Opening OpenOffice in Delphi

Michaël Van Canneyt

March 16, 2014

Abstract

In a previous article, it was shown how MS-Word can be used from withing Delphi. In this article, the same functionality is implemented using OpenOffice writer as the wordprocessing tool, or using OpenOffice Calc to create spreadsheets from a Delphi TDataset. OpenOffice is an Open Source implementation of an office suite, which offers the same functionality as MS-Office, at zero cost. It is therefore worth consideration when implementation or deployment cost is an issue.

1 Introducing the OpenOffice API

Just as MS-Office, OpenOffice offers developers the possibility to control it's various parts from external programs. This can be done using various programming languages, such as C, C++, but preferably Java or, on Windows, any language with built-in support for COM - such as Delphi.

The technology underlying the OpenOffice API is called UNO, which stands for Unified Network Objects. The API is based heavily on the concept of Interfaces - so it should be easily understandable for Delphi programmers familiar with interface or COM programming. It is impossible to cover all aspects of UNO in a single article, and no attempt will be made here. Only the basic things that are needed to control OpenOffice from within Delphi will be presented. Luckily, Toolbox has been shipped several times with a CD-ROM containing the OpenOffice Software Development Kit: It contains an elaborate manual covering all aspects of UNO, and also has the complete reference of all Interfaces, and many examples. The examples are written in Java, but are clearly written and should be very helpful in understanding the concepts of the API.

In fact, looking at the examples, one will quickly undertand that coding for OpenOffice is done much easier in Delphi than in Java, thanks to Delphi's excellent built-in support for COM.

All OpenOffice programming in Delphi must be done using variants which hold a reference to an OpenOffice interface obtained through COM: There is no TLB describing the OpenOffice API available, so it would be a lot of manual work to create all interface definitions.

The API will be presented by implementing an OpenOffice driver for the `TWordController` component presented in a previous article: This component could be used to create serial letters using MS-Word, or to create and fill tables in an MS-Word document, based on the data in a `TDataset` descendent. The component was implemented using various 'Driver' components for various versions of MS-Word. In this article a driver component will be developed that does not use MS-Word, but uses OpenOffice Writer instead.

After that, a smaller component is presented which allows to export the contents of a TDataset component to an OpenOffice spreadsheet document.

The TWordContriller driver component is called TOfficeWriterDriver and descends from the TOleWord class, presented in the previous article. The meaning of the various components and the general architecture of the component won't be repeated here, it can be found in the previous article.

Starting OpenOffice

OpenOffice implements a Windows Automation Server, which it calls the Automation Bridge: This bridge handles all conversions/communications between UNO and Windows the automation server's clients. It is described in detail in chapter 3 of the developer's guide.

To start the Automation Server, an instance of the OpenOffice Service Manager object must be created: This can then be used to create other objects (interfaces) in OpenOffice.

In the driver component, the connection with the word processor is established through the OpenWord method. In this method, the OpenOffice service manager reference must therefore be created:

```
procedure TOfficeWriterDriver.OpenWord(WithVisible: Boolean);
begin
  if VarIsEmpty(FOffice) then
     FOffice := CreateOleObject('com.sun.star.ServiceManager');
  If (VarIsEmpty(FOffice) or VarIsNull(FOffice)) then
    Raise EWordDriver.Create('Could not open OpenOffice');
  if VarIsEmpty(FDesktop) then
    FDesktop := FOffice.createInstance('com.sun.star.frame.Desktop');
end;
```

As with all COM objects, an instance is obtained through the CreateOleObject call. The name should be 'com.sun.star.ServiceManager'. This will start the OpenOffice automation bridge. The reference to the servicemanager is stored in the FOffice variant, for later reference.

The createInstance call of the OpenOffice service manager (part of its XMultiServiceFactory interface) can now be used to create other UNO objects. As a first object, the desktop is created: the desktop can be used to load or create new OpenOffice documents. The desktop is a 'Service': it is an object instance which offers various interfaces. One of the interfaces is the XComponentLoader interface, which is discussed in the next section. A reference to the instance of the desktop is stored in the FDesktop variant.

2 Opening or Creating an OpenOffice document

The OpenOffice Desktop service offers the XComponentLoader interface to load a document - any document which can be read by OpenOffice can be loaded this way.

The interface method to load a document is called LoadComponentFromURL. It is declared in IDL (Interface Description Language) as follows:

```
:com::sun::star::lang::XComponent
loadComponentFromURL(
  [in] string   URL,
  [in] string  TargetFrameName,
  [in] long    SearchFlags,
```

```
  [in] sequence<::com::sun::star::beans::PropertyValue> Arguments )
raises(::com::sun::star::io::IOException,
       ::com::sun::star::lang::IllegalArgumentException );
```

What this means is that the `loadComponentFromURL` call takes 4 arguments, returns a `XComponent` interface, and can raise 2 kinds of exception when executed.

The meaning of the arguments is as follows:

URL a URL describing the document that must be loaded. For files on a local disk, this starts with `file:///`, followed by the name of the file, with backslashes replaced by forward slashes.

TargetFrameName the name of the target frame: this frame will be used to load the document in.

SearchFlags Describes the search mechanism for the target frame.

Arguments Various loading options, described in the `MediaDescriptor` service of the OpenOffice API. This can include settings as 'Open Read-Only' etc.

When the call to `LoadComponentFromURL` is successful, it returns an interface to the asked document. The actual type of the interface depends on the loaded document.

There are some special URLs which can be used to create new documents:

private:factory/scalc Creates a new spreadsheet.

private:factory/swriter Creates a new Writer document.

private:factory/draw Creates a new drawing.

private:factory/simpress Creates a new presentation.

The `TargetFrameName` describes in what OpenOffice window (frame) the document must be loaded. This can be the name of an existing window, but can also have some special values, notably:

_blank Always opens a new frame (window) to load the document in.

_default Tries to re-use an existing frame (window), but opens a new one if none is found.

Other reserved values are possible, but these are sufficient for the current purposes.

The `Arguments` parameter is of type 'sequence': This type does not have an equivalent in COM, but appears quite often in the UNO API. Therefore, the OpenOffice Automation bridge expects a variant array, in which each of the elements in the array in itself is a variant holding a reference to an OpenOffice object. If there are no elements in the sequence, an empty array must be passed.

In the above case, each element in the variant array must hold a reference to a `PropertyValue` structure (a record). COM has no provisions for handling record types, so the OpenOffice automation bridge offers a `CoreReflection` service: It creates object instances which represent the record: The object has properties that have the same names as the fields in the record. Setting or reading the properties corresponds to setting or reading the fields of the record.

The `PropertyValue` instances must be obtained from the `CoreReflection` service offered by OpenOffice: An instance of this service must be created first. It can be obtained from the OpenOffice service manager.

To facilitate the creation of such property values, a `CreatePropertyValue` method is implemented:

```
procedure TOfficeWriterDriver.CreatePropertyValue(
  var PropertyValue: Variant; APropertyName : String);

Const
  SReflection = 'com.sun.star.reflection.CoreReflection';
  SPropV = 'com.sun.star.beans.PropertyValue';

begin
  If VarIsNull(FCoreReflection)
     or VarIsEmpty(FCoreReflection) then
    FCoreReflection:=FOffice.createInstance(SReflection);
  FCoreReflection.forName(SPropV).CreateObject(PropertyValue);
  PropertyValue.Name:=APropertyName;
end;
```

The `CreatePropertyValue` method takes 2 arguments:

1. A variant in which a reference to the `PropertyValue` instance is stored.
2. The name of the property. It is stored in the `PropertyValue` instance's Name property.

The method above first checks whether there is a reference to the `CoreReflection` service of OpenOffice. If not, a reference is obtained through the service manager. Then the CoreReflection is used to create an instance of a PropertyValue object, and the `Name` property is set.

Armed with all this, the `OpenDocument` method of the Word Driver component can be implemented:

```
procedure TOfficeWriterDriver.OpenDocument(FN: String);

Var
  OpenParams : Variant;

begin
  OpenParams:=VarArrayCreate([0, -1], varVariant);
  If (fn<>'') then
    FN:='file:///'+StringReplace(FN,'\','/',[rfReplaceAll])
  else
    FN:='private:factory/swriter';
  FDocument:=FDesktop.LoadComponentFromURL(FN,'_default',
                                          0,OpenParams);
  If (VarIsEmpty(FDocument) or VarIsNull(FDocument)) then
    Raise EWordDriver.Create('Could not open document');
  FHaveDocument:=True;
end;
```

No open parameters are needed, so an empty array is created. Then the URL is constructed for the `LoadComponentFromURL` call: if an empty filename is passed, the special URL to create a new document is used. The document is opened in the default window. A reference to the document instance is stored in the `FDocument` variant.

Handling a OpenOffice document

The instance stored in FDocument is of type TextDocument. This service offers a lot of interfaces, and each of the interfaces handles a different aspect of the document. The main interface is called XTextDocument, which allows to get a reference to the complete text (i.e. the text content) of the document. Another interface is XReplaceable: it allows to search (and replace) text in the document. The XTextTablesSupplier interface will also come in handy. There are many other interfaces, the reference section of the TextDocument service lists them all.

The Word Driver component needs to retrieve a list of 'fields' in the document. A field was defined as a piece of text: a name, enclosed in curly braces. To retrieve all the field names, the XSearchable interface of the document can be used. The main methods of this interface are findFirst and findNext, which function in much the same way as the Windows file searching calls with the same name.

This interface also allows to create a 'SearchDescriptor' (similar to the TSearchRec record in the Windows file search mechanism), which can then be used to conduct a search. The SearchDescriptor has various properties, which control the way in which the search is done. The main properties of interest are SearchString (what to look for) and SearchRegularExpression, which tells openoffice to treat the search string as a regular expression.

Armed with this interface, the GetMergeFieldNames call can be implemented as follows:

```
procedure TOfficeWriterDriver.GetMergeFieldNames(List: TStrings);

Var
  FindDescriptor : Variant;
  AText,
  Found : Variant;
  S : String;

begin
  FindDescriptor:=FDocument.createSearchDescriptor;
  FindDescriptor.setSearchString('\{[^\{]*\}');
  FindDescriptor.SearchRegularExpression:=True;
  Found:=FDocument.FindFirst(FindDescriptor);
  While Not (VarIsNull(Found) or
             VarIsEmpty(Found) or
             VarIsType(Found,varUnknown)) do
    begin
    S:=Found.getString;
    Delete(S,1,1);
    SetLength(S,Length(S)-1);
    If (Pos(' ',S)=0) and (length(S)<32) then
      List.Add(S);
    Found:=FDocument.FindNext(Found.End,FindDescriptor);
    end;
end;
```

The result of the FindFirst/FindNext calls is stored in the Found variant. For text documents, this will hold a reference to a XTextRange interface, which represents a range of text. The getString method of this interface returns the actual text, which is used in the above code to retrieve the name of the field.

The `FindNext` needs a position at which the search should proceed. This is usally after the end of the previously found text. The `End` method of the XTextRange interface returns the end of the text, in this case the found text, so it is passed to the `FindNext` call.

Note that the regular expression resembles unix regular expressions more than MS-Word search expressions. The API reference contains a complete description of supported regular expressions.

The document can not only be searched, it can also be replaced. This is done using the `XReplaceable` interface of the document. The search and replace operation is described by a `ReplaceDescriptor` instance, which is similar to the `SearchDescriptor` object used in the previous code. Since for the purposes of the Word Driver component, it is sufficient to replace all occurrences of a tag with a supplied value, the `replaceAll` call of `XReplaceable` interface will be used to seach & replace the tags with their values. The `ReplaceValues` call of the word driver component is therefore very simple:

```
procedure TOfficeWriterDriver.ReplaceValues(Value: TStrings);

Var
  I,J : Integer;
  R,S : String;
  ReplaceDescriptor : Variant;

begin
  I:=0;
  While (Not FCancelled) and (I<Value.Count) do
    begin
    R:=Value[i];
    J:=Pos('=',R);
    If (J>0) then
      begin
      S:='{'+Copy(R,1,J-1)+'}';
      System.Delete(R,1,J);
      ReplaceDescriptor:=FDocument.createReplaceDescriptor;
      ReplaceDescriptor.setSearchString(S);
      ReplaceDescriptor.setReplaceString(R);
      FDocument.replaceAll(ReplaceDescriptor);
      end;
    Inc(I);
    end;
end;
```

The method is a simple loop over all vsupplied name/value pairs: after extracting the tag name and value from the stringlist, the replacedescriptor is filled with appropriate data, and the `replaceAll` method is invoked.

3 Handling tables

The TextDocument object also implements the `XTextTablesSupplier` interface. It's single method `getTextTables` gives acces to all tables in the document. The `getTextTables` returns an `XNameAccess` interface, which is similar in functionality to Delphi's `TCollection` class. Using this interface, the tables in the document can be accessed by their names or by their (zero-based) index.

Each table is of type `TextTable`, which offers several interfaces, of which `XTextTable` is the most important one. This interface offers acces to the rows (via `getRows`) or columns (through `getColumns`) in the table. Individual cells can be accessed using the `getCellByName` method.

The `getRows` and `getColumns` methods return `XTableRows` and `XTableColumns` interfaces, respectively. They can be used to access existing rows and columns (via their index), or to insert new ones.

All these interfaces can now be used to retrieve the list of fieldnames in a table, so they can be replaced by data. This is done in the `GetTableFieldNames` method:

```
procedure TOfficeWriterDriver.GetTableFieldNames(List: TStrings;
                                                  Dataset: TDataset);

Var
  TC,TR,T : OleVariant;
  I,RCount,CCount : Integer;
  S : String;
  F : TField;

begin
  TC:=FDocument.GetTextTables;
  If TC.getCount>0 then
    begin
    // Get first table.
    T:=TC.getByIndex(0);
    // Get number of rows, columns
    RCount:=T.getRows.getCount;
    CCount:=T.getColumns.getCount;
    // Retrieve names from cells.
    For I:=0 to CCount-1 do
      begin
      S:=Trim(T.getCellByName(ColName(I)+IntToStr(RCount)).getString);
      If Length(S)>0 then
        begin
        If S[1]='{' then
          Delete(S,1,1);
        If Length(S)>0 then
          If S[Length(S)]='}' then
            S:=Copy(S,1,Length(S)-1);
        end;
      If Assigned(Dataset) and (S<>'') then
        F:=Dataset.Fields.FindField(S)
      else
        F:=Nil;
      List.AddObject(S,F);
      end;
    end
  else
    Raise EWordDriver.Create(SErrNoTablesInDocument);
end;
```

The method is quite straightforward: The `getTextTables` interface is used to retrieve a reference to the first table, if one exists. Then the number of rows and columns is retrieved

through the XTextRows and XTextColumns interfaces. Finally, the getCellByName method is used to retrieve the cells of the last row. The XText interface of each cell can be used to retrieve the text of each cell. Finally, A reference to each field in the dataset is stored in the stringlist.

The field references in this stringlist will be used to fill the table with data from the dataset, as shown in the InsertTableFromTemplate method. It uses the same interfaces as the same previous method:

```
procedure TOfficeWriterDriver.InsertTableFromTemplate(Dataset: TDataset);

Var
  TC,T,C : OleVariant;
  I,RCount : Integer;
  FieldList : TStrings;

begin
  TC:=FDocument.GetTextTables;
  If TC.getCount>0 then
    begin
    FieldList:=TStringList.Create;
    Try
      GetTableFieldNames(FieldList,Dataset);
      // Get first table.
      T:=TC.getByIndex(0);
      // Get number of rows, columns
      RCount:=T.getRows.getCount;
      Dataset.First;
      While Not Dataset.Eof do
        begin
        For I:=0 to FieldList.Count-1 do
          begin
          C:=T.getCellByName(ColName(I)+IntToStr(RCount));
          If Assigned(FieldList.Objects[i]) then
            C.setString(Tfield(FieldList.Objects[i]).AsString)
          else
            C.setString('');
          end;
        Dataset.Next;
        If Not Dataset.EOF then
          begin
          Inc(RCount);
          T.getRows.insertByIndex(RCount,1);
          end;
        end;
    finally
      FieldList.Free;
    end;
    end;
end;
```

This method closely resembles the previous one: Instead of retrieving the cell text of the last row in the table, it sets it using the setString method of the cell's XText interface.

New rows are inserted using the insertByIndex method of the XTableRows interface

of the table. It's first argument is the index of the row. The second one is the number of rows to insert (in this case: 1)

It's of course also possible to insert new tables in the document. How this can be done is shown in the `InsertTable` method. This method searches for a tag in the document, and replaces it with a new table.

A new table should always be created and then inserted in the text. Until it is inserted somewhere in the text, it will not be part of the document. The document service also offers the `XMultiServiceFactory` interface. It's `createInstance` method can be used to create a new table instance.

A table in OpenOffice can have a name; the name can be set with the `setName` method. In this example, the name will be set to the name of the tag which was used to mark the location of the table.

When the table is created, it has no rows or columns. Before accessing any of the rows and columns, the table should be initialized with the `initialize` method: it takes as arguments the initial number of rows and columns in the table. The number of rows is set initially to 1, the number of columns is set to the number of fields being exported.

All this is demonstrated in the `InsertTable` method:

```
procedure TOfficeWriterDriver.InsertTable(Dataset: TDataset;
                                          NameTag: String;
                                          FieldList: TStrings);


Var
  F,Found : Variant;
  FreeList : Boolean;
  T  : Variant;
  TR : Variant;
  Cell : Variant;
  CurrRow,I : Integer;

begin
  F:=FDocument.createSearchDescriptor;
  F.setSearchString('{'+NameTag+'}');
  Found:=FDocument.FindFirst(F);
  If (VarIsNull(Found) or VarIsEmpty(Found)) then
    Raise EWordDriver.CreateFmt(SErrNoSuchTable,[NameTag]);
  FreeList:=(FieldList=Nil);
  Try
    If FreeList then
      begin
      FieldList:=TStringList.Create;
      For I:=0 to Dataset.Fields.Count-1 do
        FieldList.AddObject(Dataset.Fields[i].FieldName,
                            Dataset.Fields[i]);
      end
    else
      For I:=0 to FieldList.Count-1 do
        FieldList.Objects[i]:=
          Dataset.Fields.FindField(Fieldlist[i]);
```

The above is initialization code: the location for the table is searched using the `XSearchable`

interface. If no fieldlist was passed, a complete fieldlist is constructed.

The actual creating of the table and filling it with data follows:

```
    T:=FDocument.createInstance('com.sun.star.text.TextTable');
    T.setName(NameTag);
    T.initialize(1,FieldList.Count);
    FDocument.Text.InsertTextContent(Found,T,True);
    For I:=0 to FieldList.Count-1 do
       begin
       Cell:=T.getCellByName(ColName(I)+IntToStr(1));
       Cell.setString(TField(FieldList.Objects[i]).DisplayName);
       end;
```

The above code creates a table, initializes, and inserts it in the document using the `InsertTextContent` method. This method takes a reference to a `XTextRange` interface as it's first parameter: this indicates the location where the content should be inserted. The second parameter is the content to be inserted (the table) and the third parameter indicates whether the content of the range should be replaced by the new content.

After that, the first row of the table is filled with the fieldnames. When this is done, the rest of the table is filled with the data in the dataset. This is done similarly to the code of the previous method:

```
     TR:=T.getRows;
     CurrRow:=1;
     Dataset.First;
     While Not Dataset.Eof do
       begin
       Inc(CurrRow);
       TR.insertByIndex(CurrRow,1);
       For I:=0 to FieldList.Count-1 do
         begin
         Cell:=T.getCellByName(ColName(I)+IntToStr(CurrRow));
         Cell.setString(Tfield(FieldList.Objects[i]).AsString);
         end;
       Dataset.Next;
       end;
  Finally
    If FreeList then
      FieldList.Free;
  end;
end;
```

Finally, if a list of fields was created at the start of the method, it is freed again.

4 Creating a merge document

For easy printing the word export creates a 'merge document' in which all generated documents are included so they form a single large document. This is done by inserting a reference to each generated document one after the other in a single document. This is accomplished with a field.

The same can be done in OpenOffice Writer. Each Writer document consists of one or more sections, either part of the document or referring to an external document. Adding a link

to an external document is therefore accomplished by adding a section (a TextSection object) which refers to an external document. Referring to an external document is done by setting the various fields of the FileLink property of the TextSection. This property is of type SectionFileLink, a structure (record). Therefore an object representing this structure must be obtained from the CoreReflection object. The FileURL field of this record must point to the external document for the section.

Finally, the section must be added at the end of the document. To do this, a text cursor (of type TextCursor) is created: In OpenOffice, Cursors are used to navigate through text, paragraphs, words, table cells etc. A text cursor can be created from the XText interface of the document text, using the createTextCursor call. Using the cursor's gotoEnd method, the cursor can be positioned at the end of the text.

Armed with these interfaces and methods, the AddMergeDoc method of the OpenOffice word driver can be implemented:

```
procedure TOfficeWriterDriver.AddMergeDoc(FN: String;
                                          AddPageBreak: Boolean);

Const
  SReflection  = 'com.sun.star.reflection.CoreReflection';
  STextSection = 'com.sun.star.text.TextSection';
  SFileLink    = 'com.sun.star.text.SectionFileLink';

Var
  Txt,TS,Curs : OleVAriant;
  LO : OleVariant;
  S : String;

begin
  Fn:=StringReplace(FN,'\','/',[rfReplaceAll]);
  FN:='file:///'+FN;
  TS:=FDocument.createInstance(STextSection);
  If VarIsNull(FCoreReflection) or
     VarIsEmpty(FCoreReflection) then
    FCoreReflection:=FOffice.createInstance(SReflection);
  FCoreReflection.forName(SFileLink).createObject(LO);
  LO.FileURL:=FN;
  TS.setPropertyValue('FileLink',LO);
  TS.setName(FN);
  Txt:=FDocument.getText;
  Curs:=Txt.createTextCursor;
  Curs.gotoEnd(False);
  Txt.insertTextContent(Curs,TS,true);
  if AddPageBreak then
    begin
    Curs.gotoEnd(False);
    S:=#13;
    Txt.insertString(Curs,S,false);
    Curs.setPropertyValue('BreakType',4);
    end;
end;
```

After rewriting the filename in the form of an URL, a new TextSection object is obtained from the document, and a SectionFileLink is obtained from the CoreReflection

service. It's `FileURL` property is set to the computed URL, and the whole is stored ini the `FileLink` property of the TextSection object.

After that, a text cursor is created to navigate to the end of the document. The `insertTextContent` call (encountered previously) is then used to insert the text section at the cursor position: the end of the document.

If the `AddPageBreak` parameter is true, an end-of-paragraph marker (ASCII code 13) is inserted at the cursor position using the `insertString` method of the Text interface, and the `BreakType` property of the paragraph is set to 4, which corresponds to the `PAGE_AFTER` enumeration constant.

5 Printing and saving

Now that the document has been filled with data, it can be saved and printed. Saving the document is handled through the documents `XStorable` interface. This interface implements 3 methods: `store`, `storeAsURL` and `storeToURL`. All methods write the document to a certain location. The first method stores the document on it's current location, if it was loaded from a location. It corresponds to the **'Save'** command. The second changes the location and name of the document (corresponding to **'Save As'**), while the last just stores a copy of the document on the indicated location, but doesn't change the location of the document.

Using this, the `SaveDocument` method of the Word driver can be implemented easily:

```
procedure TOfficeWriterDriver.SaveDocument(FN: String);

Var
  SaveParams : Variant;

begin
  SaveParams:=VarArrayCreate([0, -1], varVariant);
  FN:='file:///'+StringReplace(FN,'\','/',[rfReplaceAll]);
  FDocument.StoreAsUrl(FN,SaveParams);
end;
```

Note again that no parameters are needed, so the last parameter (store options) is again an empty variant array.

Similarly, printing a document is done using the `XPrintable` interface: It offers 3 methods; One for retrieving the current printer (`getPrinter`), one for setting it (`setPrinter`) and one to actually print the document `print`. Various options for printing can be set, but for the purposes of the word driver component, none are needed, hence the `PrintDocument` method is very simple:

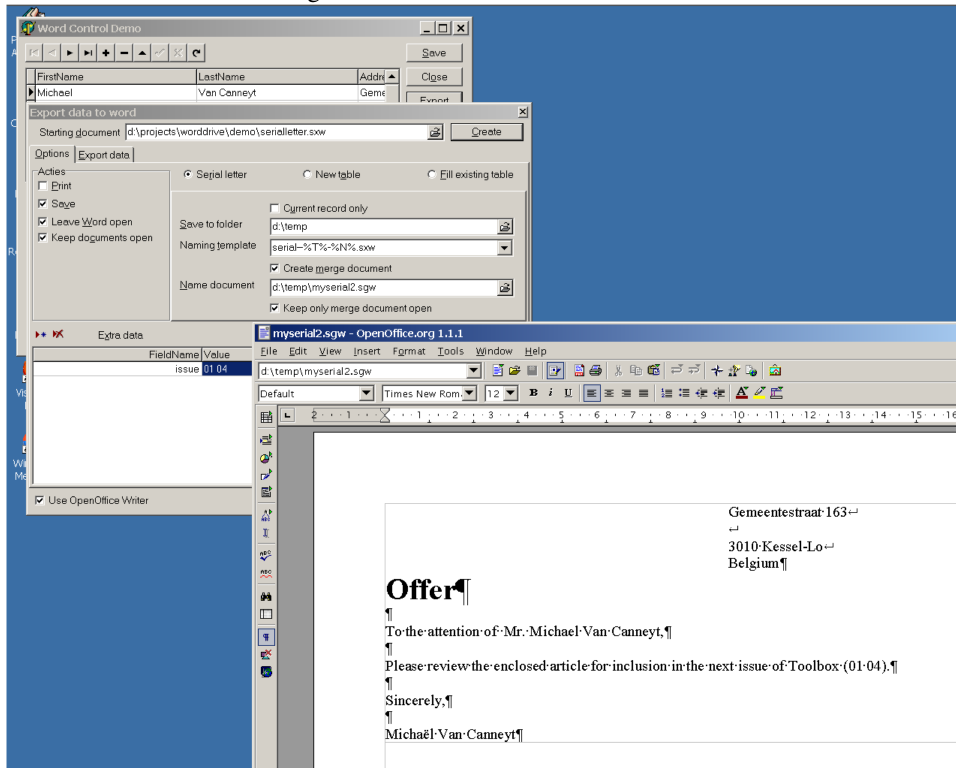
```
procedure TOfficeWriterDriver.PrintDocument;

Var
  PrintOpts : Variant;

begin
  PrintOpts:=VarArrayCreate([0, -1], varVariant);
  FDocument.Print(PrintOpts);
end;
```

The result of all this can be seen in figure 1 on page 13.

Figure 1: Office Writer driver at work



6 Using OpenOffice Calc

Just like MS-Word, OpenOffice Writer can make a serial letter ('mail merge') by merging a template document with a spreadsheet: the columns in the spreadsheet are used as fields, the rows as columns of a table.

Popular Delphi component repositories contain various components to export the contents of a `TDataset` descendent to an Excel document. To show the ease with which OpenOffice can be controlled from within Delphi, a small component is presented which exports the contents of a `TDataset` to an OpenOffice calc document. The component is called `TStarCalcExport`. It has a small list of properties:

TemplateFileName The name of an existing OpenOffice calc document. This document will be filled with the data. If it is empty, a new document will be created.

FileName The name under which the filled document will be saved.

Dataset The dataset whose data will be exported.

Column The starting column (1 based, so column A corresponds to a value of 1) at which data will be written to the spreadsheet.

Row The starting row at which data will be written to the spreadsheet.

CloseStarCalc If set to `True`, OpenOffice calc will be closed after the file is saved.

IncludeFields A stringlist with fieldnames. If it is non-empty, then only fields which appear in this list will be included in the export.

ExcludeFields A stringlist with fieldnames. If it is non-empty, then only fields which do not appear in this list will be included in the export.

IncludeFieldnames If set to `True`, then the first created row will contain the fieldnames of the exported fields (the `DisplayName` property of the field will be used)

The main procedure of this component is the `Execute` method, which does the actual export. It's a conceptually very simple method:

```
procedure TStarCalcExport.Execute;
begin
  If (FDataset=Nil) then
    Error(SErrNoDataset);
  If (TemplateFileName<>'') and FileExists(TemplateFileName) then
    LoadSpreadSheet(TemplateFileName)
  else
    LoadSpreadSheet('');
  Try
    CreateSheet('');
    Try
      FillCells;
      SaveSpreadSheet(FileName);
    Finally
      VarClear(FSheet);
    end;
  Finally
    If CloseStarCalc then
      Try
        FDocument.Close(true);
        VarClear(FDocument);
      finally
        VarClear(FDesktop);
      end;
  end;
end;
```

The `LoadSpreadSheet` call loads an existing spreadsheet in OpenOffice calc, or creates a new one. After the document was loaded, a new spreadsheet is created if the document contains no spreadsheet using the `CreateSheet` method (a reference to the found or created sheet is stored in the `FSheet` variable). When the sheet is created, it is filled with data using the `FillCells` method, which does the actual work. The `SaveSpreadSheet` method saves the filled spreadsheed. Finally, OpenOffice Calc is closed again if it needs to be.

The `LoadSpreadSheet` is very similar to the `OpenDocument` method discussed earlier:

```
procedure TStarCalcExport.LoadSpreadSheet(AFileName: String);

Var
  FN: String;
  OpenParams : Variant;

begin
  CheckDesktop;
```

```
  OpenParams:=VarArrayCreate([0, -1], varVariant);
  FN:=AFileName;
  If (AFileName='') then
    FN:='private:factory/scalc'
  else
    FN:='file:///'+StringReplace(FN,'\','/',[rfReplaceAll]);
  FDocument:=FDesktop.LoadComponentFromURL(FN,'_default',0,OpenParams);
  If (VarIsEmpty(FDocument) or VarIsNull(FDocument)) then
    If (AFileName='') then
      Error(SErrFailedToCreateDocument)
    else
      Error(SErrFailedToOpenDocument);
end;
```

The `CheckDesktop` method will connect to OpenOffice and create a reference to the `Desktop` service, which is then used to load the document. As can be seen, the special URL `private:factory/scalc` is used to create a new spreadsheet (note again that an empty array of variants is passed to indicate the absence of options). The `Error` call raises an exception using the passed string parameter as a message.

The `CreateSheet` method loads the first sheet in the spreadsheet document; if no sheet is present, new sheet is created with the passed name. It does this using the `XSpreadsheets` interface offered by the `getSheets` method of the spreadsheet object's `XSpreadSheetDocument` interface. The `XSpreadSheets` interface allows named access to the sheets in the spreadsheet document via the `getByName` method, and allows inserting a new spreadsheet using the `insertNewByName` method. Both methods are used in the `CreateSheet` method of the `TStarCalcExport` component, which is very straightforward.

```
procedure TStarCalcExport.CreateSheet(Const SheetName: String);

Var
  Sheets : Variant;
  SN : String;

begin
  CheckDocument;
  Sheets:=FDocument.getSheets;
  SN:=SheetName;
  If (SN='') then
    SN:='Sheet1';
  FSheet:=Sheets.getByName(SN);
  If VarIsEmpty(FSheet) or VarIsNull(FSheet) then
    begin
    Sheets.insertNewByName(SN, 0);
    FSheet:=Sheets.getByName(SN);
    If VarIsEmpty(FSheet) or VarIsNull(FSheet) then
      Error(Format(SErrCouldNotCreateSheet,[SN]));
    end;
end;
```

Now that the sheet in which to insert the data is available, the `FillCells` method can do the actual work of inserting the data in the spreadsheet. The `XCellRange` interface of the sheet allows to access individual cells of the sheet using the `getCellByPosition` method. It accepts a (zero-based) position of a cell and returns the `XCell` interface of the specified cell. This interface can be used to retrieve or set the contents and properties of

the cell.

The `FillCells` method is implemented again in a very straightforward manner:

```
procedure TStarCalcExport.FillCells;

Const
  FloatTypes = [ftSmallInt,ftInteger,ftWord,ftFloat,ftCurrency,
                ftBCD,ftAutoInc,ftLargeInt];

Var
  CRow,SCol,I : Integer;
  C : Variant;
  L : TStringList;
  F : TField;
  FN : String;

begin
  CRow:=FRow-1;
  SCol:=FColumn-1;
  If CRow<0 then
    CRow:=0;
  If SCol<1 then
    SCol:=0;
  L:=TStringList.Create;
  Try
    GetFieldList(L);
    If IncludeFieldNames then
      begin
      For I:=0 to L.Count-1 do
        begin
        C:=FSheet.getCellByPosition(SCol+I,CRow);
        C.setString(TField(L.Objects[i]).DisplayName);
        end;
      Inc(CRow);
      end;
```

After determinining the initial positions from the designtime properties of the component, the list of fields to export is retrieved in a stringlist, and a reference to the fields is stored in the `Objects` array of the stringlist. If the `IncludeFieldNames` property is set to `True`, then the first row is constructed using the `DisplayName` properties of the fields in the list. Note that the text is set using the cell's `XText` interface's `setString` method, and not the `setValue` or `setFormula` methods of the `XCell` interface. The `setValue` method can only be used for numerical (real or integer) values.

The actual data is inserted through a double loop procedure:

```
    With FDataset do
      While not EOF do
        begin
        For I:=0 to L.Count-1 do
          begin
          F:=TField(L.Objects[i]);
          If Not F.IsNull then
            begin
```

```
          C:=FSheet.getCellByPosition(SCol+I,CRow);
          If F.DataType in FloatTypes then
            C.setValue(F.AsVariant)
          else
            C.setString(F.AsString);
          end;
        end;
      Next;
      Inc(CRow)
      end;
  Finally
    L.Free;
  end;
end;
```

Note that the loop tries to insert numerical data as values, while all other data is inserted as text.

Finally, the filled spreadsheet is saved using the document's `XStoreable` interface, just as it was done for the text document:

```
procedure TStarCalcExport.SaveSpreadSheet(Const AFileName : String);

Var
  SaveParams : Variant;
  FN : String;

begin
  CheckDocument;
  SaveParams:=VarArrayCreate([0, -1], varVariant);
  FN:='file:///'+StringReplace(AFileName,'\','/',[rfReplaceAll]);
  FDocument.StoreAsUrl(FN,SaveParams);
end;
```

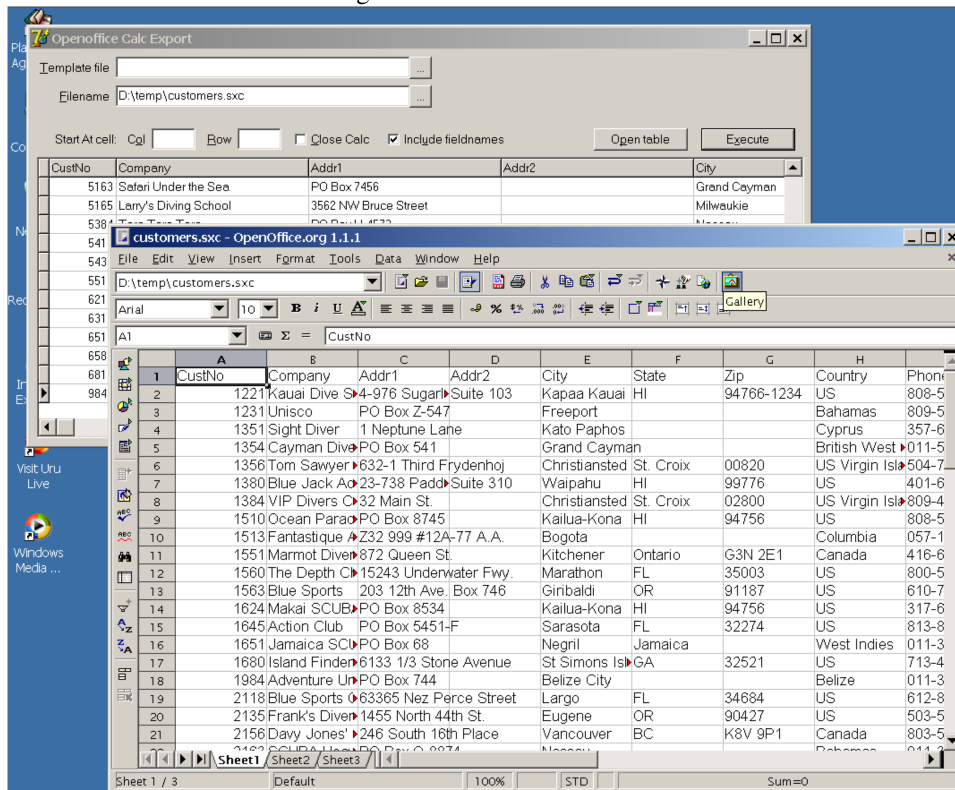
All other methods of the `TStarCalcExport` are quite simple and straightforward, they will not be discussed here.

The result of this all can be seen in figure 2 on page 18.

7 About the `TWordDriver` component

In a previous article, a `TWordDriver` control was presented which allows to control MS-Word through COM from within Delphi. In this article, the component will be expanded so it can handle OpenOffice documents as well. For the purpose of clarity, the component was renamed `TWordController`. It has also been extended to handle the use of form templates: A MS-Word template containing form fields. Instead of searching and replacing tags in the document text, form fields are searched, and their values are filled in. The bookmark names of the form fields are matched against the supplied values (from the dataset or user-supplied values). The technique for doing this was kindly donated by Mr. B. Schmelzer, and was adapted to create serial letters based on form templates. The same technique was introduced for OpenOffice writer documents, using user variable fields, as input fields do not lend themselves to the task. The interested reader should consult the `NewDocument` method for opening a document based on a template, the `GetFormFieldNames` method

Figure 2: Office calc at work



which retrieves the names of the fields, and the `ReplaceFormFieldValues` method which sets the field values.

Conclusion

OpenOffice offers an incredibly rich API, with a multitude of services, interfaces and methods; Looking at the documentation can be very intimidating, even though there are a lot of examples. Nevertheless, the two components presented here show that programming OpenOffice (especially from Delphi) is quite easy and straightforward. It is simply a matter of diving in, and browsing through the documentation, till an interface is found that will do the task which is at hand. Hopefully, the article presented here has helped to lower the treshold somewhat, and will encourage people to consider integrating this remarkably versatile package in their own software. The author wishes to thank Mr. B Schmelzer for donating the code to search and fill in form fields in a MS-Word document.