Table of Contents

Chapter 1:
OpenOffice.org Base Macros..................................................................................................................... 6
At a glance............................................................................................................................................... 6
Acknowledgments
Chapter 1: OpenOffice.org Base Macros

One of the great advantages of storing macros with in an OpenOffice.org Base file is portability. All pertinent macro libraries may be included and deployed to users—without the need to deploy the accompanying code as separate libraries. Additionally, embedding code libraries within a Base file gives access to a special global variable: **ThisDatabaseDocument**. Similar to the global variable **ThisComponent**—which gives access to the active OpenOffice.org document—the variable ThisDatabaseDocument gives access to the active database document.

At a glance

Illustration 1.1 shows the OpenOffice.org Basic Macros dialog. Note that, in addition to the usual library containers, a Base document also appears—**CookingRecipes.odb** in this case.

Note that the library container can store multiple libraries—making it ideal to organize code. Each library can, of course, contains any number of modules.

**Note**

Only the **Standard** library, and those bound to GUI element events are loaded by default. Any other libraries must be loaded explicitly before use. If a set of libraries is to be used heavily by the application, these can be loaded when the application starts.

Consider code listing 1. This code will fail the first time it runs. The reason being—the **Utils** library was
not loaded—even though it is in the same database file.

**Code Listing 1: Code Accessing a library that has not been loaded**

```vba
Sub pbPreviewChard_AfterMouseButtonReleased(Event As Object)
    Dim oForm As Object
    Dim iID As Integer

    oForm=Event.Source.Model.Parent
    iID=oForm.Columns.getByName("ID").getInt()
    Utils.RecipeUtils.createRecipeCard(iID)
End Sub
```

The error message does not much lend itself to clarity. When confronting this error, one will most likely wonder why this error complains about a property or method that clearly is defined. Illustration 1.2 shows such an error message.

Adding the following line of code will alleviate this problem:

```vba
BasicLibraries.loadLibrary("Utils")
```

Of course, it is good practice to check that this library has been loaded, and only load it once.

```vba
If Not BasicLibraries.isLibraryLoaded("Utils") Then
    BasicLibraries.loadLibrary("Utils")
End If
```

**Preparing for Development**

There are some steps that can be taken that will aid in development a database—especially when working with forms and macros. The *BaseTools* extension—primitive though it is—has some useful tools.
Form Document Setup

The sub routine `prepFormRunMode` looks at the run mode of a Base Form Document, and loads the respective toolbars; namely those needed when running in design mode, and those needed when running in data mode. The code is actually quite simple, and can be amended easily to accommodate individual needs. Code listing 2 shows the sub routine as well as the required sub routine that performs the actual work of showing each element.

**Code Listing 2: Configuring a Base Form Document for Appropriate Mode**

```vba
Sub prepFormRunMode
    Dim doc As Object
    Dim Layout As Object
    doc=ThisComponent
    If Not doc.supportsService("com.sun.star.text.TextDocument") Then Exit Sub
    Layout=Doc.CurrentController.Frame.LayoutManager
    If doc.CurrentController.isFormDesignMode() Then
        Layout.hideElement(bcFormsNavigationBar)
        showLayoutElement(Layout,bcFormDesign)
        showLayoutElement(Layout,bcFormControls)
        showLayoutElement(Layout,bcFormMenuBar)
    Else
        Layout.hideElement(bcFormDesign)
        Layout.hideElement(bcFormControls)
        Layout.hideElement(bcFormMenuBar)
        'comment out if not desired for all form
        showLayoutElement(Layout,bcFormsNavigationBar)
    End If
End Sub
```

```vba
Private Sub showLayoutElement(Layout As Object,ElementName As String)
    If IsNull( Layout.getElement(ElementName) ) Then
        Layout.createElement(ElementName)
    End If
    Layout.showElement(ElementName)
End Sub
```

**Note**

The constants with the name pattern `bcForm*` passed to the `showLayoutElement(...)` sub routine are text constants defined in the `Constants` module of the `BaseTools` Extension. These constants simply contain the full names (which tend to be long) of each element.

To enable this sub routine, select the `Tools` menu option from the Base window, and navigate to the `Customize` sub menu. Once in the customize dialog, select events, and then the `Loaded sub component` event. With this event entry selected, click on the button labeled `Macros...` at the top-right corner of the dialog to access the Macro Selector Dialog. Navigate to the respective container, library, and module to access the sub routine (e.g. My Macros/BaseTools/Forms/). Illustration 1.3 shows the
Customize Dialog.

Naturally, this sub routine can be located in any other place—including a library within a Base document.

Having performed these steps, open a Form Document in design mode, and note that all the toolbars needed during design-time are available. Close the document and open it in data mode, and note that only the navigation toolbar is available.

Creating Basic Modules for Active Document

Another sub routine I find useful, also located in the Forms module of the BaseTools extension, is the createFormBasicModule(..) sub routine. This sub routine creates a Basic Module for the active form document. The module is created in the gui library (creates the library if it does not exist) of the database document. The module's name is the form document's name with the "_BasicModule" suffix. Additionally, the following content is added to the module:

```
Option Explicit
'+------------------------------------------------------------------
'     Basic Module for form document: <document name>
'+------------------------------------------------------------------
```
The Option Explicit statement forces all variables to be defined which greatly aids in debugging. Many languages allow programmers to be lazy and not define variables before use. This practice may save time, but one often pays the price later. A program without explicitly defined variables is like the proverbial mother-in-law—it just sits there and waits for one to fail.

**Note**

It is important to note that there is no actual link between the Form Document and the corresponding Basic Module—aside from their names.

To use this sub routine, simply customize the form design toolbar. Right-click on the Form Design toolbar and select *Customize Toolbar*. Illustration 1.4 shows the Customize dialog for toolbars—note that it is the same dialog used to customize the Base Events.

To add a new entry to the Form Design Toolbar, select the appropriate toolbar from the combo box labeled *Toolbar* (Form Design for this case) and the appropriate item from the *Save In* combo box (OpenOffice.org Base Form Design for this case). Having made those selections, click on the button labeled *Add...* which is located on the right side of the dialog. Once in the *Add Command* dialog, select *OpenOffice.org Macros/My Macros/BaseTools/* from the Categories list box, and finally select *createFormBasicModule* from the *Commands* list box. Click on the button labeled *Add* to complete the selection. Once the item has been added, click on the button labeled *Modify* to change the name or add an icon—or use any of the other available options. Use the up and down arrows to the right of the *Commands* list box to move the command to the desired location. Illustration 1.5 shows the Form Design toolbar—with the new command button circled.
Importing Basic Libraries

It may be preferred, when deploying an application to end-users, to minimize the number of components delivered. Having the ability to embed macro libraries within a Base document means that external/third-party libraries can also be included.

Consider—for example—that an application makes use of the *Switchboard* extension. As the Switchboard extension is simply a Basic Library, it can be imported using the OpenOffice.org Basic Macro Organizer. Select Tools | Macros | Organize Macros | OpenOffice.org Basic … to open the Macros Dialog and then click on the button labeled Organizer… or simply select Tools | Macros | Organize Dialogs. Select the Libraries tab to manage the macro libraries. Illustration 1.6 shows the Macro Organizer.

The combo box labeled Locations allows selection of My Macros & Dialogs (global user defined macros), OpenOffice.org Macros and Dialogs, and any OpenOffice.org documents currently open. To import a library into a Base document, select the desired document from the Locations combo box, and the click on the button labeled Import… and select the library of choice.
At a glance

Before a library can be imported, it must have been exported as a BASIC library by using the OpenOffice.org Basic Macro Organizer—click on the Export... button after having selected the library of choice. This export process creates a folder with the same name as the library exported. The folder contains a file for every module and dialog in the library as well as two additional files: dialog.xlb and script.xlb. These two files contain a registry of all dialogs and basic modules in the library. These are the files that must be selected to import a library. Illustration 1.7 shows the contents of a sample scrip.xlb file.

As an alternative—libraries can be imported using code. Code listing 3 shows how to import a library using code. Modules may be imported similarly—by accessing the DialogLibraries rather than the BasicLibraries of the respective object.

Illustration 1.7: Contents of a script.xlb File

Before a library can be imported, it must have been exported as a BASIC library by using the OpenOffice.org Basic Macro Organizer—click on the Export... button after having selected the library of choice. This export process creates a folder with the same name as the library exported. The folder contains a file for every module and dialog in the library as well as two additional files: dialog.xlb and script.xlb. These two files contain a registry of all dialogs and basic modules in the library. These are the files that must be selected to import a library. Illustration 1.7 shows the contents of a sample scrip.xlb file.

As an alternative—libraries can be imported using code. Code listing 3 shows how to import a library using code. Modules may be imported similarly—by accessing the DialogLibraries rather than the BasicLibraries of the respective object.

<table>
<thead>
<tr>
<th>Code Listing 3: Importing a Basic Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub importBaseToolsLibrary</td>
</tr>
<tr>
<td>Dim oNewLib As Object</td>
</tr>
<tr>
<td>Dim oDb As Object</td>
</tr>
<tr>
<td>Dim oDbLibs As Object</td>
</tr>
<tr>
<td>Dim LibName As String</td>
</tr>
<tr>
<td>Dim oGlobalLib As Object</td>
</tr>
<tr>
<td>Dim I As Integer</td>
</tr>
<tr>
<td>Dim iCount As Integer</td>
</tr>
<tr>
<td>Dim sNames() As String</td>
</tr>
<tr>
<td>LibName=&quot;BaseTools&quot;</td>
</tr>
<tr>
<td>GlobalScope.BasicLibraries.loadLibrary(LibName)</td>
</tr>
<tr>
<td>oGlobalLib=GlobalScope.BasicLibraries.getByName(LibName)</td>
</tr>
<tr>
<td>oDb=ThisDatabaseDocument</td>
</tr>
<tr>
<td>oDbLibs=oDb.BasicLibraries</td>
</tr>
<tr>
<td>If oDbLibs.hasByName(LibName) Then</td>
</tr>
<tr>
<td>oDbLibs.removeLibrary(LibName)</td>
</tr>
<tr>
<td>oNewLib=oDbLibs.createLibrary(LibName)</td>
</tr>
<tr>
<td>oDbLibs.loadLibrary(LibName)</td>
</tr>
<tr>
<td>'get lib names</td>
</tr>
<tr>
<td>sNames=oGlobalLib.ElementNames</td>
</tr>
</tbody>
</table>
iCount=UBound(sNames)
For I=0 To iCount-1
  oNewLib.insertByName(sNames(I),_
    oGlobalLib.getByName(sNames(i))
Next I

End Sub

This may be an ideal situation for updating embedded libraries that are subject to change. Code listing 3 is stored in the database document, and therefore has access to the global variable ThisDatabaseDocument.

**Using the Global Variable** *ThisDatabaseDocument*

As mentioned in the previous section, one of the advantages of embedding code in a Base document is the availability of the global variable ThisDatabaseDocument. Let us continue with a brief example—see code listing 4.

**Code Listing 4: Working with the Current Database**

```vba
Sub pbAccounts_AfterMouseButtonReleased(Event As Object)
  Dim FormDoc As Object
  Dim FormDef As Object
  Dim oContainer As Object

  'access form documents container
  oContainer=ThisDatabaseDocument.FormDocuments

  'get ref to target form definition
  FormDef=oContainer.getByName("Accounts")

  'open form and store form doc model (writer doc model)
  FormDoc=FormDef.open()
End Sub
```

This code sample is broken down into three lines simply to illustrate the different components being accessed. This function could have been performed with one line of code:

ThisDatabaseDocument.FormDocuments.getByName("Accounts").open()

The Form Documents container is only of many properties that may be accessed from the variable ThisDatabaseDocument. Code listing 3, for example, accessed the Basic Libraries Container. The Data Source service (com.sun.star.sdb.DataSource) is often a useful property—from which a Connection service may be obtained (com.sun.star.sdb.Connection).

```vba
DataSource=ThisDatabaseDocument.getDataSource()
Conn=DataSource.getConnection("","")
```

Note that this sample uses the default (empty) user name and password pair. If the target database...
14 At a glance

connection requires a user name and password, it must be supplied to the getConnection(...) method. Often a database requires additional resources located somewhere in the file system—perhaps in sub directories within the Base file directory. Illustration 1.8 shows a database that uses external resources located in sub directories bin, Snapshots, and XMLTV.

![Illustration 1.8: Database with External Resources](image)

While it may seem trivial (and in reality it is), this often is a cause for problems. While the sub directories are located in the same directory as the database file itself, the full path may not be known. If the root folder is a network share, for instance, users may map it differently. Consider that the directory structure is located at `\phobos\entertainment\tv\recordings`. Some users may map directly to the `recordings` directory, others to `tv`, while others may map to `entertainment`. Furthermore, the drive letter may be different.

When writing code that accesses such resources, use the database location to drive the context path—the URL property of the database document (ThisDatabaseDocument variable). Code listing 5 shows an example on building resource paths.

**Code Listing 5: Building External Resource Paths**

```vba
Sub getContextPaths()
    Dim DocPath As String
    Dim ContextPath AS String
    Dim BinPath As String
    Dim SnapShotsPath As string
    Dim XmltvPath As String

    DocPath=convertFromURL(ThisDatabaseDocument.URL)
    GlobalScope.BasicLibraries.loadLibrary("Tools")
    ContextPath=Tools.Strings.DirectoryNameoutofPath(DocPath,"") & "\"
    BinPath=ContextPath & "bin\\"
    SnapShotsPath=ContextPath & "SnapShots\\"
    XmltvPath=ContextPath & "XMLTV\\"

    MsgBox "Bin: " & BinPath & chr(10) & "Snapshots: " & SnapShotsPath & chr(10) & "XML Tv: " & XmltvPath
End Sub
```

Note that this code listing makes use of the function `DirectoryNameoutofPath(...)` located in the `String` module of the `Tools` library—which must first be imported; the second parameter ("\") is the path
separator. For a platform independent solution, the path can be set based on the result of the run-time function `getGuiType()`—1 is Windows, and 4 is Unix. The function `convertFromURL(..)` is a run-time function that converts a URL to path format. Illustration 1.9 shows the results.

Illustration 1.9: External Resource Path Generation

Accessing the database document’s URL is nothing new—simply—the URL is more accessible via the global variable `ThisDatabaseDocument`.

These are the primary changes in OpenOffice.org 3.X. Any other approaches to writing macros to access or modify database component are still available.