows 95 and NT4 require Internet Explorer 3.0 or later to support this option.
marquee	m	PBS_MARQUEE	[Requires Windows XP or later] The progress bar moves like a marquee; that is, each change to its position causes the bar to slide further along its available length until it wraps around to the other side. A bar with this style has no defined position. Each attempt to change its position will instead slide the bar by one increment. This style is typically used to indicate an ongoing operation whose completion time is unknown.

II.11.2 Messages and Services

The following named messages are defined for progress bars and can be used in dialog.send_message:

Named Messages for Progress Bars				
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful
setpos	s	PBM_SETPOS	yes	no
deltapos	d	PBM_DELTAPOS	yes	no
setstep	p	PBM_SETSTEP	yes	no
stepit	t	PBM_STEPIT	no	no

PBM_DELTAPOS: advances the current position of a progress bar by a specified increment

PBM_SETSTEP: specifies the step increment for a progress bar. The step increment is the amount by which the progress bar increases its current position whenever it receives a PBM_STEPIT message. By default, the step increment is set to 10.

There are three services that apply to progress bars: set_range, set_value, and get_value. You'll almost always have to call dialog.set_range after creating or running a dialog, to set your desired <*min*>, <*max*> and <*step*> (if you don't call it, or omit a parameter when you do, defaults are: <*min*> of 0, <*max*> of 99 and <*step*> of 10).

II.11.3 Colours

Text (foreground) and background colours work as advertised, unless you're using a non-classic theme, in which case they don't. Don't ask me, ask Microsnot.

II.12: The Date-Time Control

<control_type> **value:** datetime

Used in sample script dialogPluginDemo2.powerpro

To use the date-time picker on Windows 95/NT you must have updated comctl32.dll to version 4.70 or later.

Dates accepted for input or produced for output by the control follow the same format as the dates produced by PowerPro keywords and functions like date and formatdate. Dates are represented as eight digit number yyyymmdd, such as 20021028 for October 28, 2002 or 20030101 for January 1, 2003.

II.12.1 Styles

In addition to styles applying to all controls, there are:

Styles for Date-Time controls			
this style name	**or this letter**	**#define'd symbol**	**means**
updown	u	DTS_UPDOWN	Provides an up-down control to the right of the control to modify date-time values, which replaces the of the drop-down month calendar that would otherwise be available.
shownone	n	DTS_SHOWNONE	Displays a checkbox inside the control that users can uncheck to make the control have no date/time selected. Whenever the control has no date/time, Gui Submit and GuiControlGet will retrieve a blank value (empty string).
shortdateformat	s	DTS_SHORTDATEFORMAT	Displays the date in short format. In some locales, it looks like 6/1/05 or 6/1/2005. On older operating systems, a two-digit year might be displayed. This is why DTS_SHORTDATECENTURYFORMAT is the default and not DTS_SHORTDATEFORMAT.
longdateformat	l	DTS_LONGDATEFORMAT	Format option "LongDate". Displays the date in long format. In some locales, it looks like Wednesday, June 01, 2005.
shortdatecenturyformat	c	DTS_SHORTDATECENTURYFORMAT	Format option blank/omitted. Displays the date in short format with four-digit year. In some locales, it looks like 6/1/2005. If the system's version of Comctl32.dll is older than 5.8, this style is not supported and DTS_SHORTDATEFORMAT is automatically substituted.
timeformat	t	DTS_TIMEFORMAT	Format option "Time". Displays only the time, which in some locales looks like 5:31:42 PM.
rightalign	r	DTS_RIGHTALIGN	The calendar will drop down on the right side of the control instead of the left.

There are other styles. See dialog-related_defines.txt. Look for "DTS_".

II.12.2 Messages and Services

The following named messages are defined for date-time pickers and can be used in dialog.send_message:

Named Messages for Date-Time Picker				
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful
setformat	f	DTM_SETFORMAT	no	string
getsystime	g	DTM_GETSYSTEMTIME	no	handle to SYSTEMTIME struct
setrange	s	DTM_SETRANGE	yes; details	rgst: handle array of 2 SYSTEMTIME structs

setformat: sets the display formatting string to whatever you pass in lParam.

getsystime: returns GDT_NONE (1) if "none" is selected (only if the control has the "shownone" style); returns GDT_VALID (0) and modifies SYSTEMTIME struct.

setrange: wParam: if includes GDTR_MIN (0x0001), sets the minimum allowable systemtime to rgst[0], otherwise removes minimum; if includes GDTR_MAX (0x0002), sets the maximum allowable systemtime to rgst[1], otherwise removes maximum. See dialogPluginDemo2.powerpro for an example.

You can use set_value to change the date of displayed by a date picker. The value must be in the format yyyymmdd; if the new value is the null string, clears selection.

You can use get_value to get the date of displayed by a date picker, in yyyymmdd format.

There are other messages. See dialog-related_defines.txt. Look for "DTM_".

II.12.3 Colours

In both dialog.define_control and dialog.set_colour, the <*foreground*> parameter, if a number (possible generated by dialog.rgb) or a colour name, it will send a MCSC_TEXT message to the control. Otherwise, the <*foreground*> parameter can be one of the following keywords, with the second <*background*> parameter the colour name or RGB value you wish to set:

word	#define	means
background	MCSC_BACKGROUND	the background colour (between months)
text	MCSC_TEXT	the dates
titlebk	MCSC_TITLEBK	background of the title
titletext	MCSC_TITLETEXT	title text
monthbk	MCSC_MONTHBK	background within the month cal
trailingtext	MCSC_TRAILINGTEXT	the text color of header & trailing days

II.13: The Month-Calendar Control

<control_type> **value:** monthcal

Used in sample script dialogPluginDemo2.powerpro

To use the month-calendar date picker on Windows 95/NT you must have updated comctl32.dll to version 4.70 or later.

Dates accepted for input or produced for output by the control follow the same format as the dates produced by PowerPro keywords and functions like date and formatdate. Dates are represented as eight digit number yyyymmdd, such as 20021028 for October 28, 2002 or 20030101 for January 1, 2003.

Like the Date-Time control but doesn't collapse to an edit box. Seems to need about 110 dialog units width, 90 depth to display nicely.

Unlike the Date-Time control, you can select a range of dates. There are messages you can send to control how many months are displayed, how big a selection is allowed, date range that can be displayed, and so on.

II.13.1 Styles

In addition to styles applying to all controls, there are:

Styles for Month-Calendar Controls			
this style name	**or this letter**	**#define'd symbol**	**means**
daystate	d	MCS_DAYSTATE	Makes the control send MCN_GETDAYSTATE notifications to request information about which days should be displayed in bold. [Not yet supported]
multiselect	m	MCS_MULTISELECT	Allows the user to select a range of dates rather than being limited to a single date. By default, the maximum range is 366 days, which can be changed by sending the "setmaxselcount" message to the control.
weeknumbers	w	MCS_WEEKNUMBERS	Displays week numbers (1-52) to the left of each row of days. Week 1 is defined as the first week that contains at least four days.
notodaycircle	c	MCS_NOTODAYCIRCLE	Prevents the circling of today's date within the control.
notoday	n	MCS_NOTODAY	Prevents the display of today's date at the bottom of the control.

II.13.2 Messages and Services

The following named messages are defined for month-calendar controls and can be used in dialog.send_message:

Named Messages for Month-Calendar Controls				
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful
setrange	s	MCM_SETRANGE	yes; details	rgst: handle array of 2 SYSTEMTIME structs
setdate	d	MCM_SETCURSEL	no; details	SYSTEMTIME struct
getselrange	g	MCM_GETSELRANGE	no	rgst: handle array of 2 SYSTEMTIME structs
setselrange	-	MCM_SETSELRANGE	no	rgst: handle array of 2 SYSTEMTIME structs
setmaxsel	m	MCM_SETMAXSELCOUNT	yes	no

Notes:

setdate: causes an error if control has "multiselect" style. Providing it *doesn't* have that style, you can also use set_value to change the date displayed.

You can use get_value to get the date of displayed by a month-calendar controls, in yyyymmdd format. It will return two dates separated by space if the control has the "multiselect" style.

setmaxsel: if control has "multiselect" style, sets maximum number of days selectable.

getselrange: if control has "multiselect" style, gets range. get_value is simpler to use.

setselrange: if control has "multiselect" style. set_value is easier to use.

There are other messages. See dialog-related_defines.txt. Look for "MCM_".

II.13.3 Colours

In both dialog.define_control and dialog.set_colour, the *<foreground>* parameter, if a number (possible generated by dialog.rgb) or a colour name, it will send a LVM_SETTEXTCOLOR message to the control. Otherwise, the *<foreground>* parameter can be one of the following keywords listed here, with the second *<background>* parameter the colour name or RGB value you wish to set.

II.14: The ActiveX Container Control

<*control_type*> **value:** ActiveX

Used in sample scripts:
dialogPluginDemo3.powerpro, dialogPluginDemo4.powerpro

You can embed an ActiveX object in a control of type “activeX”. To do so, set the <*text_or_var*> of your dialog.define_control service call (or equivalent field in your *<dialog_definition_file>*) to:

- the name of a valid activeX object, e.g. "OWC.Chart" (if you have Office Web Components installed); or "ShockwaveFlash.ShockwaveFlash" (if you have MacroMedia Flash installed; or "MSCAL.Calendar" (for the activeX version of the calendar control). Or

- A valid url, or absolute path to a file of a type that Internet Explorer knows how to handle. That will always include htm and html files, but, with the right plugins installed, might include .pdf, .swf, .mov or .ram files.

  If Internet Explorer options, security tab, miscellaneous node, “Allow scripting of Internet Explorer Webbrowser control” is set off, a url may not work.

Once you’ve created your dialog, you can use dialog.get_hwnd to get a handle for your activeX control, then feed that handle to com.get_object. From then on you can manipulate the com object as usual by its methods, and get and set its properties.

Unfortunately there's no way to respond to user events within the embedded controls: so if e.g. someone fills in a spreadsheet control, you'd have to have a button or other user-triggerable UI feature to allow you to read back control properties when done.

An example:

```
hDlg.define_control(  10,  50, 245, 80, "activeX",    "ax2cal",     "MSCAL.Calendar" )
```

Then, perhaps in response to the user clicking the “OK” button for the dialog:

```
local com_status, objCal
objCal = com.get_object(hDlg.get_hwnd("ax2cal"))
;Value is a property of a calendar object.
win.debug("Date picked was: " ++ objCal.Value)
```

II.15: The Statusbar Control

<*control_type*> **value:** statusbar

Used in sample script dialogPluginDemo2.powerpro

A status bar is a horizontal window that is positioned, by default, at the bottom of a parent window. It is used to display status information defined by the application. If you wish to display more than one type of status information, you can divide the status bar into sections.

If you want a standard statusbar stretched across the bottom of your dialog, use dimensions 0x80000000, 0x80000000 , 10,10. CW_USEDEFAULT = 0x80000000 is defined in dialog-related_defines.txt. You can also use –100 for default x and y positions.

set_value:

> **dialog.set_value(**<*target*>, <*pos*>, <*text_for_part_pos*> [, <*text_for_part_part_pos+1*>])
> ***<dialog_handle>*.set_value**(<*pos*>, <*text_for_part_pos*> [, <*text_for_part_part_pos+1*>])

If the statusbar has multiple parts, <*pos*> can be index of the part (lowest legal value 0 or 1), possibly or'ed with drawing operation (see the settext message for possible drawing operation values). If <*pos*> plus number of text arguments is larger than the number of parts, or you get an error message.

get_value: provide a partno as argument to return the text in that part.

II.15.1 Styles

In addition to styles applying to all controls, and those applying to some common controls, there are:

Styles for Statusbars			
this style name	**or this letter**	**#define'd symbol**	**means**
sizegrip	s	SBS_SIZEGRIP	places a sizing grip at the lower-right corner of the status bar; dialog must have the "thickframe" style for it to work.
tooltips	i	SBT_TOOLTIPS	allows the SB_SETTIPTEXT message to set the tooltip text of part of a statusbar

Styles For Some Common Controls (header controls, toolbars, and statusbars)			
this style name	**or this letter**	**#define'd symbol**	**means**
bottom	b	CCS_BOTTOM	Causes the control to position itself at the bottom of the parent window's client area and sets the width to be the same as the parent window's width. Status windows have this style by default.
nodivider	n	CCS_NODIVIDER	Prevents a two-pixel highlight from being drawn at the top of the control.
nohilite		CCS_NOHILITE	Prevents a one-pixel highlight from being drawn at the top of the control.
nomovey	y	CCS_NOMOVEY	Causes the control to resize and move itself horizontally, but not vertically, in response to a WM_SIZE message. If CCS_NORESIZE is used, this style does not apply. Header windows have this style by default.
noparentalign	p	CCS_NOPARENTALIGN	Prevents the control from automatically moving to the top or bottom of the parent window. Instead, the control keeps its position within the parent window despite changes to the size of the parent. If CCS_TOP or CCS_BOTTOM is also used, the height is adjusted to the default, but the position and width remain unchanged.
noresize	r	CCS_NORESIZE	Prevents the control from using the default width and height when setting its initial size or a new size. Instead, the control uses the width and height specified in the request for creation or sizing.
top	t	CCS_TOP	Causes the control to position itself at the top of the parent window's client area and sets the width to be the same as the parent window's width. Toolbars have this style by default.
vertical	v	CCS_VERT	
left	l	CCS_LEFT	
right	r	CCS_RIGHT	
nomovex	x	CCS_NOMOVEX	

II.15.2 Messages and Services

The following named messages are defined for statusbars and can be used in dialog.send_message:

Named Messages for Statusbars				
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful
setparts	p	SB_SETPARTS	yes; no of parts	handle to array (of integers, widths) details
settext	t	SB_SETTEXT	yes; details	string
settip	i	SB_SETTIPTEXT	yes; details	string
simple	s	SB_SIMPLE	yes; details	no
minheight	h	SB_SETMINHEIGHT	yes; minimum height, in pixels	no
setborders	b	SB_SETBORDERS	no	handle to array of ints, details; I can't get it to work

Notes:

setparts: sets the number of parts in a status bar and gets the coordinate of the right edge of each part. **lParam** is the address of an integer array that has the same number of elements as parts specified by wParam. Each element in the array specifies the position, *in client coordinates*, of the right edge of the corresponding part. If an element is 0x08000000 or –1 (SBPS_STRETCH defined in dialog-related_defines.txt) the right edge for that part extends to the right edge of the window.

Return value: TRUE if successful; FALSE otherwise.

page 215 of 259

settext: sets the text in the given part of a status bar or header window. wParam is the or'ed zero-based index of the part to set and the type of drawing operation (the same values can be used in calls to set_value for statusbars):

operation	value	means
-	0	The text is drawn so that it appears as part of the window--that is, the text appears lower than the plane of the window.
SBT_NOBORDERS	0x0100	The text is drawn without borders; doesn't work unless you use set_value
SBT_POPOUT	0x0200	The text is drawn so that it appears raised, that is, the text appears higher than the plane of the window.
-	255	the status bar is assumed to be a simple window with only one part. lParam is the address of a null-terminated string that specifies the text to set.

Return value: TRUE if successful; FALSE otherwise.

You can also use set_value to change the text of a statusbar or one of its parts.

settip: sets the tooltip text for the wParam'th part of status bar. The status bar must have been created with the SBT_TOOLTIPS style to enable ToolTips. The text you send with this message is displayed when the corresponding pane in the status bar contains text that is truncated due to the size of the pane.

simple: specifies whether a status bar displays simple text or displays all window parts set by a previous SB_SETPARTS message. If **wParam** is TRUE, the status bar displays simple text. If wParam is FALSE, the status bar displays multiple parts.

Return value: FALSE if an error occurs.

setborders: sets the widths of the horizontal and vertical borders of a status bar or header window. These borders determine the spacing between the outer edge of the window and the rectangles within the window that contain text, and the spacing between rectangles. *lParam* is a handle to an array of integers that has three elements. The first element specifies the width of the horizontal border, the second specifies the width of the vertical border, and the third specifies the width of the border between rectangles. If an element is set to -1, the default width for the border is used.

Return value: TRUE if successful; FALSE otherwise.

II.15.3 Colours

The background colour in define_control and dialog.set_colour works, after a fashion (rather ugly, the raised borders of the control and of its subsections is coloured along with the text areas); foreground colour has no effect.

II.16: The Tab Control

<*control_type*> **value:** statusbar

Used in sample script dialogPluginDemo4.powerpro

Use set_value to create tabs within a tab control and choose what show controls appear in which tab.

> **dialog.set_value(**<*dialog_handle*>*,* <*tab_ctrl_id*>, <*tab_no*>, <*text*>*,* <*image_number*>*,* [*,* <*ctrl_id1*> [, <*ctrl_id2*> [,…. [, <*ctrl_idn*> [, <*select*>]]] **)**
> **dialog.set_value(**<*dialog_handle*>*,* <*tab_ctrl_id*>, "height", <*height*>**)**
> **dialog.set_value(**<*dialog_handle*>*,* <*tab_ctrl_id*>, "imagesize", <*x*>*,* <*y*>**)**

You must only invoke set_value *after* a dialog is created or run.

You can also use the aliases for set_value, if you use the **handle.service** syntax, e.g.

> ***<dialog_handle>*.set(**<*tab_ctrl_id*>, <*tab_no*>, <*text*>*,* <*icon number*>*,* [*,* <*ctrl_id1*> [, <*ctrl_id2*> [,…. [, <*ctrl_idn*> [, <*select*>]]] **)**

<*dialog_handle*> Required: A handle returned by dialog.define. See Section 11.1

<*target*> Required: Must be either the *control id* (see Section 11.3) of a tab control.

<*tab_no*> Required (except with "height" and "imagesize" keywords): index (lowest legal value 0 or 1) of a tab you wish to add or modify.

If <*tab_no*> tab already exists, and <*text*> is identical to the caption on the existing tab, <*image_number*>is ignored and <*ctrl_id*>s are added to the existing tab. If <*text*> *isn't* identical to the caption on the existing tab, the existing tab is deleted, together with all its associated controls, and the new one inserted in its place. No idea what happens if <*tab_no*>s specified in a series of called to **set_value** aren't in sequence.

<*text*> Required (except with "height" and "imagesize" keywords):

<*image_number*> Required: index (lowest legal value 0 or 1) of icon previously added to the tab control using set_image. If no image desired or available, use one less than lowest legal value. If the size of images being inserted is greater than tab size (specified with the "height" option), tab size is increased.

<*ctrl_id*>s (at least one) Required: The *control id(s)* (see Section 11.3) of any number of controls which will be considered the exclusive property of the tab being created or modified. The controls with the specified will *control id(s)* be made visible when the <*tab_no*>th is selected, hidden otherwise.

It is your responsibility to make sure a *control id* is associated with only one tab.

The enable and show services affect controls associated with a tab control.

<*select*> Optional: 0 or 1, assumed 0 if absent. The <*tab_no*>th will be the selected tab when the tab control is first shown. Only one tab should have a <*select*> of 1; funny stuff will happen if you do otherwise.

The "height" option changes the height of the tabs on a tab control.

The "imagesize" option changes the dimensions of images stored in the tab control's image list. Invoking this option will erase all images already in the list, so you should call it before any calls to set_image. <*x*>*,* <*y*> are pixels; typical values for both are 16 (normal small icon size) or 36 (normal large icon size). By default image size is 16 x 16.

Tab controls respond to the TCN_SELCHANGE notification sent with a WM_NOTIFY message (and deliver the numeric id of the tab changed to (lowest legal value 0 or 1) as the lParam parameter sent to your <*script_to_run*>.

II.16.1 Styles

In addition to styles applying to all controls, and those applying to some common controls, there are

Styles for Tab Controls			
this style name	**or this letter**	**#define'd symbol**	**means**
buttons	-	TCS_BUTTONS	Creates a tab control that looks and works like a button.
fixedwidth	w	TCS_FIXEDWIDTH	Creates a tab control with a fixed width.
focusonbuttondown	-	TCS_FOCUSONBUTTONDOWN	Specifies that the tab will receive the input focus when the user clicks it. This style is used in conjunction with the TCS_BUTTONS style.
multiline	-	TCS_MULTILINE	Allows multiple rows of tabs in the tab control.
raggedright	-	TCS_RAGGEDRIGHT	Specifies that tabs will not be stretched to fill the row. By default, the tab control will stretch each tab item (text, icon, or combination of text and icon displayed in the tab) equally to fill the tab control.
rightjustify	-	TCS_RIGHTJUSTIFY	Right-justifies the text in the tab control. The text is left-justified by default.
singleline	-	TCS_SINGLELINE	Allows only a single row of tabs in the tab control.
tabs	t	TCS_TABS	Creates tabs that look like notebook dividers, and draws a border around the display area.
scrollopposite	s	TCS_SCROLLOPPOSITE	Unneeded tabs scroll to the opposite side of the control when a tab is selected.

Styles for Tab Controls			
this style name	**or this letter**	**#define'd symbol**	**means**
			Requires common controls v 4.70.
bottom	b	TCS_BOTTOM	tabs go at bottom.
right	r	TCS_RIGHT	tabs go on right; goes with…
vertical	v	TCS_VERTICAL	vertical tabs
multiselect	m	TCS_MULTISELECT	Multiple tabs can be selected by holding down the CTRL key when clicking. Must be used with the TCS_BUTTONS. Requires common controls v 4.70
flatbuttons	f	TCS_FLATBUTTONS	Selected tabs appear as being indented into the background while other tabs appear as being on the same plane as the background. Only affects tab controls with the TCS_BUTTONS style. Requires common controls v 4.70
forceiconleft	-	TCS_FORCEICONLEFT	Icons are aligned with the left edge of each fixed-width tab. This style can only be used with the TCS_FIXEDWIDTH style
forcelabelleft		TCS_FORCELABELLEFT	Left-aligns both the icon and label
hottrack	h	TCS_HOTTRACK	causes whichever tab the mouse is hovering over to be highlighted blue.
focusnever	n	TCS_FOCUSNEVER	Specifies that a tab never receives the input focus

II.16.2 Messages and Services

The following named messages are defined for tab controls and can be used in dialog.send_message:

Named Messages for Tab Controls

this message name	or this letter	#define'd symbol	wParam meaningful	lParam meaningful
setitemsize	z	TCM_SETITEMSIZE	no	integer
removeimage	r	TCM_REMOVEIMAGE	yes	no
setpadding	p	TCM_SETPADDING	no	integer
getrowcount	c	TCM_GETROWCOUNT	no	no
setfocus	-	TCM_SETCURFOCUS	yes	no
getfocus	-	TCM_GETCURFOCUS	no	no
setmintabwidth	m	TCM_SETMINTABWIDTH	no	integer
deselectall	d	TCM_DESELECTALL	yes	no
highlight	h	TCM_HIGHLIGHTITEM	yes	integer

Tab number, where needed, is always in wParam and 0-based.

TCM_SETITEMSIZE (width and height, in pixels) and TCM_SETPADDING (horizontal and vertical padding, in pixels), take x, y as arguments forced into a single integer thusly:

    (x & 0xffff) | ((y & 0xffff) << 16)

TCM_HIGHLIGHTITEM takes lParam: fHighlight, the highlight state to be set. If this value is TRUE, the tab is highlighted; if FALSE, the tab is set to its default state.

TCM_DESELECTALL: wParam specifies the scope of the item deselection. If this parameter is set to FALSE, all tab buttons are reset. If it is set to TRUE, then all tab items except for the one currently selected are reset.

II.16.3 Removing Tabs

Use the clear service to remove all tabs or a single tab from the control; provide the index *<tab_no>* index (lowest legal value 0 or 1) as a second argument if you want to delete a tab; if you do, all controls contained in that tab will be hidden.

II.16.4 Colours

Colours can’t be set for a tab controls , so the *<foreground>* and *<background>* parameters of define_control and calls to set_colour are ignored.

II.17: The Treeview Control

<*control_type*> **value:** treeview

Used in sample scripts:
dialogPluginDemo4.powerpro, dialogPluginDemo5.powerpro

Use set_value or one of its aliases (e.g. *add*) to add or modify nodes of a treeview control.

> **dialog.set_value(**<*dialog_handle*>*,* <*treeview_ctrl_id*>, <*node_id*>, <*insert_after*>,
> <*text*>*,* [<*image_number*>*,* [*,* <*selected_image_number*>]] **)**
> **dialog.set_value(**<*dialog_handle*>*,* <*treeview_ctrl_id*>, <*node_id*>, <*attribute*>,
> <*new_value*>**)**
> **dialog.set_value(**<*dialog_handle*>*,* <*treeview_ctrl_id*>, <*node_id*>, "imagesize", <*x*>*,*
> <*y*>**)**

You must only invoke set_value *after* a dialog is created or run.

set_value returns the HTREEITEM of the item inserted or modified.

You can also use aliases for set_value if you use the **handle.service** syntax, e.g.

> ***<dialog_handle>*.add(**<*treeview_ctrl_id*>, <*node_id*>, <*insert_after*>, <*text*>*,*
> [<*image_number*>*,*
> [*,* <*selected_image_number*>]]] **)**

<*dialog_handle*> Required: A handle returned by dialog.define. See Section 11.1

<*treeview_ctrl_id*> Required: Must be the *control id* (see Section 11.3) of a treeview control.

<*node_id*>: the handle to a treeview item (node) under which to insert or find a node; either one returned by a previous call to set_value or get_value, or 0 to inset or change a node at root level.

page 223 of 259

<*insert_after*> Required: One of these values:

if	#defined symbol	means
first 0 null string	TVI_FIRST	Inserts the node at the beginning of the list.
last	TVI_LAST	Inserts the node at the end of the list.
sort	TVI_SORT	Inserts the node into the list in alphabetical order.

<*attribute*> Required for variant of **set_value** for which it's used: one of:

name	***<new_value>***
image	number in image list
selected	number in image list
text	text

<*text*> Required (except with and "imagesize" keywords): The node's text

<*image_number*> Required, *<selected_image_number>* Optional: index (lowest legal value 0 or 1) of icon previously added to the treeview control using set_image. If no image desired or available, use a number less than lowest legal value..

You can retrieve properties of treeView nodes, or get their properties, using

dialog.get_value(<*dialog_handle*>, <*node_id*>, <*attribute_or_related_node*>**)**

<dialog_handle>, <node_id> Required: as for set_value, see above.

<*attribute_or_related_node*> Required: Can either be either on of the attributes used with set_value; or one of;

name	means
children	the number of children of <*node_id*>)
state	state: one or more of these values, or'ed together (and are defined in dialog-related_defines.txt)

or one of the following <*related_node*> names or the numeric values defined in dialog-related_defines.txt.

name	#defined symbol	name	#defined symbol
root	TVGN_ROOT	nextvisible	TVGN_NEXTVISIBLE
next	TVGN_NEXT	previousvisible	TVGN_PREVIOUSVISIBLE
previous	TVGN_PREVIOUS	selected	TVGN_CARET
parent	TVGN_PARENT	lastvisible	TVGN_LASTVISIBLE
child	TVGN_CHILD		
firstvisible	TVGN_FIRSTVISIBLE		

If you retrieve “state”, you’ll get some combination of bits as defined in the following table.

node state	meaning
TVIS_CUT	The item is marked.
TVIS_DISABLED	The item is disabled and is drawn using the standard disabled style and color.
TVIS_EXPANDED	The item's list of child items is currently expanded.
TVIS_EXPANDEDONCE	The item's list of child items has been expanded at least once.
TVIS_FOCUSED	The item has the focus and is drawn with the standard focus rectangle.
TVIS_SELECTED	The item is selected

II.17.1 Styles

In addition to styles applying to all controls, there are:

Styles for TreeView Controls			
this style name	**or this letter**	**#define'd symbol**	**means**
nobuttons	n	removes default styles TVS_HASBUTTONS, TVS_LINESATROOT and TVS_HASLINES	TVS_HASBUTTONS Displays plus (+) and minus (-) buttons next to parent items to expand or collapse the parent item's list of child items. TVS_HASLINES uses lines to show the hierarchy of items. The lines are drawn to link the parent and child items; TVS_LINESATROOT uses lines to link items at the root of the list view
notalways	-	removes default style TVS_SHOWSELALWAYS	TVS_SHOWSELALWAYS causes a selected item to remain selected when the tree-view control loses focus.
rtlreading	-	TVS_RTLREADING	Causes text to be displayed from right-to-left
notooltips	-	TVS_NOTOOLTIPS	Disables ToolTips
checkboxes	c	TVS__CHECKBOXES	Enables check box controls for items, which is displayed only if an image is associated with the item.
trackselect	t	TVS_TRACKSELECT	Enables hot tracking
singleexpand	e	TVS_SINGLEEXPAND	Causes the item being selected to expand and the item being unselected to collapse upon selection
fullrowselect	f	TVS_FULLROWSELECT	Enables full-row selection; the entire row of the selected item is highlighted, and clicking anywhere on an item's row causes it to be selected. This style cannot be used in conjunction with the TVS_HASLINES style.
noscroll	-	TVS_NOSCROLL	Disables both horizontal and vertical scrolling in the control. The control will not display any scroll bars.

nonevenheight	h	TVS_NONEVENHEIGHT	Sets the height of the items to an odd height with the TVM_SETITEMHEIGHT message. By default, the height of items must be an even value

II.17.2 Messages

The following named messages are defined and can be used in dialog.send_message:

Named Messages for TreeView Controls

this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful	**description**
expand	x	TVM_EXPAND	yes; details	handle to HTREEITEM	expands or collapses the list of child items associated with the specified parent item
getcount	c	TVM_GETCOUNT	no	no	returns the number of items in the tree view window.
getindent	-	TVM_GETINDENT	no	no	gets no pixels child items are indented relative to parent items
setindent	i	TVM_SETINDENT	indent, pixels	no	sets the amount of indentation for a child item
selectitem	s	TVM_SELECTITEM	yes; TVN_ const	handle to HTREEITEM	selects the given tree view item, scrolls it into view, and redraws it; see dialog-related_defines.txt for TVN_ values
visiblecount	v	TVM_GETVISIBLECOUNT	no	no	gets count of items that will fit in the control's client window
sortchildren	o	TVM_SORTCHILDREN	no	handle to HTREEITEM	sorts the child items of the given parent item
ensurevisible	v	TVM_ENSUREVISIBLE	no	handle to HTREEITEM	ensures that the tree view item is visible, and expands the parent item or scrolls the tree view window if necessary
setinsertmark	k	TVM_SETINSERTMARK	0/1, 1 = after	handle to	sets the insertion mark; null handle removes it

				HTREEITEM	
setitemheight	h	TVM_SETITEMHEIGHT	iItemHeight	no	sets item height
setscrolltime	t	TVM_SETSCROLLTIME	msec	no	set maximum scroll time
getitemstate	-	TVM_GETITEMSTATE	HTREEITEM	mask	compose mask from TVIS_ values; see dialog-related_defines.txt:

The following TVM_EXPAND wParam values are defined in dialog-related_defines.txt:

TVM_EXPAND codes	**value**	
TVE_COLLAPSE	0x0001	Collapses the list.
TVE_COLLAPSERESET	0x8000	Collapses the list and removes the child items. Note that TVE_COLLAPSE must also be specified.
TVE_EXPAND	0x0002	Expands the list.
TVE_EXPANDPARTIAL	0x4000	Partially expands the list. In this state the child items are visible and the parent item's plus sign (+), indicating that it can be expanded, is displayed. This flag must be used in combination with the TVE_EXPAND flag.
TVE_TOGGLE	0x0003	Collapses the list if it is currently expanded or expands it if it is currently collapsed.

II.17.3 Colours

In both dialog.define_control and dialog.set_colour, the <*foreground*> parameter, if a number (possible generated by dialog.rgb) or a colour name, it will send a TVM_SETTEXTCOLOR message to the control. Otherwise, the <*foreground*> parameter can be one of the following keywords, with the second <*background*> parameter the colour name or RGB value you wish to set:

word	**#define**	**means**
background	TVM_SETBKCOLOR	the background colour
text	TVM_SETTEXTCOLOR	text colour
insertmark	TVM_SETINSERTMARKCOLOR	insert mark colour
line	TVM_SETLINECOLOR	line colour

You can use the same colour parameters in define_control but "text" doesn’t seem to do what it's meant to.

II.17.4 Removing Nodes

Use the clear service to remove all nodes or a single node from the control. Provide the <*node_id*> as a second argument if you want to delete a specific node.

II.18: The Rich Edit Control

<*control_type*> **value:** richedit

Used in sample script regexDialog.powerpro (via associated *dialog_definition_file*)

The dialog plugin supports two other kinds of edit controls (plain edit and scintilla). See section II.18.4 "Which Type of Edit Control?" comparing them..

The standard rich edit control doesn't have a context menu; I've added one to emulate that shown in plain vanilla editboxes or scintilla controls.

Use get_value to get text or the font description for a richedit control.

In addition to the usual values of get_value <*property*>s, you can also use the <property> "selectlen" to return the length of the current selection and "selected" or "select" to return the selection itself.

Use set_value or one of its aliases (e.g. *set, modify*) to change the text of a rich edit control.

> **set_value(**<*dialog_handle*>*,* <*richedit_ctrl_id*>, <*value*> [, <*property*>
> [, <*codepage*>]]**)**

> **set_value(**<*dialog_handle*>*,* <*richedit_ctrl_id*>, <*value*> , *"rtf"***)**

Rich edit controls can now be assigned a <*right_click_command*> with define, define_control or set_response, overriding the default context menu the plugin supplies. (But a "right" <*mouse_event*> with a keyboard modifier like "ctrl" *won't* override it; nor will any other mouse event, e.g. "middle").

<*dialog_handle*> Required: A handle returned by dialog.define. See Section 11.1

<*ricedit_ctrl_id*> Required: Must be the *control id* (see Section 11.3) of a rich edit control.

<*value*> Required: The new text. MSDN says "This text is an ANSI string, unless the code page is 1200 (Unicode), in which case it's a Unicode string. If /lParam/ starts with a valid RTF ASCII sequence, for example, {\rtf or {urtf, the text is read in using the RTF reader"

"rtf": Use this if you're providing an RTF string in <*value*>.

<property> Optional: The MSDN docs say "any reasonable combination of the following flags":

#define	valu e	means
ST_DEFAULT	0	Deletes the undo stack, discards rich-text formatting, replaces all text.
ST_KEEPUNDO	1	Keeps the undo stack.
ST_SELECTION	2	Replaces selection and keeps rich-text formatting.
ST_NEWCHARS	4	

<codepage> Optional: MSDN says "The code page used to translate the text to Unicode. If codepage is 1200 (Unicode code page), no translation is done. If codepage is CP_ACP, the system code page is used." If omitted, the default for the control is used.

Use get_value or one of its aliases (e.g. *set, modify*) to fetch the text of a rich edit control.

> **get_value(***<dialog_handle>, <richedit_ctrl_id>* [, *"rtf"*]**)**
> *<richedit_ctrl_handle>***.get(**[*"rtf"*]**)**
>
> **get_value(***<dialog_handle>, <richedit_ctrl_id>,* "select"**)**
> *<richedit_ctrl_handle>***.get(**"select"**)**

With the "rtf" argument you'll get RTF back, with any formatting included.

With the "select" argument (or any argument beginning "select"), you'll get back currently selected text.

page 232 of 259

II.18.1 Styles

In addition to styles applying to all controls, you can use the styles that apply to plain edit controls, and also :

Styles for Rich Edit Controls			
this style name	**or this letter**	**#define'd symbol**	**means**
emulatesysedit	-	SES_EMULATESYSEDIT	When this bit is on, rich edit attempts to emulate the system edit control
beeponmaxtext	-	SES_BEEPONMAXTEXT	Rich Edit will call the system beeper if the user attempts to enter more than the maximum characters
extendbackcolor	-	SES_EXTENDBACKCOLOR	Extends the background color all the way to the edges of the client rectangle
usecrlf	-	SES_USECRLF	XP SP1: Turns on Text Services Framework (TSF) support
uppercase	-	SES_UPPERCASE	Converts all input characters to uppercase
lowercase	-	SES_LOWERCASE	Converts all input characters to lowercase
scrollonkillfocus	-	SES_SCROLLONKILLFOCUS	When KillFocus occurs, scroll to the beginning of the text
xltcrcrlftocr	-	SES_XLTCRCRLFTOCR	Turns on translation of CRCRLFs to CRs. When this bit is on and a file is read in, all instances of CRCRLF will be converted to hard CRs internally. This will affect the text wrapping. Note that if such a file is saved as plain text, the CRs will be replaced by CRLFs. This is the .txt standard for plain text.
draftmode	-	SES_DRAFTMODE	XP SP1: Use draftmode fonts to display text. Draft mode is an accessibility option where the control displays the text with a single font; the font is determined by the system setting for the font used in message boxes. For example, accessible users may read text easier if it is uniform, rather than a mix of fonts and styles
hidegridlines	-	SES_HIDEGRIDLINES	XP SP1: If the table gridlines width is zero, gridlines are not displayed. This is equivalent to the hide gridlines feature in Microsoft Word's table menu

II.18.2 Messages

Most of the messages applicable to edit controls also apply to rich edit controls. Consult MSDN concerning different usage for plain and rich edit controls.

The following named messages are defined and can be used in dialog.send_message. Without copying enormous chunks of the relevant part of MSDN site, I can't provide enough information here to tell you everything you need to use most messages; look them up.

.

Messages Applicable to Rich edit controls					
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful	**description**
posfromchar	-	EM_POSFROMCHAR	POINTL*	integer (chr ndx)	retrieves the client area coordinates of a specified character
charfrompos	-	EM_CHARFROMPOS	no	POINTL*	Retrieves information about the character closest to a specified point in the client area
linelength		EM_LINELENGTH	no	yes	shows or hides one of the scroll bars in the Text Host window
canpaste	-	EM_CANPASTE	yes	no	determines whether a rich edit control can paste a specified clipboard format
pastespecial	-	EM_PASTESPECIAL	yes	REPASTESPECIAL*	pastes a specific clipboard format in a rich edit control
exlinefromchar	-	EM_EXLINEFROMCHAR	no	0 based idx char	determines which line contains specified character
exgetsel	-	EM_EXGETSEL	no	CHARRANGE*	retrieves the starting and ending character positions of the selection
exsetsel	-	EM_EXSETSEL	no	CHARRANGE*	selects a range of characters and/or COM

Messages Applicable to Rich edit controls					
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful	**description**
					objects
gettextrange	-	EM_GETTEXTRANGE	no	TEXTRANGE*	retrieves a specified range of characters
hideselection	-	EM_HIDESELECTION	0/1	no	hides or shows the selection
selectiontype	-	EM_SELECTIONTYPE	no	no	determines the selection type
gettextex	-	EM_GETTEXTEX	SETTEXTEX*	string	allows you to get all of the text from the rich edit control in any particular code base you want
getseltext	-	EM_GETSELTEXT	no	string (out)	retrieves the currently selected text
gettextlengthex	-	EM_GETTEXTLENGTHEX	GETTEXTLENGTHEX*	no	calculates text length in various ways
findtext	-	EM_FINDTEXT	yes	FINDTEXT*	finds text
findtextex	-	EM_FINDTEXTEX	yes	FINDTEXTEX*	finds text
findwordbreak	-	EM_FINDWORDBREAK	yes	yes (char ndx)	finds the next word break before or after the specified character position or retrieves information about the character at that position
setrect		EM_SETRECT	0/1; absolute or relative coordinates	RECT*	set the formatting rectangle of a multiple-line edit control; causes text to be redrawn
setrectnp	-	EM_SETRECTNP	as above	RECT*	as above, but no redraw
requestresize	-	EM_REQUESTRESIZE	no	no	forces a rich edit control to send an EN_REQUESTRESIZE notification message to its parent window

Messages Applicable to Rich edit controls					
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful	**description**
setcharformat	-	EM_SETCHARFORMAT	yes	CHARFORMAT*	sets character formatting
setparaformat	-	EM_SETPARAFORMAT	no	PARAFORMAT* or PARAFORMAT2*	sets the paragraph formatting for the current selection
seteventmask	-	EM_SETEVENTMASK	no	integer (flags)	sets the event mask, which notification messages the control sends to its parent window
setbkgndcolor	-	EM_SETBKGNDCOLOR	yes (0/1)	COLORREF*	sets the background color
settextmode	-	EM_SETTEXTMODE	yes	no	set the text mode or undo level of a rich edit control. The message fails if the control contains any text
setoptions	-	EM_SETOPTIONS	int; an ECOOP_ const	int; an ECO_ const	sets options, equivalent to some control styles
redo	-	EM_REDO	no	no	redo the next action in the control's redo queue
canredo	-	EM_CANREDO	no	no	determines whether there are any actions in the control redo queue
setundolimit	-	EM_SETUNDOLIMIT	limit	no	set the maximum number of actions that can stored in the undo queue.
getundoname	-	EM_GETUNDONAME	no	no	Rich Edit 2.0 and later: retrieve the type of the next undo action, if any
getredoname	-	EM_GETREDONAME	no	no	retrieve the type of the next action, if any, in the control's redo queue

Messages Applicable to Rich edit controls

this message name	or this letter	#define'd symbol	wParam meaningful	lParam meaningful	description
stopgrouptyping	-	EM_STOPGROUPTYPING	no	no	stop the control from collecting additional typing actions into the current undo action. The control stores the next typing action, if any, into a new action in the undo queue
autourldetect	-	EM_AUTOURLDETECT	0/1	no	enables or disables automatic detection of URLs

II.18.3 Notifications

Rich edit controls only send out notifications if a mask is set to tell it so. I've added code to automatically take care of that for the following notification codes:

- EN_CHANGE
- EN_CORRECTTEXT
- EN_DRAGDROPDONE
- EN_DROPFILES
- EN_MSGFILTER
- EN_LINK
- EN_MSGFILTER
- EN_OBJECTPOSITIONS
- EN_PROTECTED
- EN_REQUESTRESIZE
- EN_VSCROLL
- EN_VSCROLL
- EN_MSGFILTER
- EN_SELCHANGE
- EN_UPDATE

As usual you'll fine the values for all the above symbols in dialog-related_defines.txt.

II.18.3 Colours

<foreground> and *<background>* parameters of define_control and set_colour work, as does the *<mouse_state>* argument of set_colour. The latter may not be ideal if you've formatted text in foreground or background colours; when the control enters your defined *<mouse_state>* foreground or background colour differences will disappear, and won't reappear when returned to normal state.

II.18.4 Which Type of Edit Control?

The dialog plugin supports no less than three types of edit controls. Talk about overkill.

If all you want to do is display plain text, consider just using a static.

If you want to get text from the dialog user, you need some kind of edit box.

If all you want to do is input plain text, use the **simple edit control.**

Otherwise, your tradeoffs:

type of control:	**plain edit**	**rich edit**	**scintilla edit**
requires additional download?	no	yes, usually dll plugin	yes; SciLexer.dll
can text be formatted?	no	yes	yes
context menu (select all, copy, etc)?	yes	yes	yes
can make white space visible?	no	no	yes
support zooming	no	no	yes
brace highlighting and matching	no	no	yes
wrap mode control	with difficulty	with difficulty	yes
undo/redo support	no	yes	yes
line end convention control	no	no	yes
input from and output to rich text format	no	yes	no
quality of available documentation	ok	terrible	excellent
scripter's interface	simple	terrible	simple
fully compatible with **simple edit control**	duh	no	no

I much prefer the scintilla interface over the rich edit one. GUI programmers moan about the Microsoft richedit API, for good reason. It's been through many versions, which don't necessarily improve with maturity. To do anything sophisticated you'll almost certainly need structs, which means getting your head around the dll plugin.

The main limitation I've found so far is there's no easy way to get a usable representation of formatted text in or out of a scintilla control (analogous to RTF for rich edits).

There a lot of scintilla features I haven't tried; macro recording, print control, autocmpletion, lexer control, auto scrolling, etc.

Wrap mode: for simple and rich edit controls, you can fake you can set the hscroll and vscroll and/or multiline for a control, but you can't change those styles at run time; so to get the appearance of turning wrap on and off, you have to make two controls, apply different styles to each, hide one, then when user desires to switch wrap mode, switch controls. Believe me, it's tedious; I did it in **regex\regexDialog.powerpro**.

Scripter's interface: To do anything amusing with any kind of edit control (beyond setting and getting its text, for which you have get_value and set_value), mostly you have to send messages (so learn about the send_message service**).**

The rich edit interface has grown like topsy, and shows it. Documentation on how it works can be hard to find. Many actions require constructing a struct, which means you got to use the dll plugin. The scintilla interface is, by comparison, very clean: (almost) everything is done with messages that take simple numeric or string arguments.

II.19: The Scintilla Control

<*control_type*> **value:** scintilla

Used in sample scripts: regexDialogScintilla.powerpro (via associated *dialog_definition_file*)

The dialog plugin supports two other kinds of edit controls (plain edit rich edit). See section II.18.4 "Which Type of Edit Control?" comparing them..

To get scintilla controls to work, you need SciLexer.dll in your PowerPro plugins or installation folders, or on your path; it's obtainable from http://www.scintilla.org/, the file wscite*.zip.

An edit control built on the SciLexer engine can do (almost) everything a rich edit control can do; it can also be persuaded to do automatic lexical highlighting; show and hide white space; adjust translucency using an alpha value; brace highlighting; zooming; undo and redo; set margins and tabs; assign markers to text; and lots of other goodies. It's all done with messages; see below.

Use get_value to get text or the font description for an scintilla control

In addition to the usual values of get_value <*property*>s, you can also use the <property> "selectlen" to return the length of the current selection and "selected" or "select" to return the selection itself.

Use set_value or one of its aliases (e.g. *set, modify*) to change the text of an edit control.

With set_font, or get_value with a <*property*> of "font", the strikethrough option is ignored, as it doesn't seem available in scintilla's messages for style modification.

Scintilla controls can now be assigned a <*right_click_command*> with define, define_control or set_response, overriding the built-in context menu. (But a "right" <*mouse_event*> with a keyboard modifier like "ctrl" won't override it; nor will any other mouse event, e.g. "middle").

II.19.1 Styles

In addition to styles applying to all controls, you can use the styles that apply to richedit controls.

II.19.2 Messages

All the named messages applicable to rich edit controls also apply to scintilla controls; there's a list of exceptions at the end of scintilla-related_defines.txt. To quote the url below:

> "Scintilla tries to be a superset of the standard windows Edit and RichEdit controls wherever that makes sense.   As it is not intended for use in a word

> processor, some edit messages can not be sensibly handled. Unsupported messages have no effect….[some listed] messages are currently supported to emulate existing Windows controls, but they will be removed in future versions of Scintilla. If you use these messages you should replace them with the Scintilla equivalent."

There are a ton of messages specifically for scintilla controls; I haven't bothered doing named messages. See scintilla-related_defines.txt for a list; see http://scintilla.sourceforge.net/ScintillaDoc.html on how to use them.

II.19.3 Colours

<foreground> and *<background>* parameters of define_control and set_colour work by sending SCI_STYLESETFORE or SCI_STYLESETBACK messages for all styles of the relevant control.

*<mouse_state>*s of set_colour work. Again, it's done by changing all styles. Once a control leaves a particular *<mouse_state>,* all styles revert to their previous colours. Even so *<mouse_state>*s may not be ideal if you've formatted text in foreground or background colours; when the control enters your defined *<mouse_state>* foreground or background colour differences will temporarily disappear.

II.20: The Listview Control

<*control_type*> **value:** listview

Used in sample scripts:
dialogPluginDemo5.powerpro, dialogPluginDemo6.powerpro,
dialogPluginDemoNonNative.powerpro

Listview controls are about the most complex in the Microsoft set of dialog components. They have numerous variants, roughly corresponding to the various views you can have in an Explorer window. Listviews have rows and columns; invididual items can have text and/or icons.

So far I've implemented what you need to create and alter a listview (add and alter rows, columns and items; remove rows and columns); but not what you might need to retrieve alternations a user might make to one (changing item values, reordering columns…). That will come (in the form of the get_value service and its aliases).

page 243 of 259

Use set_value or one of its aliases (e.g. *add*) to add or modify nodes of a listview control.

> **dialog.set_value(***<target>*, "col", *<column_no>*, *<width>* [*, <format>*]] **)**
> **dialog.set_value(***<target>*, "row", *<row_no>*, *<starting_subitem_no>,*
> *<item_text>* [*, <item_text>, …*]]**)**
> **dialog.set_value(***<target>*, "item", *<attribute>*, *<new_value>***)**
> **dialog.set_value(***<target>*, "select", *<row_no_beg>* [*, <row_no_end>*]**)**
> **dialog.set_value(***<target>*, "focus"*, <row_no>***)**
> **dialog.set_value(***<target>*, "cut"*, <row_no>***)**

You must only invoke **set_value** taking a *<dialog_handle>* and a *<listview_ctrl_id> as <target> after* a dialog is created or run.

You can use aliases for **set value** if you use the **handle.service** syntax, e.g.

> ***<dialog_handle>*.add(***<listview_ctrl_id>*, "row", *<row_no>*,
> *<starting_subitem_no>, <item_text>* **)**

Using a *<window_ handle_ to_ remote_ control>* as a *<target>* can be dangerous. It's okay if you just want change selection or focus; but I wouldn't try to alter the view, or contents of a listview owned by another process. That process will believe its listview has certain contents or is presented in a certain way; you will change that; and goodness knows what happens next.

Here are the variants:

set_value(*<target>*, "col", *<column_no>*, *<width>* [*, <format>*]]**)**

Add new column or modify existing one; probably bad idea for remote control. *<column_no>* lowest legal value is 0 or 1.

<format>: only applies when third parameter is "col": specifies the alignment of the column heading and the subitem text in the column; one of

keyword	equivalent #define	text is
centre	LVCFMT_CENTER	centered.
left	LVCFMT_LEFT	left-aligned
right	LVCFMT_RIGHT	right-aligned

page 244 of 259

set_value(<target>, "row", *<row_no>*, *<starting_col_no>, <item_text>* [, *<item_text>, …*]]**)**

Add new row or modify existing one; probably bad idea for remote control.

<row_no>, *<starting_col_no>* Required: lowest legal values 0 or 1. If *<starting_col_no>* is greater than the lowest legal value (therefore not setting the row's label), the row *<row_no>* must already exist and label already set by a previous call to set_value with "row" or "item"; otherwise you get an error message.

<item_text>: text to use to initialise item; one for each column ("subitem") you want to set.

set_value(<target>, "item", *<row_no>*, *<col_no>, <item_text>* [, *<image_no>*]**)**

Changes an *attribute* of item; probably bad idea for remote control.

<row_no>, *<col_no>* Required: lowest legal values 0 or 1. If *<col_no>* is greater than the lowest legal value (therefore not setting the row's label), the row *<row_no>* must already exist and label already set by a previous call to set_value with "row" or "item"; otherwise you get an error message.

<item_text> Required: Text of subitem you want to set.

<image_no> Optional: lowest legal values 0 or 1.

set_value(<target>, "select", *<row_no_beg>* [, *<row_no_end>*]**)**

Selects row(s). *<row_no_beg>* and *<row_no_end>* lowest legal values 0 or 1.

set_value(<target>, "deselect", *<row_no_beg>* [, *<row_no_end>*]**)**

Unselects row(s). *<row_no_beg>* and *<row_no_end>* lowest legal values 0 or 1.

set_value(<target>, "focus", *<row_no>***)**

Set focus to specified row; lowest legal value 0 or 1.

set_value(<target>, "cut", *<row_no>***)**

Set row *<row_no>* (lowest legal value 0 or 1) to "cut" state. Bet it's not a good idea for a remote control

set_value(<target>, "view",*<view_type>***)**

Probably bad idea for remote control

set_value(<target>, "selectAll"**)**

Select all rows.

set_value(*<target>*, "selectInvert"**)**

Inverts current row selection.

set_value(*<target>*, "selectClear"**)**

Unselects everything.

set_value(*<target>*, "view", *<viewType>***)**

Changes listview visual form:

LVM_SETVIEW

largeicons large	LV_VIEW_ICON
details report	LV_VIEW_DETAILS
list	LV_VIEW_LIST
smallicons smallicon	LV_VIEW_SMALLICON

Minimum operating systems	Windows XP

In the Explorer Folder Options - General - Tasks, if you have the "Show common tasks in folders" option selected instead of "Use Windows classic folders" the function will fail.

ControlListView 'ViewChange' fails:
http://www.autoitscript.com/forum/index.php?showtopic=11666&hl=

About List-View Controls
http://msdn2.microsoft.com/en-us/library/ms670558.aspx - ListView_Styles_and_Views

Use get_value or one of its aliases to retrieve information from or about a listView control.

> **dialog.get_value(***<target>***)**
> **dialog.get_value(***<target>, 0, <column_no>,* [<*vec*>] **)**
> **dialog.get_value(***<target>, <row_no>,* [[<*vec*>] **)**
> **dialog.get_value(***<target>, <row_no>* , *<column_no>***)**
> **dialog.get_value(***<target>,* "rows" | "rowcount"**)**
> **dialog.get_value(***<target>,* "cols" | "colCount"**)**
> **dialog.get_value(***<target>,* "selected" [, <*vec*>] [, <*max_rows*>
> [, <*column_no*>]]**)**
> **dialog.get_value(***<target>,* "focused" [, <*vec*>] [, <*column_no*>]**)**
> **dialog.get_value(***<target>,* "selectedIndex"**)**
> **dialog.get_value(***<target>,* "selectedallindex"**)**
> **dialog.get_value(***<target>,* "selectedCount"**)**
> **dialog.get_value(***<target>,* "focusedRowNo"**)**
> **dialog.get_value(***<target>, <state>, <row_no>* [, <*column_no*>]**)**

You must only invoke **get_value** taking a <*dialog_handle*> and a <*listview_ctrl_id*> *after* a dialog is created or run.

You can also use aliases for **set value** if you use the **handle.service** syntax, e.g.

> **<*dialog_handle*>.get(**<*listview_ctrl_id*>, <*row_no*>, <*subitem_no*>**)**

Here are the variants:

get_value(*<target>***)**
get_value(*<target>,* "rows" | "rowcount"**)**

Returns number of rows.

get_value(*<target>,* "cols" | "colCount"**)**

Returns number of columns.

get_value(*<target>,* "selectedCount"**)**

Returns number of selected rows

get_value(*<target>,* "focusedRowNo"**)**

returns index of row with focus (based at 0 or 1)

get_value(*<target>,* "selectedIndex" [, <*howMany*>]**)**

Returns indices of selected rows (based at 0 or 1), space separated. If <*howMany*> is absent, or less than base, returns all of them.

get_value(*<target>, <row_no>* , *<column_no>***)**

Returns one cell (subitem). *<row_no>* and *<column_no>* bases lowest legal value 0 or 1.

get_value(*<target>, <vec>*]**)**

Returns the entire contents of the listview..

<vec>] := ["vec" | *<handle_to_vec>*]. If you specify “vec” a two-dimensional vector of exactly the correct size will be created and it’s handle returned. If you pass in *<handle_to_vec>* that vector will be filled and *<handle_to_vec>* returned.

get_value(*<target>, <row_no>,* [*<vec>*] **)**

Returns the contents of a row. *<row_no>* (lowest legal value 0 or 1).

Without *<vec>* returns a string, items from row separated are separated by tab (\t).

If you specify “vec”, a *one*-dimensional vector of exactly the correct size will be created.

get_value(*<target>, 0, <column_no>,* [*<vec>*] **)**

Returns the contents of a column; *<column_no>* (lowest legal value 0 or 1).

Without *<vec>* returns a string; items from the column are separated by newline (\n).

If you specify “vec”, a *one*-dimensional vector of exactly the correct size will be created.

get_value(*<target>,* "selected" [, *<vec>*] [, *<max_rows>* [, *<column_no>*]]**)**

Returns the contents of selected rows. *<column_no>* (lowest legal value 0 or 1); if it’s omitted, entire rows are returned.

Without *<vec>* returns a string; items in each row are separated are separated by tab (\t); rows are separated by newline (\n).

If you specify “vec”, a *two*-dimensional vector of exactly the correct size will be created.

get_value(*<target>,* "focused" [, *<vec>*] [, *<column_no>*]**)**

Returns the contents of row with focus. *<column_no>* (lowest legal value 0 or 1); if it's omitted, entire row is returned.

Without *<vec>* returns a string, items from row separated are separated by tab (\t).

If you specify "vec", a *one*-dimensional vector of exactly the correct size will be created.

get_value(*<target>, <state>, <row_no>* [, *<column_no>*]**)**

<state>: can be one of:

keyword	equivalent #define
isselected	LVIS_SELECTED
isfocused	LVIS_FOCUSED
iscut	LVIS_CUT

II.20.1 Styles

In addition to styles applying to all controls, there are:

Styles for ListView Controls			
this style name	**or this letter**	**#define'd symbol**	**means**
alignleft	l	LVS_ALIGNLEFT	items are left-aligned in icon and small icon view
aligntop	t	LVS_ALIGNTOP	items are aligned with the top of the control in icon and small icon view
autoarrange	a	LVS_AUTOARRANGE	icons are automatically kept arranged in icon view and small icon view.
editlabels	e	LVS_EDITLABELS	Allows item text to be edited in place. The parent window must process the LVN_ENDLABELEDIT notification message.
icon	i	LVS_ICON	Specifies icon view
list	l	LVS_LIST	Specifies list view
nocolumnheader	-	LVS_NOCOLUMNHEADER	Specifies that a column header is not displayed in report view. By default, columns have headers in report view.
nolabelwrap	-	LVS_NOLABELWRAP	Displays item text on a single line in icon view. By default, item text can wrap in icon view.
noscroll	-	LVS_NOSCROLL	Disables scrolling. All items must be within the client area
nosortheader	-	LVS_NOSORTHEADER	Colume headers do not work like buttons. This style is useful if clicking a column header in report view does not carry out an action, such as sorting
report	r	LVS_REPORT	Specifies report view
showselalways	-	LVS_SHOWSELALWAYS	Always show the selection, if any, even if the control does not have the focus
singlesel	-	LVS_SINGLESEL	Allows only one item at a time to be selected. By default, multiple items can be selected
smallicon	-	LVS_SMALLICON	Specifies small icon view
sortascending	-	LVS_SORTASCENDING	Sorts items based on item text in ascending order
sortdescending	-	LVS_SORTDESCENDING	Sorts items based on item text in descending order
fullrowselect	0	LVS_EX_FULLROWSELECT	v 4.70. When an item is selected, the item and all

page 251 of 259

Styles for ListView Controls			
this style name	**or this letter**	**#define'd symbol**	**means**
			its subitems are highlighted; only in conjunction with the LVS_REPORT
gridlines	g	LVS_EX_GRIDLINES	v 4.70. Displays gridlines around items and subitems; only in conjunction with the LVS_REPORT style
flatsb	f	LVS_EX_FLATSB	Enables flat scroll bars in the list view
headerdragdrop	h	LVS_EX_HEADERDRAGDROP	v. 4.70. Enables drag-and-drop reordering of columns in a list-view control; only in conjunction with LVS_REPORT.
labeltip	0	LVS_EX_LABELTIP	v 5.80. If a partially hidden label in any list view mode lacks ToolTip text, the list-view control will unfold the label. If this style is not set, the list-view control will unfold partly hidden labels only for the large icon mode.
multiworkareas	m	LVS_EX_MULTIWORKAREAS	v 4.71. If control has LVS_AUTOARRANGE style, the control will not autoarrange its icons until one or more work areas are defined. This style must be set before any work areas are defined and any items have been added to the control.
oneclickactivate	o	LVS_EX_ONECLICKACTIVATE	v 4.70. The list-view control sends an LVN_ITEMACTIVATE notification message to the parent window when the user clicks an item; enables hot tracking.
twoclickactivate	0	LVS_EX_TWOCLICKACTIVATE	v 4.70. The list-view control sends an LVN_ITEMACTIVATE notification message to the parent window when the user double-clicks an item; enables hot tracking.
regional	0	LVS_EX_REGIONAL	v 4.71. Sets the list view window region to include only the item icons and text using SetWindowRgn; only in conjunction with LVS_ICON
simpleselect	s	LVS_EX_SIMPLESELECT	v 6.00. In icon view, moves the state image of the control to the top right of the large icon rendering. In views other than icon view there is no change. When the user changes the state by using the space bar, all selected items cycle over, not the item with the focus.
subitemimages	s	LVS_EX_SUBITEMIMAGES	v 4.70. Allows images to be displayed for subitems; only in conjunction with LVS_REPORT
trackselect	0	LVS_EX_TRACKSELECT	v 4.70. Enables hot-track selection in a list-view control. Hot track selection means that an item is

page 252 of 259

Styles for ListView Controls			
this style name	**or this letter**	**#define'd symbol**	**means**
			automatically selected when the cursor remains over the item for a certain period of time
underlinecold	c	LVS_EX_UNDERLINECOLD	v 4.71. Causes those non-hot items that may be activated to be displayed with underlined text; requires LVS_EX_TWOCLICKACTIVATE
underlinehot	h	LVS_EX_UNDERLINEHOT	4.71. Causes those hot items that may be activated to be displayed with underlined text; requires LVS_EX_ONECLICKACTIVATE or LVS_EX_TWOCLICKACTIVATE
borderselect		LVS_EX_BORDERSELECT	v 4.71. Changes border color when an item is selected, instead of highlighting the item
checkboxes		LVS_EX_CHECKBOXES	v 4.70. Enables check box controls for items; the control creates and sets a state image list with two images State image 1 is the unchecked box, state image 2 the checked box. Setting the state image to zero removes the check box

II.20.2 Messages

The following named messages are defined and can be used in dialog.send_message:

Messages Applicable to List View Controls					
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful	
arrange	-	LVM_ARRANGE	LVA_ const	no	arranges items in icon view; see LVA_ constants in dialog-related_defines.txt
deleteallitems	-	LVM_DELETEALLITEMS	no	no	removes all items (rows) from a list view window
deletecolumn	-	LVM_DELETECOLUMN	int (col no)	no	deletes a column; all subitem will be removed
deleteitem	-	LVM_DELETEITEM	int (row no)	no	index of the row to delete
editlabel	-	LVM_EDITLABEL	integer (item no)	no	begins in-place editing of the specified list view item's text To cancel editing, set iItem to -1. The control must have the focus before you send this message
ensurevisible	-	LVM_ENSUREVISIBLE	int (row no)	partial ok (0/1)	ensures that a row is entirely or at least partially visible
finditem	-	LVM_FINDITEM	yes	LV_FINDINFO*	retrieves the text of a listview subitem
getbkcolor	-	LVM_GETBKCOLOR	no	no	gets the background color of the list view window
getcolumn	-	LVM_GETCOLUMN	yes	LV_COLUMN*	gets the attributes of a list view column
getcolumnwidth	-	LVM_GETCOLUMNWIDTH	int (col no)	no	gets the width of a column in list view or report view
getcountperpage	-	LVM_GETCOUNTPERPAGE	no	no	calculates the number of items that can fit vertically in the visible area of a view control in list view or report view.
geteditcontrol	-	LVM_GETEDITCONTROL	no	no	gets the hwnd of the edit window used to edit the item

Messages Applicable to List View Controls

this message name	or this letter	#define'd symbol	wParam meaningful	lParam meaningful	
					text in place
getisearchstring	-	LVM_GETISEARCHSTRING	no	string	retrieves the incremental search string
getitem	-	LVM_GETITEM	no	LV_ITEM*	gets a list view item 's (row's) attributes
getitemcount	-	LVM_GETITEMCOUNT	no	no	gets the number of all the items (rows) in a listview
getitemposition	-	LVM_GETITEMPOSITION	int (item no)	POINT*	gets the position of a list view item in standard icon and small icon views
getitemrect	-	LVM_GETITEMRECT	int (item no)	RECT* (int code for type)	gets the bounding rectangle for an item in the current view; LVIR_ constants to set type
getitemspacing	-	LVM_GETITEMSPACING	bool	no	spacing between items; wParam set to TRUE for small icon view, or to FALSE for icon view.
getitemstate	-	LVM_GETITEMSTATE	yes	integer	retrieves the state of an item (row)
getitemtext	-	LVM_GETITEMTEXT	yes	LV_ITEM*	retrieves the text of a subitem (cell)
getnextitem	-	LVM_GETNEXTITEM	int (item to start at)	int (search flags; LVNI_ consts)	searches for the next list view item starting from a specified item. If an item does not have all of the specified state flags set, the search continues with the next item
getorigin	-	LVM_GETORIGIN	no	POINT*	gets the list view origin
getselectedcount	-	LVM_GETSELECTEDCOUNT	no	no	retrieves number of selected items (rows)
getstringwidth	-	LVM_GETSTRINGWIDTH	no	string	gets the minimum column width necessary to display the given string
gettopindex	-	LVM_GETTOPINDEX	no	no	gets the index of the first visible item (row)
getviewrect	-	LVM_GETVIEWRECT	no	RECT*	gets the bounding rectangle of all of the items in a list view in icon view

Messages Applicable to List View Controls					
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningfu l	**lParam** meaningful	
hittest	-	LVM_HITTEST	no	LV_HITTESTINFO*	determines which list view item (row) is at a specified position
insertcolumn	-	LVM_INSERTCOLUMN	no	no	inserts a new column
insertitem	-	LVM_INSERTITEM	no	LV_ITEM*	inserts a new item (row)
redrawitems	-	LVM_REDRAWITEMS	no	MAKELONG(iF, iL)	forces a redraw of a range of list view items
scroll	-	LVM_SCROLL	no	MAKELONG(dx, dy)	scrolls the contents. If report view, dx must be 0 and dy number of lines to scroll.
setcolumn	-	LVM_SETCOLUMN	yes	LV_COLUMN*	sets the attributes of a column
setcolumnwidth	-	LVM_SETCOLUMNWIDTH	yes	integer	sets the width of a column in report view or list view
setitem	-	LVM_SETITEM	no	LV_ITEM*	sets a list view item's attributes
setitemcount	-	LVM_SETITEMCOUNT	yes	no	sets the item count
setitemposition	-	LVM_SETITEMPOSITION	yes	MAKELONG(x, y)	sets the position of a list view item in standard icon or small icon view relative to the list view rectangle
setitemposition32	-	LVM_SETITEMPOSITION32	yes	POINT*	moves an item to a specified position in icon or small icon view; differs from the LVM_SETITEMPOSITION message in that it uses 32-bit coordinates
setitemstate	-	LVM_SETITEMSTATE	yes	LV_ITEM*	sets the state of an item
setitemtext	-	LVM_SETITEMTEXT	yes	LV_ITEM*	sets the text of a list view item or subitem.
update	-	LVM_UPDATE	yes	integer	updates a list view item. If the list view has the LVS_AUTOARRANGE style, the list view will be arranged

II.20.3 Colours

In both dialog.define_control and dialog.set_colour, the <*foreground*> parameter, if a number (possible generated by dialog.rgb) or a colour name, it will send a LVM_SETTEXTCOLOR message to the control. Otherwise, the <*foreground*> parameter can be one of the following keywords, with the second <*background*> parameter the colour name or RGB value you wish to set:

word	#define	means colour applied to
background	LVM_SETBKCOLOR	background
text	LVM_SETTEXTCOLOR	text
textbackground	LVM_SETTEXTBKCOLOR	text background
insertmark	LVM_SETINSERTMARKCOLOR	insertion mark
border	LVM_SETOUTLINECOLOR	border ("borderselect" style required)

You can use the same colour parameters in define_control but "text" doesn’t seem to do what it's meant to.

II.20.4 Removing Columns or Items

Use the clear service with a second argument beginning "i" or "r" (not case sensitive) to remove items (rows); if there is no third argument, all items are removed; if there is a third argument, it should be the index index (lowest legal value 0 or 1) of a row (item) to remove.

Use the clear service with a second argument beginning "c" (not case sensitive) to remove columns; the third argument should be the index index (lowest legal value 0 or 1) of a column to remove. You can’t remove all columns in a single call to **clear**.

page 258 of 259

II.21: The Animation Control

<*control_type*> **value:** Animation

Used in sample script: dialogPluginDemo7.powerpro

Animation controls are dead simple. Three specialised styles, three messages, that's it.

All you can do with them is play .avi files. I don't think you can force the avi clip to resize.

II.21.1 Styles

In addition to styles applying to all controls, there are:

Styles for Animation Controls			
this style name	**or this letter**	**#define'd symbol**	**Causes the AVI clip…**
transparent	t	ACS_TRANSPARENT	to be drawn using a transparent background rather than the background color specified in the AVI clip
autoplay	a	ACS_AUTOPLAY	to start playing as soon as it is opened. When the clip is done playing, it will automatically be repeated
centre	c	ACS_CENTER	to be centred in the control's window and leaves the animation control's size and position unchanged when the AVI clip is opened. If this style is not specified, the control will be resized when the AVI clip is opened to the size of the images in the AVI clip.

II.21.2 Messages

The following named messages are defined and can be used in dialog.send_message:

Named Messages for Animation Controls				
this message name	**or this letter**	**#define'd symbol**	**wParam** meaningful	**lParam** meaningful
open	o	ACM_OPEN	no	string
close	c	ACM_OPEN	no	no
play	p	ACM_PLAY	int; repeats, -1 forever	*from*, *to* frames; details
stop	s	ACM_STOP	no	no