LibreOffice RefCard

LibreOffice BASIC *Dialogs*

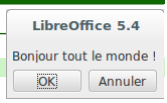
Advanced

v. 1.03 – 01/17/2018

Written using LibreOffice v. 5.3.3 – Platform : All

Dialogs In BASIC

Displaying A Simple Message




```
Print "Hello World!"
```

☞ The **Cancel** choice stops the program.

Displaying Information


```
MsgBox(MessageText[, Dialog Code[, Title]])
```

☞ Line breaks in `MessageText` with `Chr(10)` or `Chr(13)`.

Display A Message And Wait For A Response

```
Response = MsgBox(MessageText[, DialogCode[,
    Title]])
```

where

- `Response` is an integer that reflects the user's choice.
- `DialogueCode` : the sum of button codes + icon + default button

Buttons to display			
0	OK	3	Yes, No, Cancel
1	OK, Cancel	4	Yes, No
2	Stop, Retry, Ignore	5	Retry, Cancel

Icon			
0	(none)	48	ⓘ Caution
16	✘ Critical message	64	i Information (OK only)
32	❷ Question		

Default button					
0	First	256	Second	512	Last

Return values (user's selection)							
1	OK	3	Stop	5	Ignore	7	No
2	Cancel	4	Retry	6	Yes		

InputBox() Function

Function `InputBox(Message[, Title[, DefaultValue]])`
returns a string. If cancelation, returns a zero-length string.

API Dialogs

☞ The aspect of `FilePicker` and `FolderPicker` types below depend upon **Tools > Options > LibreOffice > General, Use LibreOffice dialogs**

API Dialog Types

File Selection: FilePicker Objects

`com.sun.star.ui.dialogs.FilePicker`	From above configuration.
`com.sun.star.ui.dialogs.OfficeFilePicker`	Forces LibreOffice style.
`com.sun.star.ui.dialogs.SystemFilePicker`	Forces OS style.

Directory Selection: FolderPicker Objects

`com.sun.star.ui.dialogs.FolderPicker`	From above configuration.
`com.sun.star.ui.dialogs.OfficeFolderPicker`	Forces LibreOffice style.
`com.sun.star.ui.dialogs.SystemFolderPicker`	Forces OS style.

L'objet FilePicker (ou OfficeFilePicker Ou SystemFilePicker)

```
oFilePicker = CreateUnoService("com.sun.star.ui.dialogs.FilePicker")
```

`AppendFilter()`	By pairs: `appendFilter("LiteralName", "*.xyz")` Ex : `oFilePicker.appendFilter("ODF Documents", _` `"*.odt;*.ods")`
`CurrentFilter`	Sets the default filter from the ones added using `AppendFilter` (literal name) or the user's filter selection.
`DefaultName`	Default name for the file to save.
`DisplayDirectory`	The starting directory or the user's directory selection.
`Execute`	Transfers the execution stream to the dialog and reads the return code (see return code constants below).
`Files`	An array of selected files.

`initialize()`	Dialog type selection (see type constants below).
	`Dim FPType(0) As Integer`
	`FPType(0) = 'a type constant`
	`oFilePicker.initialize(FPType())`

`MultiSelectionMode`	Disables/Enables the multi-selection mode (defaults to `False`).
`Title`	Then dialog window title.

FilePicker Type Constants

`com.sun.star.ui.dialogs.TemplateDescription.XXX` :

`FILEOPEN_SIMPLE`	0 Simple open file dialog.
`FILESAVE_SIMPLE`	1 Simple save file dialog.
`FILESAVE_AUTOEXTENSION_PASSWORD`	2 Enhanced save dialog: automatic extension + password.
`FILESAVE_AUTOEXTENSION_PASSWORD_FILTEROPTIONS`	3 Enhanced save dialog: automatic extension + password + filter options.
`FILESAVE_AUTOEXTENSION_SELECTION`	4 Enhanced save dialog: automatic extension + selection.
`FILESAVE_AUTOEXTENSION_TEMPLATE`	5 Enhanced save dialog: automatic extension + templates.
`FILEOPEN_LINK_PREVIEW_IMAGE_TEMPLATE`	6 Enhanced open dialog: insert as link + preview + template.
`FILEOPEN_PLAY`	7 Enhanced open dialog: play.
`FILEOPEN_READONLY_VERSION`	8 Enhanced open dialog: read-only + version.
`FILEOPEN_LINK_PREVIEW`	9 Enhanced open dialog: link + preview.
`FILESAVE_AUTOEXTENSION`	10 Enhanced save dialog: automatic extension
`FILEOPEN_PREVIEW`	11 Enhanced open dialog: preview.
`FILEOPEN_LINK_PLAY`	12 Enhanced open dialog: inserti as link + play.

Return Codes Constants

`com.sun.star.ui.dialogs.ExecutableDialogResults.XXX`

`CANCEL`	0 Canceled	`OK`	1 Validated

The FolderPicker Object (Or OfficeFolderPicker Or SystemFolderPicker)

```
oFldrPicker = CreateUnoService("com.sun.star.ui.dialogs.FolderPicker")
```

`Description`	Help text to display on the dialog. Does nothing on an `OfficeFolderPicker`.
`DisplayDirectory`	Starting directory.
`Execute`	Transfers the execution stream to the dialog and reads the return code (see return code constants below).
`Title`	Dialog title.
`Directory`	User's selection.

Opening A Unique File (FilePicker)

1. Create a `FilePicker`. The default type usually fits (`FILEOPEN_SIMPLE`),
2. set its properties and methods (see above),
3. execute,
4. read the return values in the`CurrentFilter`, `DisplayDirectory` and `Files` (vector) properties (`Files(0)` only has a value).

```
Dim oFilePicker As Object
Dim FileName As String

FileName = ""
'FilePicker initialization
oFilePicker = CreateUnoService("com.sun.star.ui.dialogs.FilePicker")
oFilePicker.DisplayDirectory = ConvertToURL("C:\Path\To\SomeDir")
oFilePicker.appendFilter("Calc Documents", "*.ods")
oFilePicker.CurrentFilter = "Calc Documents"
oFilePicker.Title = "Select a Calc document"
'execution and return check (OK?)
If oFilePicker.execute = _
        com.sun.star.ui.dialogs.ExecutableDialogResults.OK Then
  FileName = oFilePicker.Files(0)
End If
```

Opening Several Files (FilePicker)

1. As above,
2. set its properties and methods (esp. with `MultiSelectionMode = True`),
3. execute,
4. read the `Files()` vector that holds the user's choices.

Saving A File (FilePicker)

1. Create a `FilePicker`,
2. set its properties and methods (type `FILESAVE_XXX`) (see above),
3. execute,
4. read the return values in the`CurrentFilter`, `DisplayDirectory` and `Files` (vector) properties (`Files(0)` only has a value).

Selecting A Directory (FolderPicker)

1. Create a `FolderPicker`,
2. set its properties and methods (see above),
3. execute,
4. read the return value in `Directory`.

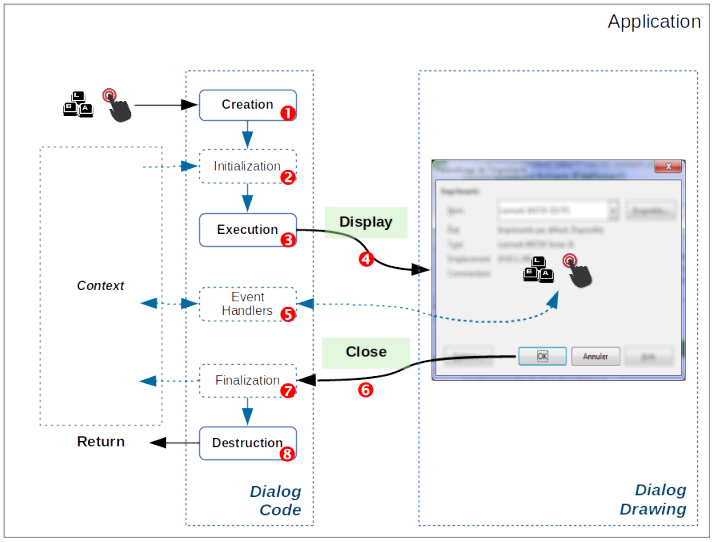
```
Dim oFP As Object
Dim DirName As String

DirName = ""
oFP = CreateUnoService("com.sun.star.ui.dialogs.FolderPicker")
oFP.DisplayDirectory = ConvertToURL("C:\Path\To\SomeDir")
oFP.Description = "Select a directory"
oFP.Title = "Select the backup directory"
If oFP.execute = _
        com.sun.star.ui.dialogs.ExecutableDialogResults.OK Then
  DirName = oFP.Directory
End If
```

Custom Dialogs 101

A BASIC dialog = a **dialog** module (drawing) + (at least) one **code** module.

Dialog Execution Sequence



The figure above illustrates a typical dialog execution sequence:

1. As a response to an application event, you **create** the dialog,
2. (**initialize** dialog controls from the application context if necessary),
3. **run** the dialog the receives the execution flow:
   4. **display**,
   5. (dialog controls **events** management),
   6. some events imply the dialog **close** (**OK**, **Cancel**) ;
7. (**finalize** to the application context if necessary),
8. the dialog is **destroyed** and the flow **returns** to the calling application.

☞ Creation, initialization, execution, finalization and destruction: processed in your code. Display, closing: automatic operations that follow the latter.

Think to the responses to control **events** (“Associating an event to a macro” and RefCard #4).

Loading Dialog Libraries

☞ Much code or several dialogs? You may want to store them in dedicated libraries.

Dialog libraries are **never** automatically loaded.

Loading libraries: beware to the **typecase**!

Modal Vs Non-modal

Modal A modal dialog takes full control upon the keyboard, mouse and screen, waiting for some action from the user. The underlying application is not accessible.

☞ By default, dialogs are modal.

Non-modal A non-modal dialog doesn’t block access to the application.
Ex : the LibreOffice **Search & replace** dialog

Multiple calls to a non-modal dialog may block the application.

Standard Custom Dialogs (modal)

This is the most frequent use.
Given a dialog module `MyDlg` and a code module `MyDlgCode` in a `MyDlgLib` library. In a code module `Sub`, we instantiate a dialog object (`oDlg`) from the dialog.

Creating / Loading In Memory

```
DialogLibraries.loadLibrary("MyDlgLib")
oLib = DialogLibraries.getByName("MyDlgLib")
oModule = oLib.getByName("MyDlg")
oDlg = CreateUnoDialog(oModule)
'on now manipulate the oDlg object
```

Calling The Dialog

`oDlg.execute` ☞ The execution flow is transferred to the dialog.

Calling And Testing The Return Value

```
If oDlg.execute = com.sun.star.ui.dialogs.ExecutableDialogResults.OK
    Then …
```

☞ The execution flow is transferred to the dialog and the return value is checked (did the user select **OK**?).

Terminating / Destroying The Dialog

```
oDlg.dispose
```

Wrap-up Example (Code Module)

Cet exemple ne montre pas de gestion d’événement.

```
Sub ShowDialog()

  Dim oLib As Object, oModule As Object, oDlg As Object

  DialogLibraries.loadLibrary("MyDlgLib")
  oLib = DialogLibraries.getByName("MyDlgLib")
  oModule = oLib.getByName("MyDlg")
  oDlg = CreateUnoDialog(oModule)
  'InitializeDlg() 'code to initialize the dialog contents
  If oDlg.execute = com.sun.star.ui.dialogs.ExecutableDialogResults.OK
    Then
    'FinalizeDlg() 'code to do something with the user's input
  End If
oDlg.dispose
End Sub
```

Non-modal Custom Dialogs

Given a dialog module `MyNMDlg` and a code module `MyNMDlgCode` in `MyNMDlgLib` library.
In a Sub of the code module, we instantiate an object (`oDlg`) from the dialog.
Apply the same technique as above, with some subtleties:

1. The dialog **display** is ensured using `oDlg.SetVisible(True)` instead of `oDlg.execute`,
2. we set two global flags:

- **a boolean flag** `gRunning` that stops multiple executions,
- **another boolean flag** `gShowMe` that controls the dialog display,

3. **events responses** (controls) must set `gShowMe to False` to close the dialog.

Displaying The Dialog

`oDlg.SetVisible(True)` ☞ The dialog is **displayed**.
The execution flow is **not** transferred to the dialog.

Wrap-up Example (Code Module)

```
'dialog display flag. Global to the module.
Dim gShowMe As Boolean
'execution flag to prevent multiple runs. Global too.
Dim gRunning As Boolean

Sub ShowNonModalDialog()
'manages the dialog creation and display

  Dim oLib As Object, oModule As Object, oDlg As Object

  'check for multiple runs
  If Not gRunning Then
    gRunning = True
    gShowMe = True
    DialogLibraries.loadLibrary("MyNMDlgLib")
    oLib = DialogLibraries.getByName("MyNMDlgLib")
    oModule = oLib.getByName("MyNMDlg")
    oDlg = CreateUnoDialog(oModule)
    'InitializeDlg() 'code to initialize the dialog contents

    'display the dialog as long as the flag is True
    Do While gShowMe
      Wait 20      'allow other software execution
      oDlg.SetVisible(True)  'keep on screen
    Loop
    'if we are here, the dialog was closed (see OnBtnOKClick)
    'FinalizeDlg() 'code to do something with the user's input
    oDlg.dispose
    gRunning = False
  End If
End Sub ' ShowNonModalDialog

Sub OnBtnOKClick(ByRef pEvt As Object)
'Response to a click on a OK button on the non-modal dialog

  'set the appropriate actions
  'then end with:
  gShowMe = False  '=> the ShowNonModalDialog while loop ends
                   'thus the dialog closes
End Sub 'OnBtnOKClick
```

Associating An Event To A Macro

A dialog communicates with the application through **events** (❺ on the figure). You thus have to write macros to respond to events occurrences (from RefCard #4):

1. **Create the macro** to call, accroding to this template:
   ```
   Sub MacroName()
   End Sub
   ```

Hint: name the macro from the object and event type.
Example : `Sub OnOKButtonClick()`

That Sub may get a parameter. Voir below “Getting Information”,

2. **select the object** that carries the event to intercept,
3. go to its settings (differs according to the object),
4. **select the event** to intercept,
5. **point to the macro** that should be run when the event fires (point 1).

☞ More information about events in the RefCard #4.

Getting Information About The Triggered Event

The event management macro can read the input parameter to get more information about the event itself:

```
Sub ReponseEvenement(ByRef Event As Object)
End Sub
```

The `Event` input object properties and methods depend from the type of event that triggered the macro call.

Most Frequent Cases For Controls

To gain access to the calling…	Interrogate
Control object	`Event.Source`
Control **model** object	`Event.Source.Model`
Control owning **dialog** object	`Event.Source.Context`

Initialization And Finalization

Initialization

(❷ in the figure) A dialog often requires information from the execution context. The initialization macro configures the dialogue contents from this data.

Finalization

(❼ in the figure) Here, we have the opposite process: setting context data from what was input in the dialog.

Managing Dialog Modules

LibreOffice manages dialog modules independently from code (see RefCard #1). We may copy such modules from a document to another.

Copying Modules From A Library To Another

(within the same document or between documents/containers)

1. In the IDE, open both source and target documents/containers,
2. open the **Macro organizer** (button),
3. go to the **Dialogs** tab, drag/drop from the source to the target.

☞ By default, modules are **moved**. To **copy**: **Ctrl** + drag/drop.

Saving A Dialog (Drawing Alone)

1. In the IDE, open the dialog module to save,
2. click the toolbar button **Export Dialog**,
3. name the file and save it.

The document is in XML format with an `.xdl` extension.

☞ Import is the reciprocal process, using the **Import Dialog** button.

Credits

Author: Jean-François Nifenecker – jean-francois.nifenecker@laposte.net

We are like dwarves perched on the shoulders of giants, and thus we are able to see more and farther than the latter. And this is not at all because of the acuteness of our sight or the stature of our body, but because we are carried aloft and elevated by the magnitude of the giants (Bernard de Chartres [attr.])

History

Version	Date	Comments
1.01	01/10/17	First version
1.03	17/01/18	Minor corrections