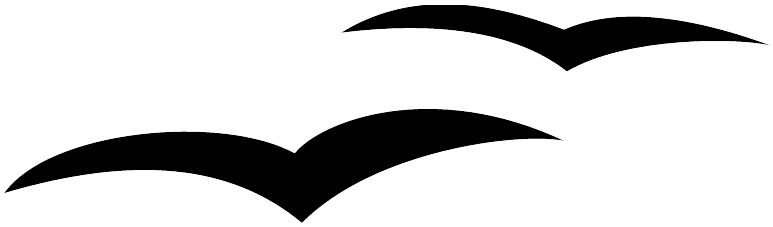
OpenOffice.org
The Open Source Office Suite

Porting Excel/VBA to Calc/StarBasic



Title: Porting Excel/VBA to Calc/StarBasic
Version: 1.0
First edition: June 6, 2004
First English edition: June 6, 2004

Contents

Overview

Although OpenOffice 1.1 Calc is able to read Microsoft Excel workbooks, compatibility extends primarily to functionality found in worksheets. Excel workbooks with Visual Basic for Applications (VBA) macros embedded do not function in Calc, even though VBA and StarBasic (SB) are syntactically the same. The reason Excel/VBA workbooks do not work under Calc/SB is due to the differences in the underlying object models for Excel and Calc.

The intent of this document is to show, by way of examples, how to port VBA macros accessing Excel objects to the equivalent SB macros accessing Calc objects. This manual is written from the perspective of an experienced Excel/VBA programmer. Hence the reader is assumed to know the VBA language and is familiar with the MS Excel Object Model. This document is not a tutorial on SB.

The information contained here is based on Excel 2000 and OpenOffice 1.1 object models. A discussion covering all aspects of the Excel object model is beyond the scope of this manual. This manual's intent is to provide sufficient examples where the reader can get started in porting VBA to SB and to point the reader to other references for more complete information.

This manual is a living document and is expected to be updated as more experience is gained. The reader should feel free to contact the author to suggest areas to expand this document.

Copyright and trademark information

Feedback

Please direct any comments or suggestions about this document to:
dev@documentation.openoffice.org and masato12610@openoffice.org

Acknowledgments

First, thank you to all the folks posting and responding on the various mailing lists and forums. These exchanges formed the basis for several examples found in this manual. Second, I'd like to thank the following individuals who took time out of their busy schedule to suggest changes to improve the document's readability and the code efficiency: dfrench, Geoff Farrell, Ian Laurenson, Andrew Pitonyak and Juergen Schmidt. Lastly, I'd like to express my sincerest thank you to my wife, Nora, for her patience and allowing me the time to work on this manual.

Modifications and updates

Version	Date	Description of Change
0.1	May 4, 2004	Preliminary version to show scope of coverage and proposed level of detail for early feedback.
0.2	May 12, 2004	Add examples for Application, Workbooks, Workbook, Worksheets, Worksheet, Range/Cell. Add description of the object information utility spreadsheet. Miscellaneous editorial changes.
0.3	May 23, 2004	Add examples to Range/Cell, UserForms, Controls. Incorporated feedback from various reviewers. Rewrote Appendix A to cover XRay tool. Add Appendix B for supporting functions supporting functions developed for this manual.
0.4	May 30, 2004	Miscellaneous editorial changes and code improvements. Add examples for processing activation and deactivation events for worksheets. Document steps to port sample Excel workbook to Calc spreadsheet. Final preliminary draft prior to public release.
1.0	June 5, 2004	Miscellaneous editorial changes and code improvements. Added discussion on Multi-page Dialogs. Added pointers to reference material throughout document.

Introduction

This chapter introduces the core concepts that provide a basis for the discussion that follows in the rest of this document.

After establishing some core concepts, the document is composed of chapters that cover the following topics:

- Examples that compare Visual Basic for Applications (VBA) code interacting with the Excel object model to StarBasic (SB) code interacting with the OpenOffice object model.
- Discussion on the differences between the integrated development environments (IDE) provide with VBA and SB
- Discussion on converting a sample Excel workbook with VBA macros into a Calc Spreadsheet with SB macros.

Terminology

The terminology used in this document is geared toward Excel/VBA programmers because they comprise the target audience. The following convention is followed. This manual uses Excel specific terms, and if there is a different Calc term for the equivalent entity, it follows the Excel term in square brackets. See the following as illustrative examples:

- workbook [spreadsheet]
- worksheet [sheet]

StarBasic Background

For the Excel/VBA programmer, SB is a Basic programming language very similar to VBA. The primary reason that VBA does not work in Calc, even though Calc is able to read the Excel workbook, is that Calc uses a different method to access the workbook [spreadsheet] components, such as cells on the worksheet [sheet]. The access mechanisms are different in Calc. Specifically the objects, attributes and methods use different names and the corresponding behavior is sometimes slightly different.

For those who wish a better understanding of SB, there are several documents publicly available that explain the language and programming environment. These documents, listed in the Bibliography, can be found on the Web.

- *StarOffice 7 Software Basic Programmer's Guide*
- *Migrating from Microsoft Office to StarOffice 7*
- *Useful Macro Information For OpenOffice*
- *How to Use BASIC Macros in OpenOffice.org*

These are excellent resources for those who are getting started in SB macro programming.

Understanding the OpenOffice Object Model

Although this manual answers many questions about porting Excel/VBA macros to Calc/SB, it is not complete – not all questions are answered. The reader may find it necessary to refer to the object model documentation for OpenOffice products. For the Excel/VBA programmer, it may take some some time to become comfortable with the way that OpenOffice objects are documented.

The primary difference between the Excel object model and the OpenOffice object model is that Excel's model does not take advantage of all of the features that constitute an object-oriented programming environment. In some publications, Microsoft's object model for their products, such as Excel, is termed "object-like".

In a true object-oriented programming model, there is the concept of inheritance. This concept allows one object's definition and implementation to be based on another object's definition and implementation. Microsoft's object-like model does not support inheritance.

To illustrate inheritance, consider the following example. There is an object called "Shape" with a method called "move()" that moves the "Shape" around on the display screen. In a true object-oriented programming environment, a new object called "Circle", which is a type of "Shape", can be implemented in the following manner. Instead of forcing "Circle" to implement its own "move()" method for moving around the display screen, the "Circle" object inherits the "move()" method from the "Shape" object.

The paradigm used in OpenOffice consists of interfaces and services. An interface defines methods. If an object implements an interface then that object must support all of the methods defined by the interface. An interface may be derived from another interface – in other word, inheritance. Assume that a "Circle" interface inherits from a "Shape" interface. Any object that implements the "Circle" interface must implement every method defined by both the "Circle" interface and the "Shape" interface. Although it is not possible to inherit from more than one interface at a time, this is scheduled to be changed in a future release of OpenOffice. A service defines an object by specifying the interfaces and properties that the object supports – a property may be defined as optional. A service may also specify that it supports other services. An interface always contains an X in its name. For example, the com.sun.star.drawing.XShape interface defines the methods to get and set a shapes position and size. The com.sun.star.drawing.Shape service (notice that the X is missing from the name) defines an object that has the XShape interface – it supports a few other interfaces and some properties as well. Although the services and interfaces contain long names, they are frequently abbreviated by dropping the first part of the name; for example XShape.

In terms of an Excel/VBA programmer understanding the Calc/SB object model, the concept of inheritance is important. Consider the following situation. In Excel, assume there exists a named range called **Range("MyMatrix")**. This respresents a two-dimensional array of cells in a worksheet. In Excel, to determine the number of rows in the range, a programmer can accesses the range property **Range("MyMatrix").Rows.Count**.

To find the equivalent information in Calc/SB, the programmer can consult the "Spreadsheet" section in the OpenOffice *Developer's Guide*. First, to access the range, there is a method defined by the XCellRange interface called getCellRangeByName. The XCellRange interface is exported by many services including CellRange and Spreadsheet.

Using the method **getCellRangeByName()** we are able to locate the range "MyMatrix" using the call .**getCellRangeByName("MyMatrix")**.

From the "Spreadsheet" section of the Developer's Guide, the programmer sees that using the service com.sun.star.sheet.SheetCellRange, we obtain access to the service XColumnRowRange. This services provides access to the columns and rows of the range. From here we see that we can invoke the method **getRows()** to retrieve the collection of rows making up the range.

However, at this point it is not clear how to get the number of rows. By remembering the concept of inheritance, the programmer should realize that "rows" is a specialization of the class "collection", and according the the object model, "rows" inherits from "collections" Now looking at the methods associated for collections, the programmer sees in the com::sun::star::container::XIndexAccess interface a method **getCount()** that retrieves the number of items in a collection.

Putting all of this together we now have a way to determine the number of rows in a range of cells. The SB call looks like
ThisComponent.CurrentController.ActiveSheet.getCellRangeByName("MyMatrix").getRows.getCount()

The moral of this little tale is that the Excel/VBA programmer, in making the transition to StarBasic, should remember to consider the concept of inheritance.

The following URL are the main reference material for this manual:

- http://api.openoffice.org/DevelopersGuide/DevelopersGuide.html
- http://api.openoffice.org/docs/common/ref/com/sun/star/module-ix.html

In doing research for this manual, a useful debugging tool was found. The tool is called XRay, developed by Bernard Marcelly, and can be found at http://www.ooomacros.org/dev.php101416. XRay allows a programmer to inspect at run-time the various Calc objects. This is similar in function to the VBA debugger. In combination with the downloaded OOo SDK, XRay is able to bring up SDK related documentation for an object while you are using XRay to view Calc objects. This feature is useful in understanding the Calc object model. Features of XRay are illustrated in Appendix A.

Examples of Porting Visual Basic for Applications to StarBasic

This section is organized by MS Excel objects. For the Excel objects covered in this manual, Visual Basic for Application (VBA) code fragments are shown using a particular method or property. Along side the VBA code fragment, the equivalent, or as close to equivalent that is possible, StarBasic (SB) code fragment is shown.

One general note on the difference between VBA and SB. In VBA, when an Excel object is referenced, such as a range of cells, unless explicitly coded, the cell range is assumed to be in the currently active Excel container, such as the workbook (ActiveWorkbook) and worksheet (ActiveSheet). In SB, on the other hand, no such assumption is made, so each reference to a Calc object must be fully qualified. In other words, you have to specify the workbook [spreadsheet] and worksheet [sheet].

One technique in Excel/VBA to determine macro code is to use the macro recording function to get an initial set of code. This resultant code can often be generalized.

While the same technique can be used in Calc/SB, the experience to date in using the technique has not been very successful. The code generated by the macro recorder is based on interacting with the spreadsheet versus recording the resultant manipulations of the spreadsheet object model.

It is possible to generalize the recorded code. However, it provides little insight into use of the spreadsheet object model. An alternative to using the native macro recorder feature in Calc is to download the Calc macro recorder from http://ooomacros.org/user.php written by Paolo Mantovani. Paolo's macro recorder creates a macro that primarily uses references to the Calc objects rather than the more cryptic dispatcher calls. The macro guide at http://www.math.umd.edu/~dcarrera/openoffice/docs/ contains an excellent description of how to use the macro recorder and arrange your macros into libraries.

General Programming Notes

Indicator in Excel [Calc] that indicates a macro is currently executing.

Excel	The mouse pointer changes from an arrow to an hourglass.
Calc	The mouse pointer does not change. There does not appear to be any indication that a macro is running in Calc.

Manually terminating a macro executing

Excel	Ctrl-Break
Calc	• **Tools** > **Macros** > **Marco** > **Edit** • Press **Stop** button

Assigning an object to a variable

VBA	`Sub MyProc`
	`  Dim wksh as Worksheet`
	`  set wksh = ActiveWorksheet`
	`End Sub`

SB	`Sub MyProc`
	`  Dim oSheet as Object`
	`  oSheet = ThisComponent.CurrentController.ActiveSheet`
	`  'or`
	`  set oSheet = ThisComponent.CurrentController.ActiveSheet`
	`End Sub`

Usage Note: While the **set** statement is defined in SB, its use does not seem to be enforced, as in VBA. In Excel, in addition to the generic Object type, there are various specific object types (**Worksheet**, **Workbook**, **Range**, etc.), however, in Calc, there is only the generic **Object** type.

Application

Object representing the workbook [spreadsheet] that is active

VBA	`ActiveWorkbook`
SB	`ThisComponent`

Reference:
http://api.openoffice.org/docs/DevelopersGuide/ProfUNO/ProfUNO.htm#1+Professional+UNO
http://api.openoffice.org/docs/DevelopersGuide/BasicAndDialogs/BasicAndDialogs.htm#1+3+2+2+ThisComponent

Object representing the worksheet [sheet] in the workbook [spreadsheet] that is active

VBA	`ActiveSheet`
SB	`ThisComponent.CurrentController.ActiveSheet`

Reference:
http://api.openoffice.org/docs/DevelopersGuide/OfficeDev/OfficeDev.htm#1+Office+Development
http://api.openoffice.org/docs/common/ref/com/sun/star/sheet/XSpreadsheetView.html#getActiveSheet

Object representing the cell that is active

VBA	`ActiveCell`
SB	`ThisComponent.getCurrentSelection`

Usage Note: See Pitonyak's document (Chapter 18) for qualifications about this. In a nutshell, **getCurrentSelection** returns the object that currently has the focus just prior to the start of the macro execution. This is the **ActiveCell** only if a single cell has focus just prior to start of the macro.

Turn-off screen updating

VBA	`Application.ScreenUpdating = False`
SB	`ThisComponent.LockControllers`

Reference:
http://api.openoffice.org/docs/common/ref/com/sun/star/frame/XModel.html#lockControllers

Turn-on screen updating

VBA	`Application.ScreenUpdating = True`
SB	`ThisComponent.UnlockControllers`
Reference: http://api.openoffice.org/docs/common/ref/com/sun/star/frame/XModel.html#unlockControllers	

Temporarily suspend the execution of the macro program for 1 second

VBA	`Application.Wait(Now + TimeValue("00:00:01"))`
SB	`Wait 1000`

Usage Note: In SB, the Wait is part of the SB environment. The argument to the **Wait** statement is the number of milliseconds to delay. In testing this code, the VBA procedure drove the processor to 100 per cent busy. On the other hand the SB **Wait** statement does not drive the processor to 100 percent busy.

Reference: OpenOffice.org Basic On-line Help

Calling Excel [Calc] worksheet [sheet] function in VBA [SB]

VBA

```
Sub MyProc
    msgbox WorksheetFunctions.Average(Range("A1:A5"))
    msgbox WorksheetFunctions.Max(Range("A1:A5"), _
         Range("C1:C5"))
End Sub
```

SB

```
Sub MyProc
    Dim oSheet, FuncService
    Rem Create service to access sheet functions
    FuncService = _
       createunoservice("com.sun.star.sheet.FunctionAccess")

    oSheet = ThisComponent.CurrentController.ActiveSheet

    msgbox FuncService.callFunction("AVERAGE", _
        array(oSheet.getCellRangeByName("A1:A5")))
    msgbox FuncAcc.CallFunction("MAX", _
        array(oSheet.getCellRangeByName("A1:A5"), _
           oSheet.getCellRangeByName("C1:C5")))

End Sub
```

Usage Note: Two arguments are needed for **callFunction()** method. The first is a string containing the name of the worksheet [sheet] function to invoke. The second is an array containing the arguments to that function.

Reference:
http://api.openoffice.org/docs/DevelopersGuide/Spreadsheet/Spreadsheet.htm#1+4+2+1+Calculating+Function+Results and http://api.openoffice.org/docs/common/ref/com/sun/star/sheet/XFunctionAccess.html

Workbooks/Workbook

List names all open workbooks [spreadsheets]

VBA

```
Sub MyProc
    Dim wbk as Workbook

    For Each wbk in Workbooks
        msgbox wbk.Name
    next
End Sub
```

SB

```
Sub ListDocs
  Dim oDocs As Object, oDoc As Object
  REM Load the included "Tools" library
  GlobalScope.BasicLibraries.LoadLibrary("Tools")
  oDocs = StarDesktop.getComponents().createEnumeration()
  Do While oDocs.hasMoreElements()
    oDoc = oDocs.nextElement()
    REM Ignore any component that is not a document.
    REM The IDE, for example
    If HasUnoInterfaces(oDoc, "com.sun.star.frame.XModel") Then
      REM If there is no URL, then do not try to find it
      If oDoc.hasLocation() Then
        REM Use the FileNameOutOfPath routine included with OOo
        MsgBox FileNameOutOfPath(oDoc.getURL()) &_
            " is of type " & GetDocumentType(oDoc)
      End If
    End If
  Loop
End Sub
```

Usage Note: The **oDoc.nextElement()** call returns all opened OO.o documents including Writer documents. So the potential exists to return more than just open Calc documents.

Reference:
http://api.openoffice.org/docs/DevelopersGuide/BasicAndDialogs/BasicAndDialogs.htm#1+3+2+1+StarDesktop,

Open workbook "My2ndWorkbook" that is located in the same directory as the currently active workbook.

VBA

```
Sub MyProc
    Dim NewWorkbook as Workbook

    set NewWorkbook = Workbooks.Open (ActiveWorkbook.Path _
       & "\My2ndWorkbook.xls")
End Sub
```

SB

```
Sub MyProc
    Dim DirectoryName as String
    Dim NewWorkbook as Object
    Dim NoArgs()   'empty array for no arguments

    Rem Assume DirectoryName variable contains directory
    Rem location of the currently active workbook

    NewWorkbook = StarDesktop.loadComponentFromURL _
       ("file:///" & DirectoryName & "/My2ndWorkbook.sxc", _
       "_blank",0 ,NoArgs() )
End Sub
```

Usage Note: See example for obtaining directory of currently active workbook [spreadsheet] later on in this manual. For both Excel and Calc, if there is a macro to be executed when the workbook [spreadsheet] is opened, the macro will not be executed using the above code fragment. Regarding the specific SB code show above, another side-effect is that no macro associated with any event, such as "When Initiating" for controls, will execute. If it is desired to execute the macros based on events occurring in the workbook [spreadsheet], see the next example.

Reference:
http://api.openoffice.org/docs/DevelopersGuide/OfficeDev/OfficeDev.htm#1+1+5+1+Loading+Documents and
http://api.openoffice.org/docs/common/ref/com/sun/star/frame/XComponentLoader.html#loadComponentFromURL

Open workbook "My2ndWorkbook" that is located in the same directory as the currently active workbook and execute the macro associated with the opening of the workbook [spreadsheet].

VBA

```
Sub MyProc
    Dim NewWorkbook as Workbook

    set NewWorkbook = Workbooks.Open (ActiveWorkbook.Path _
       & "\My2ndWorkbook.xls")
    NewWorkbook.RunAutoMacros xlAutoOpen
End Sub
```

SB

```
Sub MyProc
    Dim DirectoryName as String
    Dim NewWorkbook as Object
    Dim Args(0) as new com.sun.star.beans.PropertyValue

    Rem Assume DirectoryName variable contains directory
    Rem location of the currently active workbook

    Args(0).Name = "MacroExecutionMode"
    Args(0).Value = _
          com.sun.star.document.MacroExecMode.ALWAYS_EXECUTE
    NewWorkbook = StarDesktop.loadComponentFromURL _
       ("file:///" & DirectoryName & "/My2ndWorkbook.sxc", _
       "_blank",0 ,Args() )
End Sub
```

Usage Note: This enables the macro associated with the "Open Document" event to execute when the spreadsheet is opened. In addition, other macros associated with other events, such as "When Initiating" event for controls, will function as well.
Reference: ***http://api.openoffice.org/servlets/ReadMsg?list=dev&msgNo=10707*** , http://api.openoffice.org/docs/common/ref/com/sun/star/document/MediaDescriptor.html and http://api.openoffice.org/docs/common/ref/com/sun/star/beans/PropertyValue.html

Close workbook [spreadsheet] opened in the previous example

VBA	`NewWorkbook.Close`
SB	`NewWorkbook.Close(False)`

Reference: http://api.openoffice.org/docs/DevelopersGuide/OfficeDev/OfficeDev.htm#1+1+5+2+Closing+Documents and http://api.openoffice.org/docs/common/ref/com/sun/star/util/XCloseable.html

Execute a macro when a workbook [spreadsheet] is opened.

VBA	`Excel predefined procedure Workbook_Open() associated with the workbook component`
SB	`User macro assigned to the "Open Document" event through the` **`Tools > Macros > Macro > Assigned...`** `sequence.`

Execute a macro when a workbook is closed

VBA	`Excel predefined procedure Workbook_BeforeClose() associated with the workbook component`
SB	`User macro assigned to the "Close Document" event through the Tools > Macros > Macro > Assigned... sequence.`

Usage Note: In the Excel environment, the signature for the procedure is **Workbook_BeforeClose(Cancel as Boolean)**. This allows the macro to cancel the close operation by setting **Cancel = True**. To cancel the close in OOo, you must register a listener for the close event and then veto the close.
Reference: http://www.oooforum.org/forum/viewtopic.php?t=3576 and http://api.openoffice.org/docs/DevelopersGuide/OfficeDev/OfficeDev.htm#1+1+5+2+Closing+Documents

Get filename of the ActiveWorkbook [ThisComponent]

VBA	`Sub MyProc`
	`  ActiveWorkbook.Name`
	`End Sub`

SB	`Sub MyProc`
	`  Dim URLStr as String`
	`  Dim FileName as String`
	
	`  REM Load the included "Tools" library`
	`  GlobalScope.BasicLibraries.LoadLibrary("Tools")`
	`  REM This code assumes that the file has been saved`
	`  REM at least once so that it has a URL.`
	`  URLStr = ThisComponent.getURL()`
	`  FileName = FileNameOutOfPath(URLStr)`
	
	`End Sub`

Usage Note: Format of URL, at least for file based documents are "file:///<directory>/<filename>".

Get location (directory) of the ActiveWorkbook [ThisComponent]

VBA	`Sub MyProc`
	`  ActiveWorkbook.Path`
	`End Sub`

SB	`Sub MyProc`
	`  Dim URLStr as String`
	`  Dim Path as String`
	`  REM Load the included "Tools" library`
	`  GlobalScope.BasicLibraries.LoadLibrary("Tools")`
	`  URLStr = ThisComponent.getURL()`
	`  Path = DirectoryNameoutofPath(URLStr, "/")`
	
	`End Sub`

Usage Note: Format of URL, at least for file based documents are "file:///<directory>/<filename>". Use the method **ConvertFromURL()** to convert from URL notation to the standard notation.

Worksheets/Worksheet

Add a new worksheet [sheet] named "MyNewSheet" to the current workbook

VBA

```
Sub MyProc
    Dim wksh as Worksheet

    Rem Add new worksheet before ActiveSheet
    set wksh = Worksheets.add
    wksh.Name = "MyNewSheet"

    Rem Add new worksheet after ActiveSheet
    set wksh = Worksheets.add after:=ActiveSheet
    wksh.Name = "MyNewSheet"

    Rem Add new worksheet before Worksheet "SomeOtherSheet"
    set wksh = Worksheets.Add before:= _
       Worksheets("SomeOtherSheet")
    wksh.Name = "MyNewSheet"

    Rem Add new worksheet after worksheet "SomeOtherSheet"
    set wksh = Worksheets.Add after:= _
        Worksheets("SomeOtherSheet")
     wksh.Name = "MyNewSheet"
End Sub
```

SB

```
''This code fragment makes use of a user defined function
'''findSheetIndex() documented in Appendix B.

Sub MyProc
    Dim oSheeet as object
    Dim oSheets
    oSheets = ThisComponent.Sheets

    Rem Add new sheet at end of collection
    oSheets.insertNewByName("SomeOtherSheet", _
        oSheets.getCount())

    Rem Add new sheet before ActiveSheet
    oSheet = _
        ThisComponent.CurrentController.ActiveSheet
    ThisComponent.Sheets.InsertNewByName( _
       "NewSheet_2", _
        findSheetIndex(oSheet.Name) )

    Rem Add new sheet after ActiveSheet
    oSheet = _
       ThisComponent.CurrentController.ActiveSheet
    oSheets.InsertNewByName( "NewSheet_3", _
        findSheetIndex(oSheet.Name)+1 )

    Rem Add new sheet before sheet "SomeOtherSheet"
    oSheets.InsertNewByName( "NewSheet_4", _
        findSheetIndex("SomeOtherSheet"))

    Rem Add new sheet after sheet "SomeOtherSheet"
    oSheets.InsertNewByName( "NewSheet_5", _
        findSheetIndex("SomeOtherSheet")+1 )
End Sub
```

Usage Note: For purposes of illustration in this manual, no error checking is done for the return value of **findSheetIndex()**. In the event the worksheet [sheet] is not found, -1 is returned by the function.

Delete worksheet [sheet] named "MyNewSheet" from the current workbook [spreadsheet]

VBA	`Worksheets("MyNewSheet").Delete`
SB	`ThisComponent.Sheets.removeByName("MyNewSheet")`
Reference: http://api.openoffice.org/docs/DevelopersGuide/ProfUNO/ProfUNO.htm#1+3+5+Collections+and+Containers and http://api.openoffice.org/docs/common/ref/com/sun/star/container/XNameContainer.html#removeByName	

Do worksheet [sheet] specific processing when "Sheet1" or "Sheet2" are activated or deactivated.

VBA	In Excel, predefined procedures exist the events of activating and deactivating worksheets. For each worksheet, add code to the predefined procedures. The procedure stubs are shown below.
	
	`Private Sub Worksheet_Activate()`
	
	`  Rem code worksheet specific processing here`
	
	`End Sub`
	
	`Private Sub Worksheet_Deactivate()`
	
	`  Rem code worksheet specific processing here`
	
	`End Sub`

SB

```
' Listener for "ActiveSheet" property changes
Global oActiveSheetListener as Object
' Work variable to hold name of current worksheet
Global CurrentWorksheetName as String

'Procedure to turn-on ActiveSheet Property Change Listener
Sub WorksheetActivationListenerOn

  ' Initialize variable
  CurrentWorksheetName = ""

  'create listner
  oActiveSheetListener = createUnoListener("ACTIVESHEET_", _
          "com.sun.star.beans.XPropertyChangeListener")

  'attach listener to ActiveSheet property
  ThisComponent.CurrentController. _
      addPropertyChangeListener("ActiveSheet", _
         oActiveSheetListener)

End Sub

' procedure to turn-off ActiveSheet property change listener
sub WorksheetActivationListenerOff
   ThisComponent.CurrentController. _
         removePropertyChangeListener("ActiveSheet", _
             oActiveSheetListener)
end sub

' procedure to process each change to the ActiveSheet property
Sub ACTIVESHEET_propertyChange(oEvent)

   'first do deactivate processing
   select case CurrentWorksheetName
      case "Sheet1"
         'do Sheet1 specific deactivate processing

      case "Sheet2"
         'do Sheet2 specific deactivate processing

   end select

   'now execute activate processing for
   'the newly activated sheet
   select case oEvent.Source.ActiveSheet.Name
      case "Sheet1"
         'Do Sheet1 specific activate processing

      case "Sheet2"
         'Do Sheet2 specific activate processing

   end select

    'Save name of newly activated sheet
    CurrentWorksheetName = oEvent.Source.ActiveSheet.Name
end Sub
```

Usage Note: In Calc, there are no predefined events specific to activating or deactivating a sheet. The approach taken above is to add a PropertyListener to the **ActiveSheet** property of the **ThisComponent.CurrentController** object. This technique is described in this posting: http://www.oooforum.org/forum/viewtopic.php?t=5135 One limitation of this approach is if the view changes, such as performing a print preview, the event listener for changes in the ActiveSheet property is lost. As of this writing it is not clear how to regain control to re-establish the event listener.

The approach shown above has three procedures. The procedure **WorksheetActivationListenerOn** is called only once to initialize the Listener. This procedure can be executed by the macro assigned to the "Open Document" event for the spreadsheet.

Procedure **WorksheetActivationListenerOff** is called whenever it is desired to disable the activate/deactivate functions.

Procedure **ACTIVESHEET_propertyChange(oEvent)** is called whenever the value of the **ActiveSheet** property in the **ThisComponent.CurrentController** object is changed. **oEvent** describes the change property event. The **oEvent** object is described http://api.openoffice.org/docs/common/ref/com/sun/star/beans/PropertyChangeEvent.html

Activate worksheet [sheet] named "MySheet"

VBA	`Sub MyProc`
	`  Worksheets("MySheet").Activate`
	`End Sub`

SB	`Sub MyProc`
	`  Dim oSheet as Object`
	
	`  oSheet = ThisComponent.Sheets.getByName("MySheet")`
	`  ThisComponent.CurrentController.setActiveSheet(oSheet)`
	`End Sub`

Range/Cell

Storing a number into a cell.

VBA	`Range("B1").Value = 12`
SB	`ThisComponent.CurrentController.ActiveSheet.getCellRangeByName ("B1").Value = 12`
Reference: http://api.openoffice.org/docs/DevelopersGuide/Spreadsheet/Spreadsheet.htm#1+3+1+5+Cells and http://api.openoffice.org/docs/common/ref/com/sun/star/table/XCell.html	

Retrieving a number from a cell.

VBA	`MyNumber = Range("MyCell").Value`
SB	`MyNumber = ThisComponent.CurrentController.ActiveSheet. getCellRangeByName("MyCell").Value`

Storing a string into a cell.

VBA	`Range("B1").Value = "DOG"`
SB	`ThisComponent.CurrentController.ActiveSheet. getCellRangeByName ("B1").String = "DOG"`
Reference: http://api.openoffice.org/docs/DevelopersGuide/Spreadsheet/Spreadsheet.htm#1+3+1+5+Cells and http://api.openoffice.org/docs/common/ref/com/sun/star/text/XTextRange.html	

Retrieving a string from a cell.

VBA	`MyString = Range("MyCell").Value`
SB	`MyString = ThisComponent.CurrentController.ActiveSheet. getCellRangeByName("MyCell").String`

Access the cell C4 in the Range("B1:E5") by relative position

VBA	`Range("B1:E5").Cells(4,2).Value`
	`'or`
	`Range("B1:E5").Offset(3,1).Value`

SB	`ThisComponent.CurrentController.ActiveSheet.getCellRangeByName ("B1:E5").getCellByPosition(1,3).Value`

Usage Note: In VBA there are two ways for relatively accessing a cell. The first method **Cells()** uses the relative row and column locations and the numbering is "1" based, i.e., the top left cell is **.Cells(1,1)**. The second method is **Offset()** and uses a "0" based row and column locations, i.e., the top left cell is **.Offset(0,0)**. In SB, the arguments for **. getByCellPosition()** are reversed from the VBA arguments, i.e., the column number is the first argument, followed by the row number. The numbering is "0" based, i.e., the top left cell is **.getByCellPosition(0,0)**. This makes . **getByCellPosition()** similar to the VBA **. Offset()** method.
Reference:
http://api.openoffice.org/docs/DevelopersGuide/Spreadsheet/Spreadsheet.htm#1+3+1+4+Cell+Ranges and http://api.openoffice.org/docs/common/ref/com/sun/star/table/XCellRange.html

Acccess the cell F2 in the Range("B1:E5") by relative position. (**Note:** Cell F2 is out of the Range("B1:E5"))

VBA	`Range("B1:E5").Cells(2,5).Value` `Range("B1:E5").Offset(1,4).Value`
SB	***Not possible in SB***

Usage Note: If the same SB technique is used as in the previous example of accessing cell C4, the execution of the macro program will be interrupted with an **com.sun.star.lang.IndexOutOfBoundException**. VBA does not enforce any bounds checking on the row and column indices for the cell range. If accessing a cell outside the specified range is a requirement for a SB macro program, the only solution is to calculate the absolute cell locations on the worksheet [sheet].

Print the address of a cell or range of cells on the ActiveSheet

VBA

```
Sub MyProc
   'Following will display $B$3
   msgbox Range("B3").Address

   'Following will display $B$3:$D$5
   msgbox Range("B3:D5").Address

   'Cell B5 is named "MyCell", following will display $B$5
   msgbox Range("MyCell").Address
End Sub
```

SB

```
''This code fragment makes use of a user defined function
'''CellRangeAddressString() documented in Appendix B.

Sub MyProc
   Dim oSheet as Object

   oSheet = ThisComponent.CurrentController.ActiveSheet

   'Following will display $B$3
   msgbox CellRangeAddressString( _
      oSheet.getCellRangeByName("B3"))

   'Following will display $B$3:$D$5
   msgbox CellRangeAddressString( _
      oSheet.getCellRangeByName("B3:D5"))

   'Cell B5 is named "MyCell", following will display $B$5
   msgbox CellRangeAddressString( _
      oSheet.getCellRangeByName("MyCell")

   'following will display $D$7
   msgbox CellRangeAddressString( _
      oSheet.getCellByPosition(3,6))


End Sub
```

Usage Note: After some time spent researching and experimentation, the approach of using the Calc sheet function "ADDRESS()" was selected. It may be possible that a method or property exists in the UNO object model to obtain a string representation of the address of a cell or range of cells but it is not clear as of this writing.

Find the cell at the end of a row or column of data in a worksheet [sheet]. Assume all the cells in range B3:E15 contains data. This does not depend on knowing the actual number of rows or columns in the data range.

VBA

```
Sub MyProc
   'go to upper left corner of range
   Range("B3").Select

   'Find the last cell in the current row of data
   'This takes the cursor to cell E3
   Selection.end(xlToRight).Select

   'Find the last cell in the column
   'this takes the cursor to cell E15
   Selection.end(xlDown).Selection

   'Find the first cell in the current row
   'this takes the cursor to cell B15
   Selection.end(xlToLeft).Select

   'Find first cell in the current column
   'this takes the cursor to cell B3
   Selection.end(xlUp).Select

End Sub
```

SB	'''This code fragment makes use of a user defined function
	'''MoveCursorToEnd() documented in Appendix B.
	
	Sub MyProc
	  Dim oSheet as Object, oCell as Object
	
	  '''get some useful objects
	  oSheet = ThisComponent.CurrentController.ActiveSheet
	
	  '''go to upper slef corner of range
	  oCell = oSheet.getCellRangeByName("B3")
	  ThisComponent.CurrentController.select(oCell)
	
	  'Find the last cell in the current row
	  'this takes cursor to cell E3
	  oCell = MoveCursorToEnd(oCell,"xlToRight")
	  ThisComponent.CurrentController.select(oCell)
	
	  'Find last (bottom) cell in the current column
	  'This takes cursor to cell E15
	  oCell = MoveCursorToEnd(oCell,"xlDown")
	  ThisComponent.CurrentController.select(oCell)
	
	  'Find first cell in the current row
	  'This takes cursor to cell B15
	  oCell = MoveCursorToEnd(oCell,"xlToLeft")
	  ThisComponent.CurrentController.select(oCell)
	
	  'Find first (top) cell in the current column
	  'This takes cursor to cell B3
	  oCell = MoveCursorToEnd(oCell,"xlUp")
	  ThisComponent.CurrentController.select(oCell)
	
	End Sub

Clear the contents in the range of cells on the ActiveSheet. This does not affect any formatting of the cells.

VBA	Range("B1:E5").ClearContents

SB	ThisComponent.CurrentController.ActiveSheet.getCellByName
	("B1:E5").clearContents(_
	  com.sun.star.sheet.CellFlags.VALUE _
	  +com.sun.star.sheet.CellFlags.STRING _
	  +com.sun.star.sheet.CellFlags.DATETIME)

Reference: http://api.openoffice.org/docs/DevelopersGuide/Spreadsheet/Spreadsheet.htm#1+3+1+4+8+Operations, http://api.openoffice.org/docs/common/ref/com/sun/star/sheet/XSheetOperation.html#clearContents and http://api.openoffice.org/docs/common/ref/com/sun/star/sheet/CellFlags.html	

Clear a range of cells on the worksheet [sheet] "MySheet". This clears everything associated with the cell including formatting.

VBA	`Worksheets("MySheet").Range("B1:E5").Clear`
SB	`ThisComponent.Sheets.getByName("MySheet").getCellRangeByName ("B1:E5").clearContents(_ com.sun.star.sheet.CellFlags.VALUE _ + com.sun.star.sheet.CellFlags.STRING _ + com.sun.star.sheet.CellFlags.DATETIME _ + com.sun.star.sheet.CellFlags.ANNOTATION _ + com.sun.star.sheet.CellFlags.FORMULA _ + com.sun.star.sheet.CellFlags.HARDATTR _ + com.sun.star.sheet.CellFlags.STYLES _ + com.sun.star.sheet.CellFlags.OBJECTS _ + com.sun.star.sheet.CellFlags.EDITATTR)`

Assign a user defined name "MyCells" to the cells B2:C3 via the Excel [Calc] user interface

Excel	One of two methods: 1) Highlight cells B2:C3, select the tool bar options: **Insert > Name > Define** , enter name "MyCells" in pop-up window and press "Add" button. 2) Highlight cells B2:C3, then enter "MyCells" in the Name Field on the active window.
Calc	One of two methods: 1) Highlight cells B2:C3, select the tool bar options: **Insert > Names > Define**, enter name "MyCells" in pop-up window and press "Add" button. 2) Highlight cells B2:C3, press Ctrl-F3, enter name "MyCells" in pop-up window and press "Add" button.

Assign a user defined name "MyCells" to the cells B2:C3 on "Sheet1" and the same name to cells A1:B3 on "Sheet2" via the Excel [Calc] user interface

Excel	Select "Sheet1" and highlight cells B2:C3 then do one of the following: 1) Select the tool bar options: **Insert > Name > Define** , enter name "Sheet1!MyCells" in pop-up window and press "Add" button. 2) Enter "Sheet1!MyCells" in the Name Field on the active window. Select "Sheet2" and highlight cells A1:B3 then repeat either steps 1 or 2 from above only this time use the name "Sheet2!MyCells".
Calc	It is not possible. Calc appears not to allow the same range name, e.g. "MyCells", to exist on two or more worksheets [sheets].

Access ranges with the same name relative to worksheets [sheets]. Assume in worksheet [sheet] "Sheet1" cell B1 is named "MyCell" and contains the string "Sheet1MyCell". In worksheet [sheet] "Sheet2" cell C3 is named "MyCell" and contains the string "Sheet2MyCell".

VBA	`Rem The following will display "Sheet1MyCell"`
	`Rem followed by "Sheet2MyCell"`
	`Worksheets("Sheet1").Activate`
	`MsgBox Range("MyCell").Value`
	`Worksheets("Sheet2").Activate`
	`MsgBox Range("MyCell").Value`

SB	It is not possible. Calc appears not to allow the same range name, e.g. "MyCell", to exist on two or more worksheets [sheets].

Charts/Chart

Create a bar chart on the current worksheet [sheet] using the range "MyChartData".

VBA

```
Sub SomeProcedure
    Range("MyChartData").Select
    Charts.Add
    ActiveChart.ChartType = xlColumnClustered
    ActiveChart.Name = "Sample Chart"
    ActiveChart.SetSourceData _
       Source:=Sheets("Example3").Range("ChartData"), _
       PlotBy:= xlColumns
    ActiveChart.Location Where:=xlLocationAsObject, _
       Name:="Example3"
    With ActiveChart
        .HasTitle = True
        .HasLegend = False
        .ChartTitle.Characters.Text = "Sample Chart"
        .Axes(xlCategory, xlPrimary).HasTitle = True
        .Axes(xlCategory, xlPrimary).AxisTitle.Characters.Text =
"Category"
        .Axes(xlValue, xlPrimary).HasTitle = True
        .Axes(xlValue, xlPrimary).AxisTitle.Characters.Text =
"Amount"
    End With
End Sub
```

SB

```
Sub SomeProcedure
  'define rectangle to hold chart
   Dim aRect as new com.sun.star.awt.Rectangle
   with aRect
      .X = 8000 : .Y = 1000 : .Width = 16000 : .Height = 10000
   end with

' Now add a chart to the spreadsheet.
   'get data to chart
   oSheet = ThisComponent.CurrentController.ActiveSheet
   oCellRangeAddress = _
      oSheet.getCellRangeByName( "MyChartData" ). _
           getRangeAddress()

   ' Get the collection of charts from the sheet
   oCharts = oSheet.getCharts()

   ' Add a new chart with a specific name,
   '  in a specific rectangle on the drawing page,
   '  and connected to specific cells of the spreadsheet.
   oCharts.addNewByName( "Sample Chart", aRect , _
       Array( oCellRangeAddress ) , True, True )

   ' Get the new chart we just created.
   oChart = oCharts.getByName( "Sample Chart" )
   ' Get the chart document model.
   oChartDoc = oChart.getEmbeddedObject()
   oChartDoc.HasLegend = False
   ' Get the drawing text shape of the title of the chart.
   oChartDoc.getTitle().String = "Sample Chart"  'Change title
   ' Create a diagram.
   oDiagram = _
    oChartDoc.createInstance("com.sun.star.chart.BarDiagram")

   ' Set its parameters.
   oDiagram.Vertical = True
   ' Make the chart use this diagram.
   oChartDoc.setDiagram( oDiagram )
   oDiagram.getXAxisTitle().String = "Category"
   oDiagram.HasXAxisTitle = true
   oDiagram.getYAxisTitle().String = "Amount"
   oDiagram.HasYAxisTitle = true
   ' Make more changes to the diagram.
   oDiagram.DataRowSource = _
          com.sun.star.chart.ChartDataRowSource.COLUMNS
End Sub
```

Reference: http://api.openoffice.org/docs/DevelopersGuide/Charts/Charts.htm#1+Charts

Delete the chart created in the preceding example.

VBA	`Sub MyProc`
	
	`  ActiveSheet.Charts("Sample Chart").Delete`
	
	`End Sub`

SB	`Sub MyProc`
	
	`  ThisComponent.CurrentController.ActiveSheet. _`
	`  getCharts().removeByName("Sample Chart")`
	
	`End Sub`

Controls

This section describes placing controls, such as check boxes, option buttons, combo boxes, on a worksheet [sheet].

Create a control (Check Box, Combo Box, Option Button, Button, etc.) on the worksheet [sheet] and give the control a user defined name.

Excel	• Drag and drop the controls from the Controls menu onto the worksheet • Select the control on the worksheet and press right mouse button • Select "Properties" option • Enter user defined name into the "(Name)" property
Calc	• Drag and drop the controls from the Forms Function menu onto the sheet • Select the control on the sheet and press right mouse button • Select the "Control..." option • Select the "General" tab and enter user defined name into the "Name" property

Usage Note: The **Name** property for option buttons in Calc is used to group option buttons for selecting one and only one option from a group of options, radio button operation. This same function in Excel is accomplished by assigning the same value to the **Group** property of the option buttons.
Reference: OpenOffice Calc Help

Turn on/off Design Mode

Excel	Press Design Mode icon on Controls menu to toggle on/off
Calc	Press Design Mode icon on the Form Functions menu to toggle on/off

Reference: OpenOffice Calc Help

Assign a worksheet [sheet] cell to hold current state of the control, i.e., is the check box checked or unchecked, selected item in a combo box.

Excel	• Turn Design Mode on • Select the control on the workhseet and press right mouse button • Select **Properties** option • In the property **LinkedCell** enter a worksheet cell address (e.g., B4, $B$4) or a user defined named range (e.g., "MyControlState")
Calc	• Turn Design Mode on • Select the control on the sheet and press right mouse button • Select the **Control...** option • Select **Data** tab and enter sheet cell address (e.g.,B4) into the **Linked Cell....** property.

Usage Note: Unable to use named ranges (e.g., "MyControlState") to specify a cell location in Calc.

Assign to a list box or combo box the cell range on the worksheet [sheet] that holds the list of items to display.

Excel	• Turn Design Mode on • Select the control on the workhseet and press right mouse button • Select "Properties" option • In the property "ListFillRange" enter a worksheet cell address (e.g., B4:B6, $B$4:$B$6) or a user defined named range (e.g., "MyListOfChoices")
Calc	• Turn Design Mode on • Select the control on the sheet and press right mouse button • Select the "Control..." option • Select "Data" tab and enter sheet cell address (e.g.,B4:B6) into the "Source Cell Range...." property.

Assign multi-line caption to a button control

Excel	• Select button control on worksheet • Click right mouse button • **Properties > Word Warp > True**
Calc	Not supported

UserForms

Create a UserForm [Dialog] "MyForm"

VBA	• Start the Visual Basic IDE • Select from the tool bar **Insert > UserForm** • Select UserForm object • Right-click mouse button on selected UserForm object • Click on the **Properties** option • Enter "MyForm" in the **(Name)** attribute • Design the layout of the UserForm
SB	• Start the StarBasic IDE • Press **Organizer** button • Press **New Dialog...** button • Enter "MyForm" in the name menu for new dialog panel and press "OK" button • Select the new dialog just created with mouse pointer, press **Edit** button • Design the layout of the Dialog
***Reference**:* http://api.openoffice.org/docs/DevelopersGuide/BasicAndDialogs/BasicAndDialogs.htm#1+1+0+2+A+Simple+Dialog	

Create a control (CheckBox, ComboBox, RadioButton, Button, etc.) on the UserForm [Dialog] and give the control a user defined name.

VBA	• Drag and drop the controls from the Controls menu onto the UserForm • Select the control and press right mouse button • Select **Properties** option • Enter user defined name into the **(Name)** property
SB	• Drag and drop the controls from the Controls menu onto the Dialog • Select the control and press right mouse button • Select the **Properties...** option • Enter user defined name into the **Name** property

Group related option buttons such that only one option button can be selected.

VBA	Drag a Frame control to encompass the set of option buttons that are related.
SB	On a Dialog, option buttons are grouped by consecutive **Order** attribute. To access this attribute, select the option button, press right mouse button, select **Properties....**. Consecutive numbers in the **Order** attribute are part of one group. To designate another group, there has to be a break in the number.
***Reference**:* http://api.openoffice.org/docs/DevelopersGuide/BasicAndDialogs/BasicAndDialogs.htm#1+5+2+4+Option+Button and http://api.openoffice.org/docs/common/ref/com/sun/star/awt/XRadioButton.html	

Create TabStrip or Multi-Page Control on the UserForm [Dialog]

VBA	Drag and drop the controls from the Controls menu onto the UserForm
SB	These controls do not exist in SB.
Usage Note: While a multi-page control in the sense of Excel, where folder tabs are used to select among different pages, does not exist in Calc/SB, a possible work-around is the use of "Multi-page Dialogs". The Dialog's **Step** attribute provides the means to display different controls on the Dialog panel based on its value and the corresponding value in the **Step** attributes of the various Controls. See reference for details. ***Reference**:* http://api.openoffice.org/docs/DevelopersGuide/BasicAndDialogs/BasicAndDialogs.htm#1+5+1+6+Multi-Page+Dialogs and http://api.openoffice.org/docs/common/ref/com/sun/star/awt/UnoControlDialogElement.html	

Display UserForm [Dialog] called "MyForm"

VBA	`Sub MyProc`
	`  MyForm.Show`
	`End Sub`

SB	`Rem oDlg should be visible at the module level`
	`Dim oDlg As Object`
	
	`Sub MyProc`
	`  DialogLibraries.LoadLibrary("Standard")`
	`  oDlg = CreateUnoDialog(DialogLibraries.Standard.MyForm)`
	`  oDlg.execute()`
	`End Sub`

Usage Note: The **oDlg** variable is visible at the module level to all other procedures that are accessing controls on the Dialog. This means all the procedures manipulating or accessing controls on this Dialog panel are housed in a single module.
Reference:
http://api.openoffice.org/docs/DevelopersGuide/BasicAndDialogs/BasicAndDialogs.htm#1+5+1+1+Showing+a+Dialog

Display message "Button Clicked" when the users clicks on button "MyButton"and then disable the button.

VBA	• Select button on the UserForm
	• Right mouse click to bring up options menu
	• Select **View Code** option
	• Select **Click** event for this button control
	• In the predefined procedure MyButton_Click():
	`Sub MyButton_Click()`
	`  msgBox "Button Clicked"`
	`  MyButton.Enabled = False`
	`End Sub`

SB	• Select button "MyButton" on the Dialog
	• Right mouse click to bring up options menu
	• Select **Properties** option
	• Select **Events** tab
	• Assign user defined macro "MyButton_Click", see below, to event **When Initiating....**:
	`Rem oDlg should be visible at the module level`
	`Dim oDlg As Object`
	`Sub MyButton_Click`
	`  msgbox "Button Clicked"`
	`  oDlg.getControl("MyButton").Enable = False`
	`End Sub`

Usage Note: The **oDlg** variable is the same variable, visible at the module level, that was used when the Dialog frame was displayed. In the case of SB, the procedure name is arbitrary. In the case of VBA, the procedure name is predefined by the Excel object model.

***Reference**:* http://api.openoffice.org/docs/common/ref/com/sun/star/awt/UnoControlButtonModel.html

Determine if checkbox "MyCheckBox" has been checked.

VBA

```
Sub SomeProcedure

   if MyCheckBox.Value then

      '''Do processing for checkbox selected

   else

      '''Do processing for checkbox unselected or undetermined

   end if

End sub
```

SB

```
Dim oDlg as Object

Sub SomeProcedure

   if oDlg.getControl("MyCheckBox").State = 1 then

      '''Do processing for checkbox selected

   else

      '''Do processing for checkbox unselected or undetermined

   end if

End Sub
```

Usage Note: The **oDlg** variable is the same variable, visible at the module level, that was used when the Dialog frame was displayed. In VBA the **CheckBox** value is either **False**, **True** or **Null** (undetermined) and in SB it is a numeric value 0 (unchecked), 1 (checked) or 2 (undetermined).

Reference:
http://api.openoffice.org/docs/DevelopersGuide/BasicAndDialogs/BasicAndDialogs.htm#1+5+2+3+Check+Box and http://api.openoffice.org/docs/common/ref/com/sun/star/awt/XCheckBox.html

Initialize the ListBox named "MyListBox"

VBA

```
Sub SomeProcedure

   with MyListBox
      .addItem "Choice1"
      .addItem "Choice2"
      .addItem "Choice3"
   end with

End sub
```

SB

```
Dim oDlg as Object

Sub SomeProcedure

   with oDlg.getControl("MyListBox")
      .addItem("Choice1",0)
      .addItem("Choice2",1)
      .addItem("Choice3",2)
   end with

End Sub
```

Usage Note: The **oDlg** variable is the same variable, visible at the module level, that was used when the Dialog frame was displayed. When there are many items to load into the ListBox, the method **addItems()**, provides a faster way of loading the ListBox.
Reference:
http://api.openoffice.org/docs/DevelopersGuide/BasicAndDialogs/BasicAndDialogs.htm#1+5+2+7+List+Box and http://api.openoffice.org/docs/common/ref/com/sun/star/awt/XListBox.html

Based on the selected item in "MyListBox" do appropriate processing.

VBA

```
Sub SomeProcedure

   select case MyListBox.ListIndex
      case 0
         'Do processing for "Choice1"
      case 1
         'Do processing for "Choice2"
      case 2
         'Do processing for "Choice3"
      case else
         'Something wrong do error processing
   end select

End sub
```

SB

```
Dim oDlg as Object

Sub SomeProcedure

   select case _
      oDlg.getControl("MyListBox").SelectedText

      case "Choice1"
         'Do processing for "Choice1"
      case "Choice2"
         'Do processing for "Choice2"
      case "Choice3"
         'Do processing for "Choice3"
      case else
         'Something wrong do error processing
  end select

End Sub
```

Usage Note: The **oDlg** variable is the same variable, visible at the module level, that was used when the Dialog frame was displayed.

Reference:
http://api.openoffice.org/docs/DevelopersGuide/BasicAndDialogs/BasicAndDialogs.htm#1+5+2+7+List+Box and http://api.openoffice.org/docs/common/ref/com/sun/star/awt/XListBox.html

Initialize the ComboBox named "MyComboBox"

VBA

```
Sub SomeProcedure

   with MyComboBox
      .addItem "Choice1"
      .addItem "Choice2"
      .addItem "Choice3"
   end with

End sub
```

SB

```
Dim oDlg as Object

Sub SomeProcedure

   with oDlg.getControl("MyComboBox")
      .addItem("Choice1",0)
      .addItem("Choice2",1)
      .addItem("Choice3",2)
   end with

End Sub
```

Usage Note: The **oDlg** variable is the same variable, visible at the module level, that was used when the Dialog frame was displayed. Like the ListBox, the ComboBox has method **addItems()** to load many entries at once.

References:
http://api.openoffice.org/docs/DevelopersGuide/BasicAndDialogs/BasicAndDialogs.htm#1+5+2+8+Combo+Box and http://api.openoffice.org/docs/common/ref/com/sun/star/awt/XComboBox.html

Based on the selected item in "MyComboBox" do appropriate processing.

VBA

```
Sub SomeProcedure

   select case MyComboBox.ListIndex
      case 0
         'Do processing for "Choice1"
      case 1
         'Do processing for "Choice2"
      case 2
         'Do processing for "Choice3"
      case else
         'Something wrong do error processing
   end select

End sub
```

SB

```
Dim oDlg as Object

Sub SomeProcedure

   select case _
      oDlg.getControl("MyComboBox").Text

      case "Text for Choice1"
         'Do processing for "Choice1"
      case "Text for Choice2"
         'Do processing for "Choice2"
      case "Text for Choice3"
         'Do processing for "Choice3"
      case else
         'Something wrong do error processing
  end select

End Sub
```

Usage Note: The **oDlg** variable is the same variable, visible at the module level, that was used when the Dialog frame was displayed. Please note the difference for determining the selected choice between VBA and SB. In VBA the index value of the selected choice is used, in SB, the text of the selected item is used.

Reference: http://api.openoffice.org/docs/common/ref/com/sun/star/awt/XComboBox.html

For option buttons, "Option1", "Option2" and "Option3", determine which one is selected.

VBA

```
Sub SomeProcedure

  If Option1.Value = True Then
      ' perform Option1 tasks
  ElseIf Option2.Value = True Then
      ' perform Option2 tasks
  ElseIf Option3.Value = True Then
      ' perform Option3 tasks
  End If

End sub
```

SB

```
Dim oDlg as Object

Sub SomeProcedure

   if oDlg.getControl("Option1").getState() = True Then
      ' perform Option1 tasks
   ElseIf oDlg.getControl("Option2").getState() = True Then
      ' perform Option2 tasks
   ElseIf oDlg.getControl("Option3").getState() = True Then
      ' perform Option3 tasks
   End If

End Sub
```

Usage Note: The **oDlg** variable is the same variable, visible at the module level, that was used when the Dialog frame was displayed.

Make Textbox read-only

VBA	Select Text Box control and set property "Locked" to "True"
SB	Select text box control and set property "Ready Only" to "Yes"

Integrated Development Environment (IDE) Differences

Listed in this section are functional or appearance differences between the IDEs for VBA and SB. For the Excel/VBA programmer here is an overview of the IDE for SB: http://api.openoffice.org/docs/DevelopersGuide/BasicAndDialogs/BasicAndDialogs.htm

Starting the IDE

VBA	From the menu bar (**Tools > Macro > Visual Basic Editor**) or with the short-cut key Alt-F11.
SB	From the menu bar (**Tools > Macros > Macro**). There is no short-cut key stroke to start the IDE.

Create a VBA [SB] Module

VBA	• From the Project Window select the appropriate workbook • **Insert > Module**
SB	• From the Macro Dialog select the **Standard** library of the appropriate spreadsheet • Press **New** button

Reference:
http://api.openoffice.org/docs/DevelopersGuide/BasicAndDialogs/BasicAndDialogs.htm#1+1+0+1+Step+By+Step+Tutorial

Encountering a breakpoint during execution of a macro initiated from the workbook [spreadsheet]

VBA	The IDE Editor window is automatically opened and there is a small yellow area pointing to the line that triggered the breakpoint and the line is highlighted.
SB	The IDE Editor window is not automatically opened. The programmer must manually open the window. There is a small yellow arrow pointing to the line that triggered the breakpoint.

Finding and replacing text in a module

VBA	The programmer is able to specify the scope of the find & replace, e.g., the module or only the selected text range in the module.
SB	The scope of the find & replace is the current module.

Differences in the Object Viewer

VBA	Able to view properties and methods for Excel and user defined objects. Information available through the object viewer are return values from methods, method signature, properties and MS/Office defined constants.
SB	No equivalent functionality

Prompting with object attributes

VBA	When typing an object in the Editor, the editor will provide a selection of the specific object properites.
SB	No equivalent functionality

View object and object properties while debugging using breakpoints

VBA	Specify object or object property in Watch window
SB	No equivalent functionality

Usage Note: There is an add-in macro library called XRay that provides the function of viewing objects and object properties. See Appendix A for an overview. This macro library can be found at http://www.ooomacros.org/dev.php101416

Prompting with object attributes

VBA	When typing an object in the Editor, the editor will provide a selection of the specific object properites.
SB	No equivalent functionality

Print debugging information

VBA	**debug.print** statement prints data to the Immediate window in the VBA IDE
SB	No Immediate window and no **debug.print** available in the SB IDE

Porting Sample Workbook [Spreadsheet]

In this chapter, the steps taken to port an Excel workbook to Calc are described. The reader should be aware that the approaches discussed are one of several ways of performing the porting task.

To illustrate the differences between VBA and SB, the original VBA code is left in the module as comment statements ("Rem" statement). The equivalent SB statements follow the commented VBA code.

Porting Tasks

Steps taken to port the sample Excel/VBA workbook to Calc/SB spreadsheet.

Open Excel workbook in Calc. For the most part the general appearance of worksheets will be the same in Calc. The following differences were noted after opening the Excel workbook in Calc.

- On several worksheets the caption for the buttons do not appear the same. In Excel buttons can be multi-lined. In Calc, multi-line captions do not appear to be supported.
- While all the VBA code was read in, all the statements were turned into comments by prefixing "Rem" to each line. For existing VBA Modules, those modules exist in SB under the same name, e.g., "ChartDemocode" and "SampleCode". For VBA code associated with the workbook or worksheets, such as those executed by events or controls on the worksheet, the code is contained in separate modules, one for each worksheet or for the workbook. The workbook code is contained in a SB module called "ThisWorkbook". Basic code for each worksheet is contained in a separate module named after the internal Excel worksheet name. In this case the modules are called "Sheet1", "Sheet2", etc. VBA code associated with the UserForm is contained in a module named the same as the Excel Userform name. In this situation the module name is "UserForm1" Lastly, the "Rem"ed statements are encased within a procedure definition that is named the same as the module name.
- The UserForm panel itself did not transfer over. The UserFrom will have to be recreated in a SB Dialog panel.
- All named cell ranges transferred over subject to the limitation of not being able to have a same named cell in multiple worksheets.
- The text boxes on the various worksheets in Excel are transferred over to Calc. However, not all the resulting objects in Calc allow the text to be modified. As of this writing, no determination has been made on why some of the resulting text boxes allow modifications and others do not.

Convert the workbook related procedures. These procedures are found in the module "ThisWorkbook". Following are the steps to convert procedures in this module:

- Remove the encapsulating **Sub ThisWorkbook** (first statement) and **End Sub** (last statement) statements. These are the encasing procedure statements that are automatically inserted by Calc when reading in the Excel workbook.

- Procedure **Workbook_Open()** - Convert VBA Worksheet and Cell/Range methods/properties to equivalent SB constructs. Remove Private attribute from the Workbook_Open() procedure definition statement. Leaving the Private attribute causes intermittent run-time errrors. Assign the **Open Document** event to this macro. With SB IDE active, select the module "ThisWorkbook". The select **Assign... > Events**. Select the **Document** option button. Specify the **Workbook_Open** procedure for the **Open Document** event.
- Add to **Workbook_Open** procedures to turn on or off Listener for change events to the **ActiveSheet** property of the **ThisComponent.CurrentController**. This event is used to call procedures for **Worksheet_Activate** and **Worksheet_Deactivate** for various worksheets.
- Create a new module called "SupportModule". Add **MoveCursorToEnd()** function described in Appendix B to "SupportModule".

Convert worksheet Example1. Rename the module for the associated SB code from "Sheet1" to "Example1Code". Rationale for renaming the module is to support long-term maintenance by clearly associating the SB code with the worksheet [sheet]. One disadvantage of this method is if the worksheet [sheet] is renamed, the programmer must remember to rename the SB module as well to maintain the association. To rename the module, select the module tab displayed in the SB IDE, right-click the mouse button and enter the new name.

- Remove the encapsulating **Sub Sheet1** and **End Sub** statements.
- Uncomment **Sub Worksheet_Activate** and associated **End Sub** statements. Remove the Private attribute for this procedure. Leaving the Private attribute causes intermittent run-time errors.
- Replace the Excel/VBA **Range()** method calls with the equivalent Calc/SB calls.
- Unable to determine SB code to monitor cell selection changes on this worksheet [sheet].

Convert worksheet Example2. These are the steps taken in the SB module "SampleCode":

- Delete the **Sub SampleCode** and **End Sub**.
- In the procedure "generateDataToSort", convert the Excel/VBA Range objects to the equivalent SB CellRange objects. On the sheet "Example2", assign the **When Initiating** events of the two button controls for generating random sort data to this macro procedure.
- In the procedures "SortWithScreenUpdating" and "SortWithNoScreenUpdating", convert the **ScreenUpdating=True** and **ScreenUpdating=False** to **UnlockControllers** and **LockControllers** method invocations, respectively. Convert the Excel Range and Cell object references to Calc CellRange object references. Assign the **When Initiating** events for the buttons initiating the sort to the appropriate procedures. Manually adjust button sizes to fit text in the caption [label].
- In the procedure "BubbleSort" convert the procedure definition to change reference from Excel Range object to generic object type in Calc. Replace references to the Excel **Interior.ColorIndex** attribute (cell background color) with the Calc **BackColor** attribute for a cell. In addition translate specific Excel **ColorIndex** values to SB calls to the **RGB()**

function to specify colors.

Convert Worksheet Example3. This SB module underlying this worksheet is module "Sheet9". These are the steps taken to convert this module:

- Rename SB Module from "Sheet9" to "Example3Code".
- Remove the encapsulating **Sub Sheet9** and **End Sub** statements.
- Resize the button to fit the button caption.
- Cell G10 contains text "Range Selected". In Excel, this cell is right aligned and the text string is completely visible. In Calc, while the cell is right aligned, only the leftmost portion of the text string is shown with a small red arrow visible on the right-side of the cell. To correct this problem, cells F10 & G10 are merged. The text "Range Selected" is reentered and right aligned in the merged cell.
- Modify the Excel Range method calls to the equivalent SB calls.
- Add user developed function **CellRangeAddressString()** to "SupportModule" to extract a string representation of a cell or range. This function is described in Appendix B.

Convert worksheet Example4:

- Remove the **Sub ChartDemoCode** and **End Sub** statements.
- Uncomment **Sub GenerateChart** and **End Sub** statements.
- Add SB statements for defining a chart. Since Calc/SB does not automatically provide default size for a chart, experiment with several different values for the Rectangle object (**aRect**).
- Connect the button on worksheet Example4 to the procedure "GenerateChart" by selecting the button, right-click mouse button, **Control... > Events** and enter into the **When Initiating....** property the procedure "GenerateChart" found in module "ChartDemoCode".

Convert worksheet Example5:

- Rename SB module Sheet3 to Example5Code and remove the encapsulating **Sub Sheet3** and **End Sub**
- Split the VBA Worksheet_Change() procedure into separate procedures for each distinct cell ranges to monitor for cell content changes. This results in two new procedures MYCELL_modified and MYVECTOR_modified to listen for modification events. The procedures are named based on the names for the cell ranges.
- Create procedure Worksheet_Activate to register the listener procedures above.
- Create procedure Worksheet_Deactivate to remove the listener procedures above.
- Add code to Workbook_Open procedure in module "ThisWorkbook" to call the **Sub Worksheet_Activate** or **Sub Worksheet_Deactivate** procedures.
- In module "SampleCode", modify procedure "ElementOperation" to call Calc sheet functions. Assign event **When Initiating....** for button on worksheet to procedure "ElementOperation".

Convert Worksheet Example6:

- For the ComboBox. To re-establish the ComboBox, select it and right-click the mouse. Select **Control... > Data**. Enter into property **Linked Cell....** the cell location to display the selected choice. In Calc, only the cell address can be entered, i.e., D16. User defined range names, such as "SelectedChoice", are not allowed. To connect the ComboBox to the list of choices to display enter the cell range address, F16:F18, in the property **Source cell range.....**
- To convert the option buttons, for each button, select it and right-click the mouse. Select **Control... > Data**. Enter into property **Linked Cell...** the cell location to the display state of the option button. Again the cell address, e.g., D12, must be used since user-defined range names, e.g., "StateOfOption1", are not supported. Note that the **Name** property (**Control... > General**) must be the same for all the option buttons for them to function as radio buttons, i.e., only one button can be selected at a time.
- To convert the CheckBox, select it and right-click the mouse. Select **Control... > Data** and enter the cell address, e.g., D7, to hold the state of the CheckBox. As in the other controls, user defined range names are not supported.
- Rename SB module "Sheet2" to "Example6Code". Remove the encapsulating **Sub Sheet2** and **End Sub** statements and uncomment all the SB statements. No other source code modification is needed. Select the button on the worksheet, right-click the mouse and select **Control... > Events**. In the **When Initiating...** property, enter the procedure "CommandButton1_Click" found in this module. Remove the **Private** attribute to the CommandButton1_Click procedure definition.

Convert worksheet Example7. The most important step in converting this worksheet is rebuilding the UserForm as a Calc Dialog panel. The UserForm itself is not imported into Calc from Excel.

- Rename SB module "UserForm1" to "UserForm1Code". Remove the encapsulating "Sub UserForm1" and "End Sub" statements.
- Create dialog panel. Name the panel "UserForm1".
- Add controls on the dialog panel to match the Excel UserForm.
- Create procedure "Button_Click_To_Show_UserForm" in "UserForm1Code" module. This facilitates access to the shared global object variable representing the UserForm. Assign **When Initiating** event to procedure "Button_Click_To_Show_UserForm" to display the UserForm.

The following table summarizes the results of porting ExcelExamples.xls to PortedExcelExamples.sxc:

Excel Component	Result of Porting
Workbook	Able to port functionality. However, required additional coding to handle events.

Excel Component	Result of Porting
Worksheet Example1	Able to port a subset of the functions demonstrated on this worksheet. Unable to duplicate function of tracking cursor movement on this worksheet [sheet].
Worksheet Example2	Able to port all functions on this worksheet [sheet]..
Worksheet Example3	Able to port all functions on this worksheet [sheet].
Worksheet Example4	Able to port all functions on this worksheet [sheet].
Worksheet Example5	Able to port all functions on this worksheet [sheet]. Additional coding needed to support handing events on this worksheet [worksheet].
Worksheet Example6	Able to port all functions on this worksheet [sheet].
Worksheet Example7	Able to port all functions on this worksheet [sheet]. Needed to manually recreate the UserForm [Dialog].

Run-time Experiences

Porting and testing of the workbook [spreadsheet] was accomplished on the Windows/XP platform. After porting on Windows was complete, the workbook [spreadsheet] was tested on Linux. This section describes platform specific experiences of testing the ported workbook [spreadsheet].

The Windows/XP environment:

- Operating System: Windows/XP Home with Service Pack 1
- OpenOffice: OpenOffice.org 1.1.1

The Linux environment:

- Operating System: SuSe Linux 9.1 (running under Vmware 4.5.1 on Windows/XP)
- OpenOffice: OpenOffice.org 1.1.1

After testing PortedExcelExamples.sxc on Windows, the workbook [spreadsheet] was transferred to the Linux environment. The workbook [spreadsheet] performed the same on Linux with the following exception:

- Although the button controls on the worksheets [sheets] were adjusted to display all the text in the control's caption [label] for Windows, the text did not fit completely in the buttons on OpenOffice on Linux. One way to avoid this problem is to specifically set the font and character set size of the control and not just take the default specification when the control is created. This specification is set through the **Character Set** property of the control.

Appendix A: XRay tool

Unlike the Excel/VBA environment, the documentation for what a Calc object can or cannot do is spread among several locations. In addition, the debugger found in the SB IDE is not capable of displaying the structure of an object. Fortunately, a tool, freely available on the Intertnet, is available to allow a programmer to explore the Calc objects during run-time.

While the XRay tool has many useful features, the one disadvantage of the tool is that code must be inserted into the application to invoke XRay. Based on testing as of this writing, the inserted XRay code does not appear to affect the running of the SB program when the XRay tool is not activate. However, the XRay statements must be removed or commented out to avoid run-time errors for systems where the XRay tool is not installed.

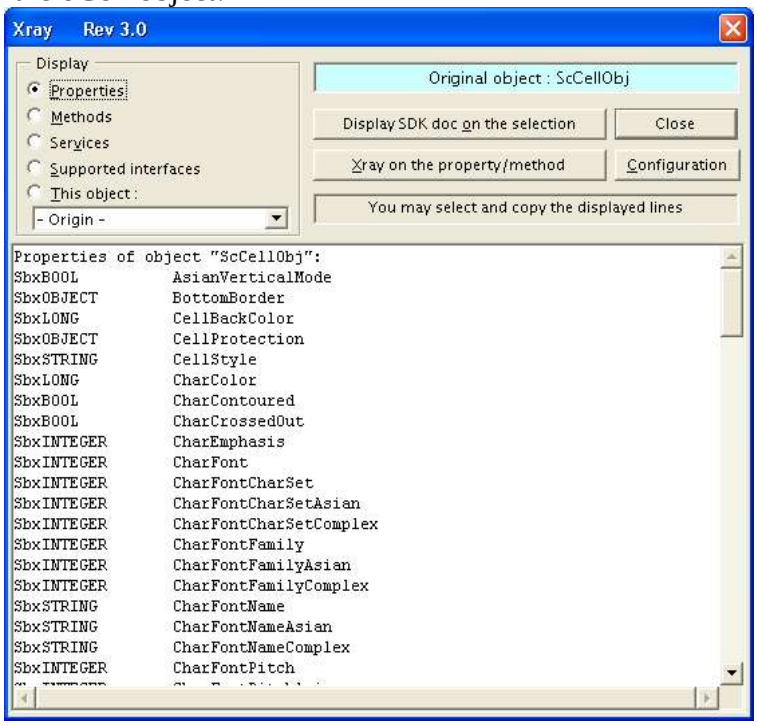
Details on the use of XRay can be found at ooomacros.org http://www.ooomacros.org/dev.php101416 .

Once the tool is installed and activated, XRay provides the following capabilities. For purposes of this illustration, assume the following was coded:

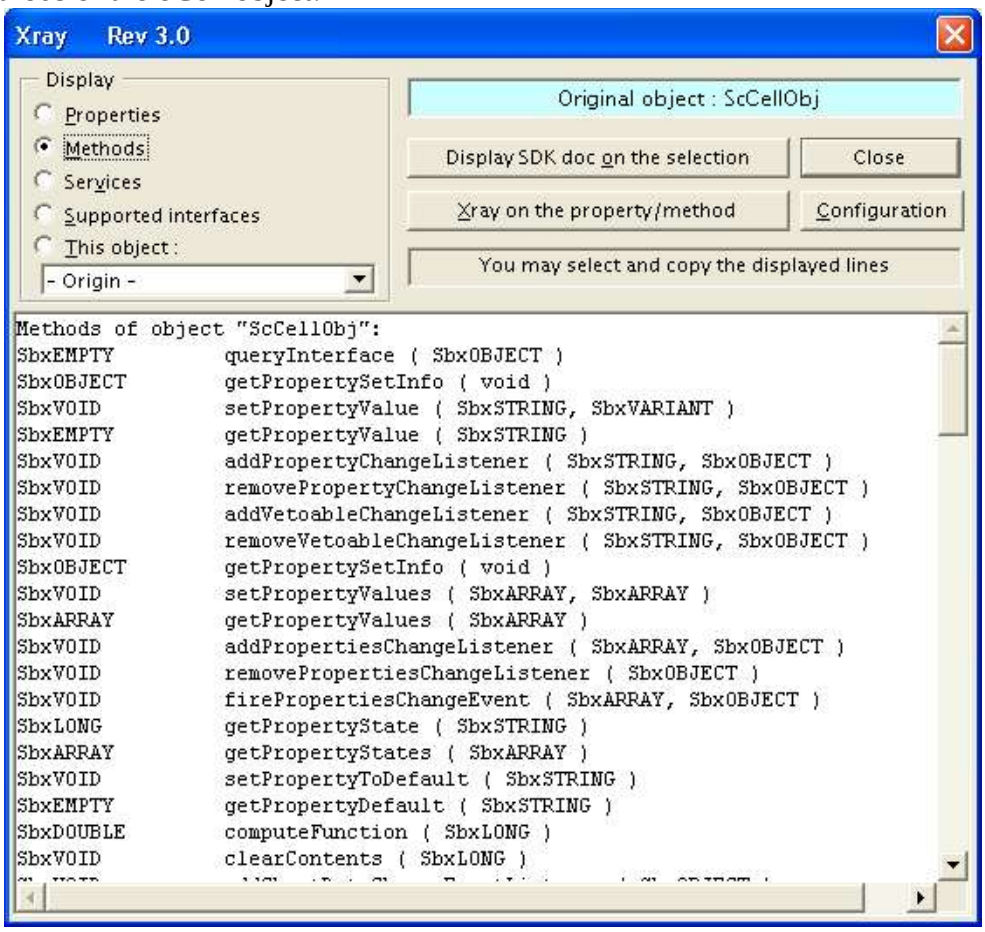
```
Dim oSheet as Object, oCell as Object

oSheet = ThisComponent.CurrentController.ActiveSheet
oCell = oSheet.getCellRangeByName("B3:E15")
Xray.XRay oCell   'this invokes the XRay tool for the oCell Object
```

XRay is able to show the following:

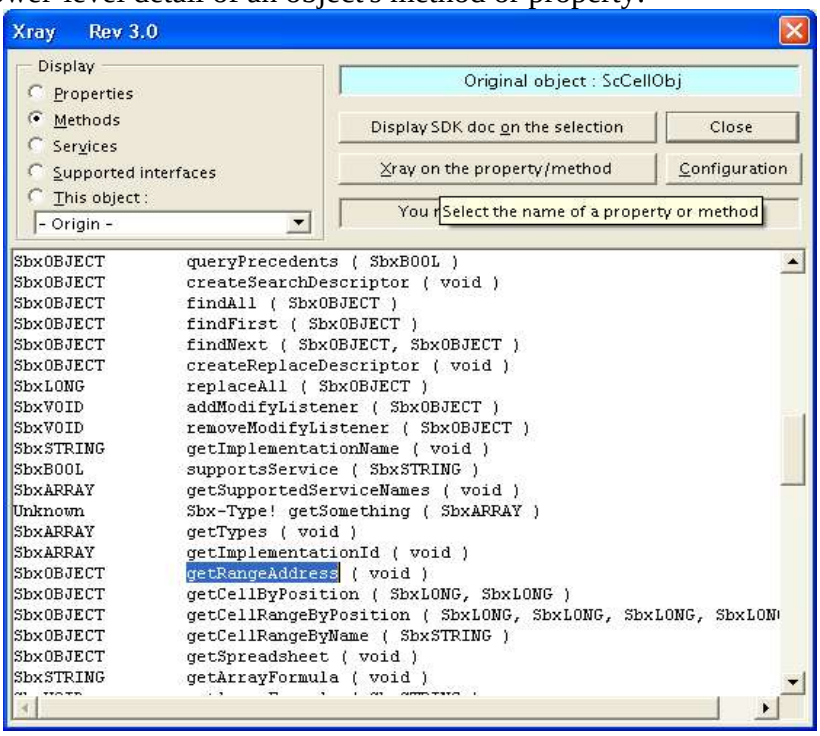
- Properties of the oCell object:

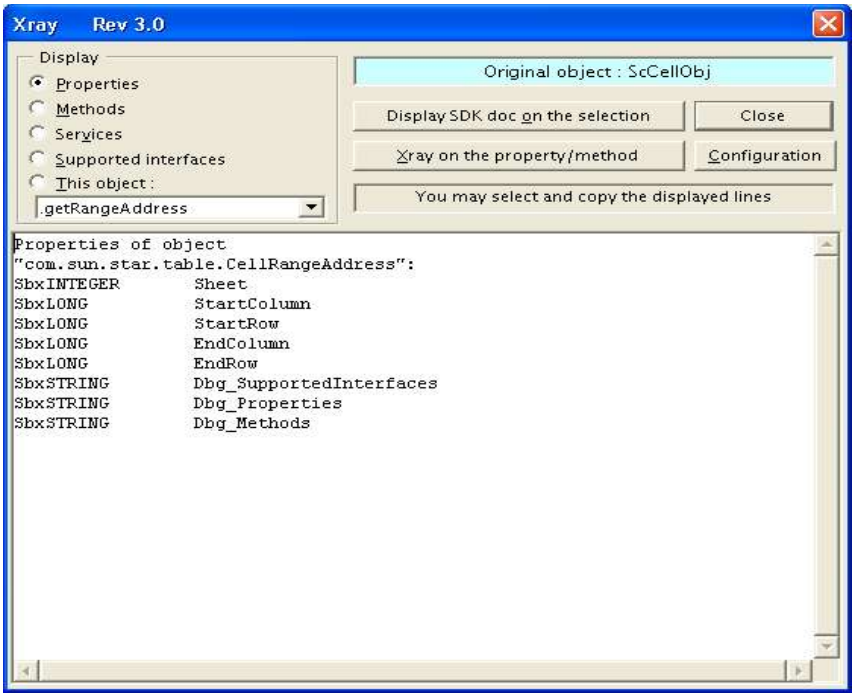


- Methods of the oCell object:

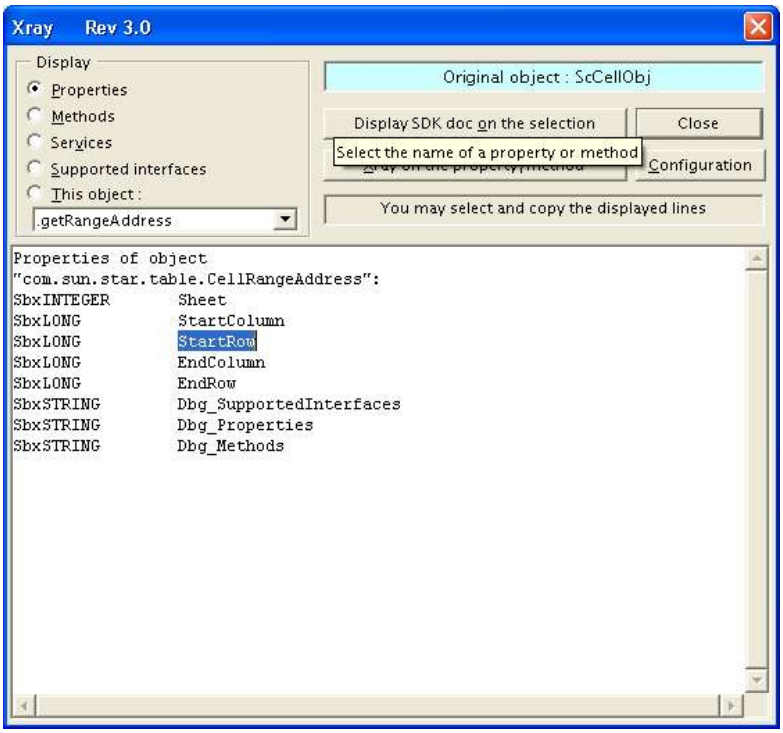


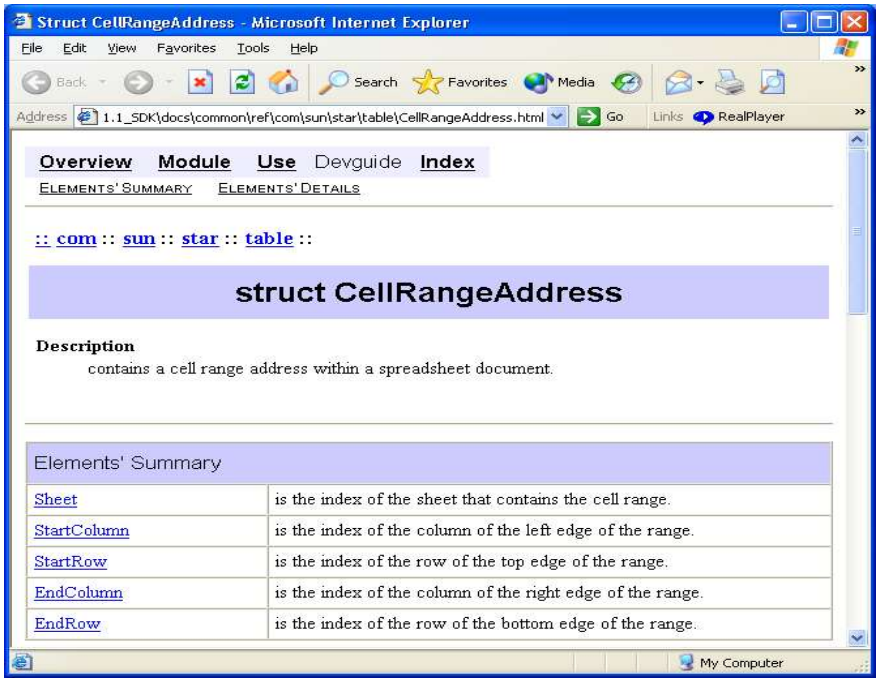
- Display lower-level detail of an object's method or property:





- Display SDK documentation for a method or property:





Appendix B: Supporting Functions

This Appendix contains user defined functions to support the porting of VBA to SB.

Functions contained in this section:

- CellRangeAddressString
- findSheetIndex
- MoveCursorToEnd

Function CellRangeAddressString(oCellRng)

This function creates a string for the addresss of a cell or range object.

Parameters:

oCellRng – Object reference to cell or range object

```
function CellRangeAddressString(oCellRng as Object) as String
    Dim FuncService
    Rem Create service to access sheet functions
    FuncService = _
       createunoservice("com.sun.star.sheet.FunctionAccess")

    select case oRng.getImplementationName()
       case "ScCellObj"
          CellRangeAddressString = FuncService.CallFunction( _
             "ADDRESS", _
          array(oCellRng.CellAddress.Row+1, _
              oCellRng.CellAddress.Column+1))

          case "ScCellRangeObj"
             CellRangeAddressString = FuncService. _
                 CallFunction("ADDRESS", _
                array(oCellRng.RangeAddress.StartRow+1, _
                      oCellRng.RangeAddress.StartColumn+1))
             CellRangeAddressString = CellRangeAddressString _
                    & ":" & FuncService.CallFunction( _
                       "ADDRESS", _
                       array(oCellRng.RangeAddress.EndRow+1, _
                             oCellRng.RangeAddress.EndColumn+1))

    end select

 End Function
```

Function findSheetIndex(SheetName)

Function to find collection index for a worksheet [sheet]. The function will return either the index value or -1 if the sheet is not found

Parameters:

SheetName – string containing name of worksheet to find

```
Function findSheetIndex(SheetName as String) as Integer
    dim i as integer

    for i = 0 to ThisComponent.Sheets.Count - 1
       if ThisComponent.Sheets.getByIndex(i).Name = _
            SheetName then
            findSheetIndex = i
            exit function
        end if
    next i

    findSheetIndex = -1
End Function
```

Function MoveCursorToEnd(pCellRange, pDirection)

The following function "MoveCursorToEnd" moves the cursor to the start or end of a row or column of data in a worksheet [sheet]. This is analogous to the "End()" method for the Excel Range object. While analogous, this function is not semantically equivalent to the "End" method. When there are empty cells in between the start location and end location, the MoveCursorToEnd function behaves differently.

Parameters:

pCellRange – object reference to starting location of cursor.

pDirection – string parameter specify direction of movement. Valid values are "xlToRight", "xlToLeft", "xlDown", "xlUp".

```
function MoveCursorToEnd(pCellRange as Object, _
       pDirection as String) as Object
  Dim StartContentType as Long, nRow as Long, nColumn as Long
  Dim ThisSheet as Object, StartRow as Long, StartColumn as Long
  Dim oCell as Object
  Dim EMPTYCELLTYPE

  ThisSheet = pCellRange.SpreadSheet
  nRow = pCellRange.CellAddress.Row
  StartRow = nRow
  nColumn = pCellRange.CellAddress.Column
  StartColumn = nColumn

  StartContentType = pCellRange.getType()
  EMPTYCELLTYPE = com.sun.star.table.CellContentType.EMPTY

  select case pDirection
     '''find last cell in the current row
     case "xlToRight"
          nColumn = nColumn + 1
          do while nColumn <= 255
             if (StartContentType <> EMPTYCELLTYPE and _
```

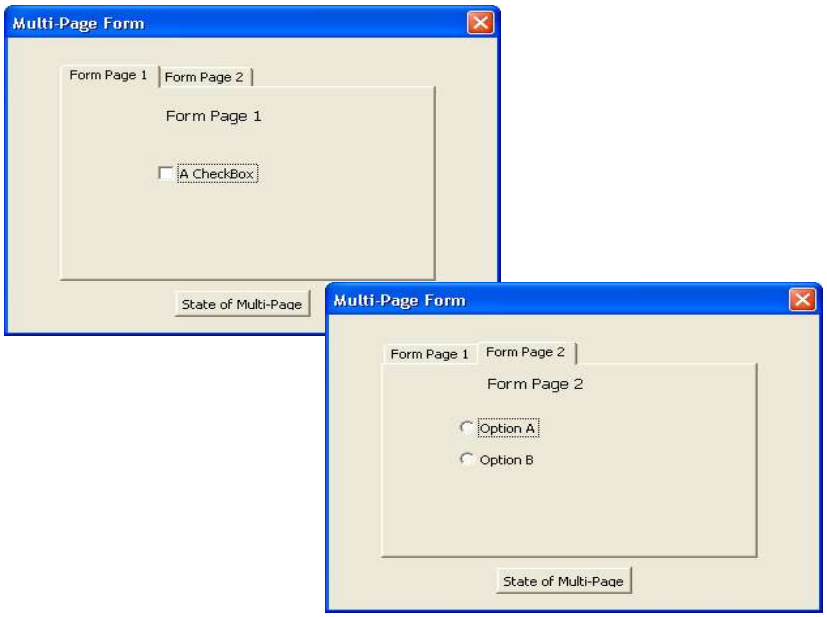
```
                ThisSheet.getCellByPosition(nColumn,nRow).getType() _
                    = EMPTYCELLTYPE) then
                nColumn = nColumn - 1
                exit do
             elseif (StartContentType = EMPTYCELLTYPE and _
                  ThisSheet.getCellByPosition(nColumn,nRow).getType() _
                     <> EMPTYCELLTYPE) then
                exit do
             end if
             nColumn = nColumn + 1
          loop

   '''find first cell in current row
   case "xlToLeft"
          nColumn = nColumn - 1
          do while nColumn >= 0
             if (StartContentType <> EMPTYCELLTYPE and _
                 ThisSheet.getCellByPosition(nColumn,nRow).getType() _
                     = EMPTYCELLTYPE) then
                nColumn = nColumn + 1
                exit do
             elseif (StartContentType = EMPTYCELLTYPE and _
                  ThisSheet.getCellByPosition(nColumn,nRow).getType() _
                      <> EMPTYCELLTYPE) then
                exit do
             end if
             nColumn = nColumn - 1
          loop

   '''find last (bottom) cell in current column
   case "xlDown"
          nRow = nRow + 1
          do while nRow <= 31999
             if (StartContentType <> EMPTYCELLTYPE and _
                ThisSheet.getCellByPosition(nColumn,nRow).getType() = _
                     EMPTYCELLTYPE) then
                nRow = nRow - 1
                exit do
             elseif (StartContentType = EMPTYCELLTYPE and _
                     ThisSheet.getCellByPosition(nColumn,nRow). _
                         getType() <> EMPTYCELLTYPE) then
                exit do
             end if
             nRow = nRow + 1
          loop

    '''find first (top) cell in current column
    case "xlUp"
          nRow = nRow - 1
          do while nRow >= 0
             if (StartContentType <> EMPTYCELLTYPE and _
                  ThisSheet.getCellByPosition(nColumn,nRow).getType() _
                     = EMPTYCELLTYPE) then
                nRow = nRow + 1
                exit do
             elseif (StartContentType = EMPTYCELLTYPE and _
```

```
                    ThisSheet.getCellByPosition(nColumn,nRow).getType() _
                         <> EMPTYCELLTYPE) then
                   exit do
               end if
               nRow = nRow - 1
            loop

  end select
        '''make sure we are in bounds
        if nColumn > 255 then
           nColumn = 255
        end if
        if nColumn < 0 then
          nColumn = 0
        end if
        if nRow > 31999 then
           nRow = 31999
        end if
        if nRow < 0 then
          nRow = 0
        end if

        MoveCursorToEnd = ThisSheet.getCellByPosition(nColumn,nRow)


end function
```

Appendix C: Multi-Page Control

This appendix will walk the reader through creating and programming a Userform [Dialog] that contains the equivalent of an Excel/VBA Multi-Page control.

The following shows an Excel/VBA UserForm containing a Multi-Page control. Depending on the tab selected by the user, different information is shown on the UserForm.



While Calc/SB does not, as of this writing, have a native Multi-Page control, it is possible to craft equivalent functionality using the existing controls available in Calc/SB. The key feature that allows this is each Control and Dialog panel itself have a property called **Step**. When the **Step** value of a Control matches the **Step** value of the Dialog panel, that control is visible, otherwise the Control is invisible.

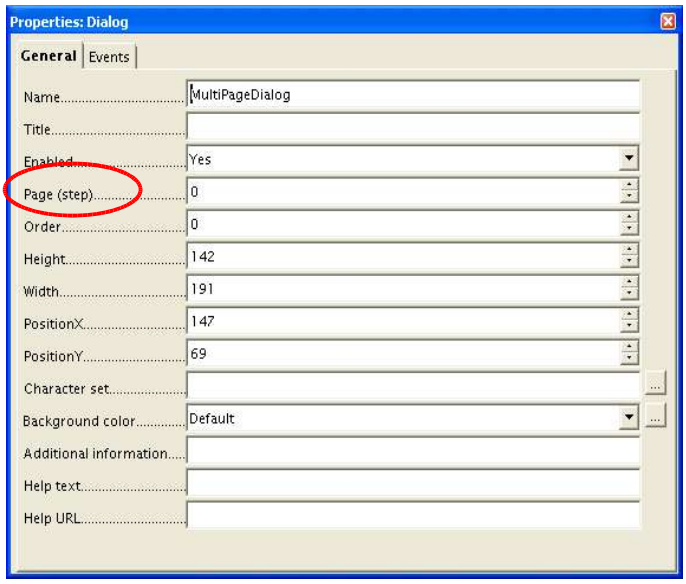
The **Step** property takes on values 0, 1, 2, ... up to the maximum value of a Long variable. **Step** value 0 signifies that the Control is visible at all times, regardless of the **Step** value of the Dialog panel.

Note: A sample spreadsheet "Multi-page Form.sxc" demonstrates the concepts discussed below.

To build the same functionality shown above, perform the following steps:

1. Create a Dialog panel **Tools > Macros > Macro > Organizer... > New Dialog**. Specify Dialog panel name, e.g., "MultiPageDialog".
2. Select **Edit** to edit the newly created Dialog panel
3. Select the **Dialog** panel, press right-click the mouse, select **Properties...**. This will bring

up the following window:



Ensure the **Page (step)** property is set to 0. *Usage Note*: Selecting the **Dialog** panel requires clicking the left mouse button when the mouse pointer is on the border (edge) of the **Dialog** panel.

4. Place a **Group Box** Control on the Dialog panel to define the area where the multiple pages will display. Clear the **Label** property for this Control.
5. Place two **CommandButton** Controls, adjacent to each other, at the outside, top left of the **Group Box** Control to function as the Tabs. Change the **Label** property for the first (leftmost) **CommandButton** to "Form Page 1" and the **Name** property to "TabForPage1". For the second button change the **Label** and **Name** properties to "Form Page 2" and "TabForPage2", respectively. Size each button as needed.
6. Place a single **CommandButton** at the bottom of the **Dialog** Panel, outside of the **Group Box** Control and set its **Label** and **Name** property to "State Of Multi-Page" and "CommandButton1", respectively, and size as needed.
7. Select the **Dialog** Control, press right-click mouse and select **Properties...**. Change **Page (step)** property to "1". Now place the Controls for "Form Page 1" within the **Group Box** Control. In this example, this is the**CheckBox** Control. Note: The **Page (step)** property for any Control placed on the **Dialog** panel will take the value of the **Page (step)** proprerty of the **Dialog** at the time the Control is created.
8. Select the **Dialog** Control and change the **Page (step)** property value to "2". Now place the two **OptionButton** Controls within the **Group Box** Control for "Form Page 2". Ensure the **Order** property for the two **OptionButton** buttons are consecutive numbers, ensure that only of the **OptionButton** buttons can be selected at a time.

9. Create a SB module called "MultiPageDialogCode" and add the following SB procedures to the module.

```
Dim oDialog as Object  'Module level variable for the Dialog

Rem Procedure to initialize the Dialog and various controls for
Rem MultiPage operation
Sub Main
  'Initialize Dialog object
  DialogLibraries.LoadLibrary("Standard")
  oDialog = createUnoDialog(DialogLibraries.Standard.Dialog1)

  'Initialize Controls on Dialog panel and display Dialog
  with oDialog
      'Button is selected
      .getControl("TabForPage1").Model.State = 1

      'Button is not selected
      .getControl("TabForPage2").Model.State = 0

      'Initialize Step property of the Dialog
      .Model.Step = 1

      'Display the Dialog
      .execute()
  end with

End Sub

Rem Handle clicking of button TabForPage1
Sub Page1_Button_Click
    with oDialog

         'Button for Page 1 selected
        .getControl("TabForPage1").Model.State = 1

        'Button for Page 2 not selected
        .getControl("TabForPage2").Model.State = 0

        'Dialog to display Page 1 Controls
        .Model.Step = 1
    end with
End Sub

Rem Handle clicking of button TabForPage2
Sub Page2_Button_Click
    with oDialog
        'Button for Page 1 not selected
        .getControl("TabForPage1").Model.State = 0

        'Button for Page 2 selected
        .getControl("TabForPage2").Model.State = 1

        'Dialog to display Page 2 Controls
        .Model.Step = 2
    end with
```

```
End Sub

Rem Handle click on CommandButton1
Sub Button1_Click
    With oDialog

        'Determine the Step property for the Dialog
        select case .Model.Step

            'Dialog in Step 1 (Page 1 displayed)
            case 1
                msgbox "Page 1 is active, Check Box is " & _
                    .getControl("CheckBox1").Model.State

            'Dialog in Step 2 (Page 2 displayed)
            case 2
                msgbox "Page 2 is active, Option A is " & _
                    .getControl("OptionButton1").Model.State & _
                    ", Option B is " & _
                    .getControl("OptionButton2").Model.State

        end Select

    End With

End Sub
```

10. Assign "When Initiating" event for the **CommandButton**s to the appropriate SB procedure.

CommandButton	Assigned To SB Procedure
TabForPage1	Page1_Button_Click
TabForPage2	Page2_Button_Click
CommandButton1	Button1_Click

After completing the above steps, test the "MultiPageDialog" by executing the "Main" procedure.

Final Tips & Tricks:

- After creating the **Dialog**, to display the different pages, change the **Page (step)** property of the **Dialog** to 1 or 2 as desired.
- To move a Control, e.g., **CommandButton, CheckBox,** etc., from one page to another page, select that Control, right-click mouse, select **Properties...** and alter **Page (step)** property to match the value for the other page.
- An alternative to using **CommandButtons** to identify the page to display, a **ListBox,** with the **DropDown** property set to **True**, can be used. In this situation, the page labels are loaded into the **ListBox** entries (e.g., "Form Page 1"). Then based on the entry selected in the **ListBox** the **Page (step)** property for the **Dialog** is set to the appropriate value. See

sample code below.

```
Dim oDialog as Object

Rem Initalize ListBox and Display the Dialog
Sub Main
  DialogLibraries.LoadLibrary("Standard")
  oDialog = createUnoDialog(DialogLibraries.Standard.MultiPageDialog)

  'This assumes the following:
  ' ListBox Position 0 = "Form Page 1" (Step 1)
  ' ListBox Position 1 = "Form Page 2" (Step 2)
  with oDialog
    'Initialize ListBox
    with .getControl("ListBox1")
       .addItem("Form Page 1",0)
       .addItem("Form Page 2",1)
       .selectItemPos(0,True)
    end with
    .Model.Step = 1
    .execute()
  end with
End Sub

Rem Assign "When Initiating" Event for ListBox1 to this
Rem procedure.
Sub ListBox1_Selected_Item
   with oDialog
       .Model.Step = .getControl("ListBox1").SelectedItemPos + 1
   end with

End Sub
```

- Another example for simulating the MultiPage function can be found in the spreadsheet MyDataPilot.sxcdeveloped by Ian Laurenson (http://homepages.paradise.net.nz/hillview/OOo/MyDataPilot.sxc). . It also has the equivalent of a RefEdit control (a control for selecting a range on a spreadsheet from within a dialog), text fields which only allow valid characters to be typed in, and a simple tree control.
- It is possible to have the one routine called by events for more than one control, to know which control called the event use an event parameter as shown is this example that could be used as the associated code for the initiate event for the TabForPage button controls in the above example:

```
Sub Page_Buttons_Click(oEvent)
  with oDialog
    'Change state of all tab controls
    for i = 1 to 2
      .getControl("TabForPage" & i).Model.State = 0
    next
    'Change the state of the one that was pressed
    oEvent.source.model.state = 1
```

```
      'Change the page to corresponding tab control
      'This assumes there are no more than 9 pages
      .Model.Step = val(right(oEvent.source.model.name, 1))
   end with
End Sub
```

Bibliography

OpenOffice.org, *Developer's Guide*, August, 26, 2003, http://api.openoffice.org/DevelopersGuide/DevelopersGuide.html

Pitonyak, A., *Useful Macro Information For OpenOffice*, May 4, 2004, http://www.pitonyak.org/AndrewMacro.sxw

Strome, D., *How to Use BASIC Macros in OpenOffice.org,* October 15, 2002, http://documentation.openoffice.org/HOW_TO/various_topics/How_to_use_basic_macros.sxw

Sun MicroSystems, *StarOffice 7 Software Basic Programmer's Guide*, July 2003, http://docs.sun.com/db/doc/817-1826?q=StarOffice

Sun Microsystems, *Migrating from Microsoft Office to StarOffice 7*, January 2004, http://se.sun.com/edu/staroffice/so_migration_guide_0104.pdf

Sun Microsystems, *star module*, 2003, http://api.openoffice.org/docs/common/ref/com/sun/star/module-ix.html

Public Documentation License, Version 1.0

1.0 DEFINITIONS.

1.1. "Commercial Use" means distribution or otherwise making the Documentation available to a third party.

1.2. "Contributor" means a person or entity who creates or contributes to the creation of Modifications.

1.3. "Documentation" means the Original Documentation or Modifications or the combination of the Original Documentation and Modifications, in each case including portions thereof.

1.4. "Electronic Distribution Mechanism" means a mechanism generally accepted for the electronic transfer of data.

1.5. "Initial Writer" means the individual or entity identified as the Initial Writer in the notice required by the Appendix.

1.6. "Larger Work" means a work which combines Documentation or portions thereof with documentation or other writings not governed by the terms of this License.

1.7. "License" means this document.

1.8. "Modifications" means any addition to or deletion from the substance or structure of either the Original Documentation or any previous Modifications, such as a translation, abridgment, condensation, or any other form in which the Original Documentation or previous Modifications may be recast, transformed or adapted. A work consisting of editorial revisions, annotations, elaborations, and other modifications which, as a whole represent an original work of authorship, is a Modification. For example, when Documentation is released as a series of documents, a Modification is:

A. Any addition to or deletion from the contents of the Original Documentation or previous Modifications.

B. Any new documentation that contains any part of the Original Documentation or previous Modifications.

1.9. "Original Documentation" means documentation described as Original Documentation in the notice required by the Appendix, and which, at the time of its release under this License is not already Documentation governed by this License.

1.10. "Editable Form" means the preferred form of the Documentation for making Modifications to it. The Documentation can be in an electronic, compressed or archival form, provided the appropriate decompression or de-archiving software is widely available for no charge.

1.11. "You" (or "Your") means an individual or a legal entity exercising rights under, and complying with all of the terms of this License or a future version of this License issued under Section 5.0 ("Versions of the License"). For legal entities, "You" includes any entity which controls, is controlled by, or is under common control with You. For purposes of this definition, "control" means (a) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (b) ownership of more than fifty percent (50%) of the outstanding shares or beneficial ownership of such entity.

2.0 LICENSE GRANTS.

2.1 Initial Writer Grant.

The Initial Writer hereby grants You a world-wide, royalty-free, non-exclusive license to use, reproduce, prepare Modifications of, compile, publicly perform, publicly display, demonstrate, market, disclose and distribute the Documentation in any form, on any media or via any Electronic Distribution Mechanism or other method now known or later discovered, and to sublicense the foregoing rights to third parties through multiple tiers of sublicensees in accordance with the terms of this License.

The license rights granted in this Section 2.1 ("Initial Writer Grant") are effective on the date Initial Writer first distributes Original Documentation under the terms of this License.

2.2. Contributor Grant.

Each Contributor hereby grants You a world-wide, royalty-free, non-exclusive license to use, reproduce, prepare Modifications of, compile, publicly perform, publicly display, demonstrate, market, disclose and distribute the Documentation in any form, on any media or via any Electronic Distribution Mechanism or other method now known or later discovered, and to sublicense the foregoing rights to third parties through multiple tiers of sublicensees in accordance with the terms of this License.

The license rights granted in this Section 2.2 ("Contributor Grant") are effective on the date Contributor first makes Commercial Use of the Documentation.

3.0 DISTRIBUTION OBLIGATIONS.

3.1. Application of License.

The Modifications which You create or to which You contribute are governed by the terms of this License, including without limitation Section 2.2 ("Contributor Grant"). The Documentation may be distributed only under the terms of this License or a future version of this License released in accordance with Section 5.0 ("Versions of the License"), and You must include a copy of this License with every copy of the Documentation You distribute. You may not offer or impose any terms that alter or restrict the applicable version of this License or the recipients' rights hereunder. However, You may include an additional document offering the additional rights described in Section 3.5 ("Required Notices").

3.2. Availability of Documentation.

Any Modification which You create or to which You contribute must be made available publicly in Editable Form under the terms of this License via a fixed medium or an accepted Electronic Distribution Mechanism.

3.3. Description of Modifications.

All Documentation to which You contribute must identify the changes You made to create that Documentation and the date of any change. You must include a prominent statement that the Modification is derived, directly or indirectly, from Original Documentation provided by the Initial Writer and

include the name of the Initial Writer in the Documentation or via an electronic link that describes the origin or ownership of the Documentation. The foregoing change documentation may be created by using an electronic program that automatically tracks changes to the Documentation, and such changes must be available publicly for at least five years following release of the changed Documentation.

3.4. Intellectual Property Matters.

Contributor represents that Contributor believes that Contributor's Modifications are Contributor's original creation (s) and/or Contributor has sufficient rights to grant the rights conveyed by this License.

3.5. Required Notices.

You must duplicate the notice in the Appendix in each file of the Documentation. If it is not possible to put such notice in a particular Documentation file due to its structure, then You must include such notice in a location (such as a relevant directory) where a reader would be likely to look for such a notice, for example, via a hyperlink in each file of the Documentation that takes the reader to a page that describes the origin and ownership of the Documentation. If You created one or more Modification(s) You may add your name as a Contributor to the notice described in the Appendix.

You must also duplicate this License in any Documentation file (or with a hyperlink in each file of the Documentation) where You describe recipients' rights or ownership rights.

You may choose to offer, and to charge a fee for, warranty, support, indemnity or liability obligations to one or more recipients of Documentation. However, You may do so only on Your own behalf, and not on behalf of the Initial Writer or any Contributor. You must make it absolutely clear than any such warranty, support, indemnity or liability obligation is offered by You alone, and You hereby agree to indemnify the Initial Writer and every Contributor for any liability incurred by the Initial Writer or such Contributor as a result of warranty, support, indemnity or liability terms You offer.

3.6. Larger Works.

You may create a Larger Work by combining Documentation with other documents not governed by the terms of this License and distribute the Larger Work as a single product. In such a case, You must make sure the requirements of this License are fulfilled for the Documentation.

4.0 APPLICATION OF THIS LICENSE.

This License applies to Documentation to which the Initial Writer has attached this License and the notice in the Appendix.

5.0 VERSIONS OF THE LICENSE.

5.1. New Versions.

Initial Writer may publish revised and/or new versions of the License from time to time. Each version will be given a distinguishing version number.

5.2. Effect of New Versions.

Once Documentation has been published under a particular version of the License, You may always continue to use it under the terms of that version. You may also choose to use such Documentation under the terms of any subsequent version of the License published by _____ [Insert name of the foundation, company, Initial Writer, or whoever may modify this License]. No one other than _____[Insert name of the foundation, company, Initial Writer, or whoever may modify this License] has the right to modify the terms of this License. Filling in the name of the Initial Writer, Original Documentation or Contributor in the notice described in the Appendix shall not be deemed to be Modifications of this License.

6.0 DISCLAIMER OF WARRANTY.

DOCUMENTATION IS PROVIDED UNDER THIS LICENSE ON AN "AS IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES THAT THE DOCUMENTATION IS FREE OF DEFECTS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. THE ENTIRE RISK AS TO THE QUALITY, ACCURACY, AND PERFORMANCE OF THE DOCUMENTATION IS WITH YOU. SHOULD ANY DOCUMENTATION PROVE DEFECTIVE IN ANY RESPECT, YOU (NOT THE INITIAL WRITER OR ANY OTHER CONTRIBUTOR) ASSUME THE COST OF ANY NECESSARY SERVICING, REPAIR OR CORRECTION. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY DOCUMENTATION IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER.

7.0 TERMINATION.

This License and the rights granted hereunder will terminate automatically if You fail to comply with terms herein and fail to cure such breach within 30 days of becoming aware of the breach. All sublicenses to the Documentation which are properly granted shall survive any termination of this License. Provisions which, by their nature, must remain in effect beyond the termination of this License shall survive.

8.0 LIMITATION OF LIABILITY.

UNDER NO CIRCUMSTANCES AND UNDER NO LEGAL THEORY, WHETHER IN TORT (INCLUDING NEGLIGENCE), CONTRACT, OR OTHERWISE, SHALL THE INITIAL WRITER, ANY OTHER CONTRIBUTOR, OR ANY DISTRIBUTOR OF DOCUMENTATION, OR ANY SUPPLIER OF ANY OF SUCH PARTIES, BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER DAMAGES OR LOSSES ARISING OUT OF OR RELATING TO THE USE OF THE DOCUMENTATION, EVEN IF SUCH PARTY SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES.

9.0 U.S. GOVERNMENT END USERS.

If Documentation is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in Documentation will be only as set forth in this Agreement; this is in accordance with 48 CFR 227.7201 through 227.7202-4

(for Department of Defense (DOD) acquisitions) and with 48 CFR 2.101 and 12.212 (for non-DOD acquisitions).

10.0 MISCELLANEOUS.

This License represents the complete agreement concerning the subject matter hereof. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. This License shall be governed by California law, excluding its conflict-of-law provisions. With respect to disputes or any litigation relating to this License, the losing party is responsible for costs, including without limitation, court costs and reasonable attorneys' fees and expenses. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any law or regulation which provides that the language of a contract shall be construed against the drafter shall not apply to this License.

Appendix

Public Documentation License Notice

The contents of this Documentation are subject to the Public Documentation License Version 1.0 (the "License"); you may only use this Documentation if you comply with the terms of this License. A copy of the License is available at http://www.openoffice.org/licenses/PDL.rtf.

The Original Documentation is _____. The Initial Writer of the Original Documentation is _____ (C) ____. All Rights Reserved. (Initial Writer contact(s):_____[Insert hyperlink/alias].)

Contributor(s):
_____.

Portions created by _____ are Copyright (C)_____[Insert year(s)]. All Rights Reserved. (Contributor contact(s): _____[Insert hyperlink/alias]).

Note: The text of this Appendix may differ slightly from the text of the notices in the files of the Original Documentation. You should use the text of this Appendix rather than the text found in the Original Documentation for Your Modifications.