Using MS Word in a Delphi application

Michaël Van Canneyt

August 4, 2009

Abstract

Two techniques to control MS-Word from a Delphi application are discussed in this article. As a demonstration, a small tool is developed which allows to use Word for producing a serial-letter, or to produce tables from data in a TDataset. Based on the techniques shown, it should be possible to create any kind of document - or indeed, control MS-Wor - from a Delphi application.

1 Introduction

Finding a PC that does not have MS-Word installed may prove a difficult task: The word-processor from Microsoft is ubiquitous. It therefore makes sense to use it whenever it is necessary to create beautiful output from a program. What is more, most people doing administrative work are probably familiar with it: For such people it will be easier to use MS-Word as a reporting tool instead of Crystal Reports or some other reporting solutions that come with Delphi, such as Quickreport, FastReport, Preport, Reportbuilder and many others. For most people, typing some small letter in MS-Word with some placeholders for names, addresses or telephone numbers and amounts will be much easier than mastering a powerful but complex reporting tool in order to produce a simple invoice.

So, rather than relying on some reporting solution, why not use MS-Word to produce output for a Delphi application ? Delphi makes it actually very easy: The language support for Variants as references to OLE automation servers and the support for COM interfaces make steering MS-Word (or indeed Excel or MS-Access) quite easy. In this article, both approaches are shown.

2 Using OLE automation server classes

Controlling MS-Word is done through its COM interface: Through Delphi's OLE mechanism, an OLE reference is obtained to the MS-Word application, and then the various commands exposed by the MS-Word interface can be used to let MS-Word do virtually anything that is needed. The same can be done with Excel and other members of the MS-Office suite.

Delphi 5 and higher versions come with a set of components on the component palette that make controlling an MS-Office application quite easy: `TWordApplication` and `TWordDocument` are classes that present the complete Word interface. Technically, these are very simple classes created by importing Microsoft's Type Library for its MS-Office applications: They can be generated quite easily: in the 'Project' menu, the item 'Import Type Library' will present a list of type libraries installed on the PC. Selecting the correct type library will generate some pascal units that contain the Pascal Interface declarations

corresponding to the COM interface defined in MS-WORD - or any other application that has registered a type library.

The type libraries imported by the Borland developers come in various flavours: The 'word97' unit contains the MS-Office 97 interface, the Word2000 unit contains the MS-Office 2000 interface. On the CD, a set of units is included which present the Word-XP interface (These are not delivered e.g. with Delphi 5)

The interface exposed by these components is called 'Word.Application'. It is documented completely in the MS-Word help files, if the 'Visual Basic Reference Help for Word' was installed together with MS-Word. There, the complete set of commands, properties can be explored and subsequently put to use in the Delphi application.

The use of the Ole automation server components is very simple: A `TWordApplication` component is dropped on a form, the 'Connect' method is called, and MS-Word will be started - or an error is raised if MS-Word is not installed on the PC.

The `TWordApplication` component has a 'Documents' property, which is a reference to the collection of documents, opened in word. The 'Open' call can be used to open an existing document, or the 'New' method can be used to start a new document. For instance, a procedure to open a document could look as follows:

```
Procedure TMainForm.OpenWordDocument(FN : String);

Var
  Word : TWordApplication;
  D : _Document;
  O : OleVariant;

begin
  Word:=TWordApplication.Create(Self);
  Try
    Word.Connect;
    Word.Visible:=True;
    O:=FN;
    D:=Word.Documents.Open(O,EmptyParam,EmptyParam,
                           EmptyParam,EmptyParam,
                           EmptyParam,EmptyParam,
                           EmptyParam,EmptyParam,
                           EmptyParam);
    // Maybe do other things.
  Finally
    Word.Free;
  end;
end;
```

As can be seen from the code above, after connecting to Word, it is not visible on the screen: Only after setting the 'Visible' property to `True`, the MS-Word window will appear on screen.

The call to `Open` needs a lot of parameters: the full list is documented in Word, but the Delphi IDE will show them when code completion features are turned on. All parameters are declares as OleVariants passed by reference (NOT by value), i.e. declared as a 'var' parameter. This means that a variable of type OleVariant must be passed - no automatic type conversion will be done: That is why the filename is first stored in a OleVariant variable ('O'), which is then used in the `Open` call. This is in fact an artefact of the translation into pascal code: The parameters could have been passed as `Const` parameters, which would

have allowed to pass a variable of any compatible type, and the Delphi compiler would have done the conversion itself. As it is done using `var` parameters, the options have to be of type `OleVariant`.

Most of the parameters to the `Open` method are declared optional in MS-Word, but they are required when using the Interface in Delphi. To indicate that a value is not passed on to Word, the pre-defined `EmptyParam` variable can be used.

The return value of the `Open` call is a reference to a `_Document` interface: Its methods can be used to fill the document with content.

The above example immediatly shows a problem with this approach: Because Delphi knows the interface of `TWordApplication` (and the `Documents` property, it will complain if the wrong kind or wrong number of parameters is passed to a call. Normally, this is good: This way, one is sure that the call to Word is made correct.

However: The semantics of the `Open` call changes from version to version: The above call is correct for Word 97, but is wrong for Word 2000, which expects not 10, but 12 arguments in its Open call. And Word XP expects 15 arguments. Since at compile time it is not known which version of MS-Word will be installed on the PC that will run the application, all three possibilities must be foreseen, and the correct one must be used depending on the actual version of Word that is installed on the PC where the code runs - with the possibility that when a new version of word is released, the code won't work again.

However, there is another approach, as will be shown in the next section.

3 Using (OLE)Variants

According to the documentation, a variable of type `Variant` can also hold a reference to an interface. This is of little use unless the calls, exposed by the interface, can somehow be accessed by Delphi. Delphi does this using 'late binding': It can treat the Variant as a Class: any code which could be interpreted as a method, it interprets as a call to a method of the interface to which the variant refers. It does this by generating code that will dynamically look up the method to call, and encodes the arguments in a special way, understood by the COM system. No compile-time checking whether the arguments are valid is done.

Basically, this means that the following code is possible:

```
Var
  Word : Variant;

begin
  Word:=CreateOleObject('Word.Application');
  Word.Documents.Open('mydoc.doc');
end;
```

Delphi will not check whether the last statement is correct. Instead, it will generate code to get a reference to the 'Documents' collection of the Word interface, and then it will use this reference to generate code that will call the `Open` method of this collection with 1 parameter, the filename.

It is important to realize that this code is not checked at compile time in any way: the code will compile fine. Only when the code is actually run, then an error may occur if something was wrong. This means that errors in the code will *not* be detected unless the code is actually run.

Note that the number of parameters in the above code is not correct: only one parameter is passed. The compiler (actually, the RTL) generates the code to pass 'EmptyParam' for all

missing parameters. To help the compiler encoding the parameters correctly, it is possible to specify the parameter names:

```
Word.Documents.Open(FileName:='mydoc.doc',ReadOnly:=True);
```

Note that this can only be done for optional parameters.

The above technique can only be used for OLE Automation servers which expose the `IDispatch` interface: this interface allows Delphi to encode the calls correctly. It has the advantage that the code will still function when a newer version of e.g. MS-Word is installed on the PC, since Microsoft usually maintains more or less backward-compatibility and the parameters are encoded with their names.

While it is is of course tempting to use this powerful feature of Delphi, one must be careful when using this. Since the code is not checked at compile-time, errors may go un-noticed for a very long time, namely till the code is actually executed. For small projects and interfaces, this should not be a problem. For large interfaces (such as word) with a lot of calls and a lot of code, it may take along time before the cause of an error is spotted.

In the below code, a hybrid approach is taken: A separate class is developed with an interface that suits the needs of the application, and this class will the call the methods of an appropriate 'driver' class, which contains compile-time checked calls to the interface of Word: there is a driver class for each version of Word - and as an extra, there is a 'generic' class, which will use Variants and late binding to access Word.

4 Exporting data to MS-Word

The purpose of the `TWordDriver` class is to open a document and fill in parameters in the document using some pre-defined placeholders. For this, the search and replace mechanism of MS-Word is used. After all placeholders have been replaced, the document with markers replaced can be printed or saved (or both).

The markers are simple words, enclosed with curly brackets, as can be seen in figure 1 on page 5 The driver will recognize the texts `{Title}` and `{FirstName}` as placeholders which should be replaced with some value. This value can come from 2 sources:

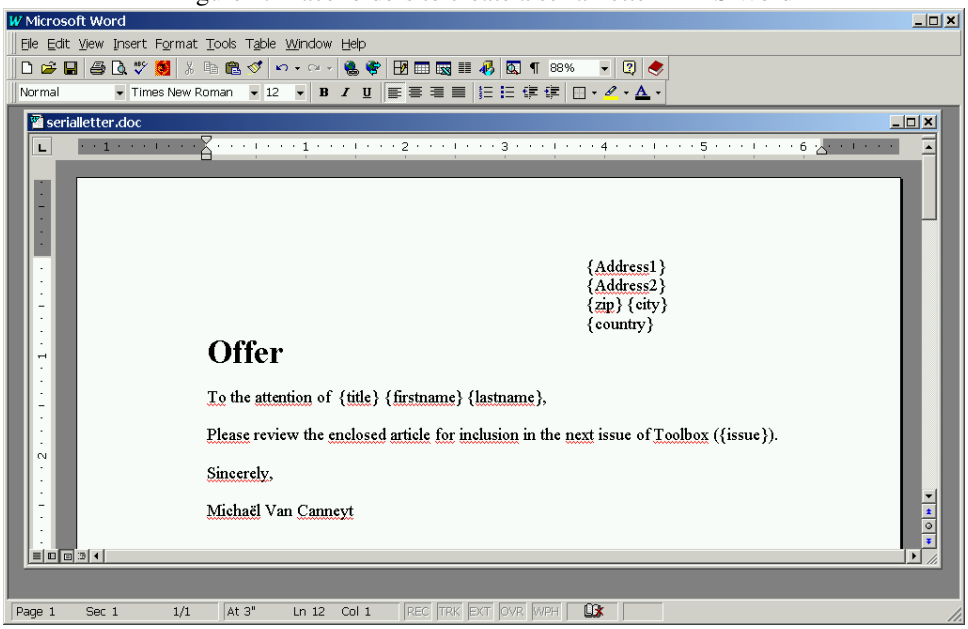
1. A list of `Name=Value` pairs. A placeholder whose text matches one of the names will be replaced with the corresponding value.
2. A `TDataset` descendent. A placeholder whose text matches one of the fieldnames in the dataset will be replaced with the field value.

To create a serial letter, the dataset is scrolled through, and for each record in the dataset, the document is loaded freshly, the placeholders are substituted with their contents, and the document is either saved or printed at once. The saving can be done with a template: the directory to save the resulting documents in should be specified, and a name template must be specified. The template can also contain placeholders:

1. `%N%` will be replaced with the record number.
2. `%D%` will be replaced with the current date.
3. `%FIELDNAME%` will be replaced with the value of the named field.

For instance to create a serial letter to all customers, the files could be saved with a template of

Figure 1: Placeholders to create a serial letter in MS Word




```
C:\My Documents\Offers\offer-%D%-%CustNo%.doc
```

This would create one document for each customer.

To make printing at a later time easier, a 'merge' document can be made: this is simply a document which includes (via the MS-Word INCLUDETEXT field) all generated documents. Opening this document will open all generated documents.

A second functionality is simply to generate a table with selected fields from a dataset: This starts from a document which contains a placeholder for the table: The placeholder will be replaced by the generated table. A list of fields (and their order) can be specified. An example could look as in figure 2 on page 6. The programmer should indicate which placeholder is supposed to contain the table. All other placeholders will be substituted with their contents. Note that the other placeholders will be substituted only once, and no value should be specified for the table placeholder.

A last functionality is to take an existing table, and fill it with data from a dataset. This is done by taking the last row of the table, and duplicating it, replacing any placeholders that are found: Any formatting is preserved, and the header of the table can be freely designed. Here, the same remark applies as for the creation of a new table: Placeholders outside the table will be replaced only once.

5 The `TWordDriver` class

The class which implements all this is the `TWordDriver` class. It contains all logic needed to create or fill a table or create a serial letter. The actual calling of of MS-Word is delegated to a `TOLEWord` class: This is an abstract class with the following interface:

```
TOLEWord = class(TComponent)
  procedure CloseWord(QuitWord : Boolean);
  procedure OpenWord(WithVisible : Boolean);
  procedure OpenDocument(FN : String);
```

Figure 2: Creating a table in MS Word

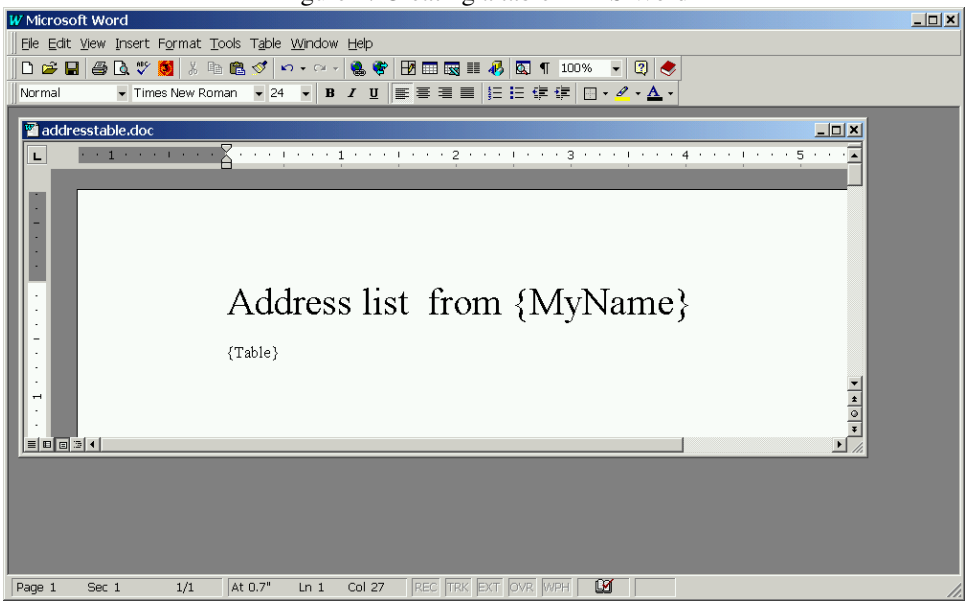
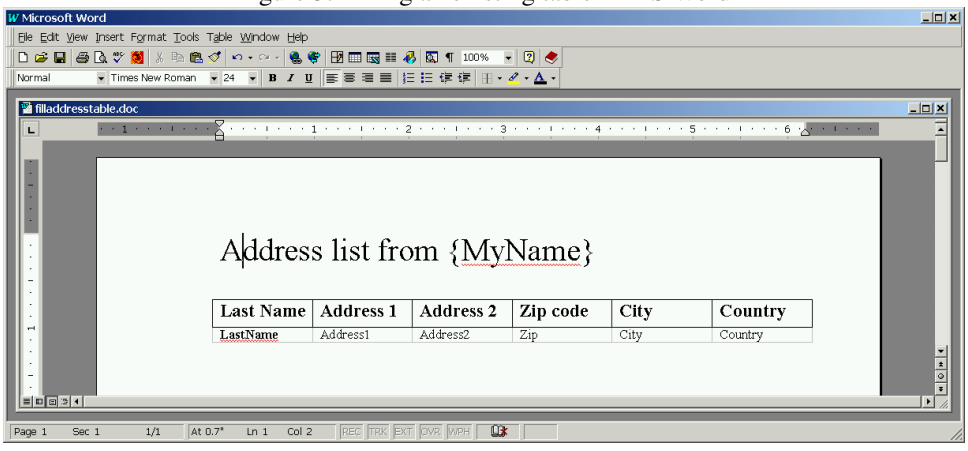


Figure 3: Filling an existing table in MS Word



```
  procedure SaveDocument(FN : String);
  procedure CloseDocument(KeepOpen : Boolean);
  procedure PrintDocument; Virtual; Abstract;
  Procedure ReplaceValues(Value : TStrings);
  procedure GetMergeFieldNames(List: TStrings);
  Property  HaveDocument : boolean;
  procedure GetTableFieldNames(List : TStrings;
                               Dataset : TDataset);
  procedure InsertTable(Dataset: TDataset; NameTag: String;
                        FieldList : TStrings);
  procedure InsertTableFromTemplate(Dataset : TDataset);
  Procedure AddMergeDoc(FN: String;AddPageBreak: Boolean);
end;
```

From this class, 4 descendent classes are made:

```
TWord97Driver = Class(TOleWord)    // version 8.X
TWord2000Driver = Class(TOleWord)  // version 9.X
TWordXPDriver = Class(TOleWord)    // version 10.X
TGenericWordDriver = Class(TOleWord)
```

The first 3 directly call the appropriate interface methods. The last one uses late binding to make the calls to Word. The actual code for these classes is the same: only the interface they use is different. The comment tells us which version is reported by MS-Word for each of these.

The `TWordDriver` class can try to decide which of these 'driver' classes it will use by determining the installed MS-Word version and creating the appropriate driver. Determining which word version is installed is done with a late binding method:

```
function TWordDriver.GetWordVersion : String;

Var
  M : Variant;

begin
  Try
    try
      M:=CreateOleObject('Word.Application');
      Result:=M.Application.Version;
    Except
      DriverError(SErrGettingVersion);
    end;
  Finally
    VarClear(M);
  end;
end;
```

This will return a string with the version of MS-Word, e.g; "8.2". Based on this, the `TWordDriver` class can decide what driver class to create. The programmer (or user) can also sexplicitly specify which version must be used.

The TWordDriver class has a lot of properties, which must be set appropriately before it can do its work. The following properties can be set:

FileName Name of the document which will be used as the template.

SaveFileName Filename in which to save the generated document when a table is created or filled. This is ignored for a serial letter.

Print Set to `True` if the generated document should be printed at once.

Save Set to `True` if the enerated document should be saved. The `SaveFileName` property must also be set in that case.

KeepOpen Should MS-Word be left open (i.e. visible) on screen after the work is done.

OutputDir Folder where the saved documents must be stored.

SaveTemplate Filename template used when saving documents. Should not include a path.

Datasource Datasource to use.

Values Extra `Name=Value` pairs to use when substituting placeholders with values.

WordVersion Word version to use: if set to `wvDetect`, the component will attempt to detect the installed version.

OnlyCurrentRecord Only create a serial letter for the current record.

Busy Is 'True' while the component is working.

DocumentCount Number of documents generated.

OnProgress Callback called after each document, usefull for displaying a progress bar.

DocumentMode What kind of document should be created: this can be one of `dmSerialLetter`, `dmNewtable` or `dmFillTable`.

TableTag Name of the placeholder where a new table should be inserted.

ColumnList List of columns (fields) to use when creating a new table. The order in the columnlist determines the order of the columns.

CreateMergeDocument If set to `True`, a merge document is created after a serial letter was made.

MergeDocumentFileName Name of the file in which the merge document is saved.

After these properties have been set, the `Execute` method will start the actual work. The execute method looks as follows:

```
FBusy:=True;
Try
  FDocumentNames.Clear;
  FDocumentCount:=0;
  If Not FileExists(FFileName) then
    DriverError(Format(SErrNoSuchFile,[FFileName]));
  OpenWord;
  Try
    ResetCancel;
    Case DocumentMode of
      dmSerialLetter : CreateSerialDoc;
      dmNewTable     : CreateNewTable;
      dmFillTable    : FillTable;
    end;
```

```
  Finally
    CloseWord(Not FKeepOpen);
  end;
Finally
  FBusy:=False;
end;
```

There is not much mysterious about this method: It checks whether the document exists, then proceeds to start word, and finally calls the actual method which will do the work, depending on the `DocumentMode` property.

Opening word happens as follows:

```
procedure TWordDriver.OpenWord;
begin
  If FDriver=Nil then
    CreateDriver;
  FDriver.OpenWord(FKeepOpen);
end;

procedure TWordDriver.CreateDriver;

Var
  V : TWordVersion;
  RV : String;

begin
  V:=FWordVersion;
  If (V=wvDetect) then
    begin
    RV:=GetWordVersion;
    If Pos('8.',RV)=1 then
      V:=wvWord97
    else if pos('9.',RV)=1 then
      V:=wvWord2000
    else if pos('10.',RV)=1 then
      V:=wvWordxp
    else // try generic driver.
      V:=wvGeneric;
      //DriverError(Format(SErrUnSupportedWordVersion,[RV]));
    end;
  Case V of
    wvWord97 : FDriver:=TWord97Driver.Create(Self);
    wvWord2000 : FDriver:=TWord2000Driver.Create(Self);
    wvWordXP : FDriver:=TWordXPDriver.Create(Self);
    wvGeneric : FDriver:=TGenericWordDriver.Create(Self);
  else
    DriverError(SErrNoSuchDriver);
  end;
end;
```

Again, nothing mysterious about these 2 methods: If necessary, the word driver is created. To create the driver the installed version is determined if needed. Depending on the specified or detected version, a driver is created which will be used in the rest of the methods.

The `Create` constructor and `Open` method of the drivers look as follows:

```
constructor TWord97Driver.Create(AOwner: TComponent);
begin
  inherited;
  FWord:=Word97.TWordApplication.Create(Self);
  FDocument:=Word97.TWordDocument.Create(Self);
end;

procedure TWord97Driver.OpenWord(WithVisible : Boolean);
begin
  FWord.Connect;
  If WithVisible then
    FWord.Visible:=True;
end;
```

Note that the unit name is specified when creating an instance of the `TWordApplication` class. This is the only difference between the various driver classes.

The `CreateSerialDoc` method contains the code to create a serial letter. It is again a very simple method, which is essentially a simple loop:

```
Procedure TWordDriver.CreateSerialDoc;
Var
  I : Integer;

begin
  If (Datasource=Nil) or (DataSource.Dataset=Nil) or
     (FOnlyCurrentRecord) then
    DoDocument(1)
  else
    begin
    With DataSource.Dataset do
      begin
      First;
      I:=0;
      While Not EOF or FCancelled do
        begin
        Inc(I);
        DoDocument(i);
        Next;
        end;
      end;
    If CreateMergeDocument then
      CreateMergeDoc;
    end;
end;
```

The real work of creating a new serial document is done in `DoDocument`:

```
procedure TWordDriver.DoDocument(NR : Integer);

Var
  TheValues : TStringList;

begin
  OpenDocument(FFileName);
```

```
  Inc(FDocumentCount);
  If (NR=1) then
    GetMergeFieldNames(FDocumentParams);
  Try
    TheValues:=TStringList.Create;
    Try
      GetValues(TheValues);
      GetDatasetValues(TheValues);
      ReplaceValues(TheValues);
      If Not FCancelled then
        begin
        If Print then
          PrintDocument;
        If Save then
          SaveDocument(GetSaveFileName(Nr,TheValues));
        end;
    Finally
      TheValues.Free;
    end;
  finally
    CloseDocument(FKeepOpen);
  end;
end;
```

After opening the start document, a list of placeholders is retrieved if this is the first time the document is opened. This is done to optimize the search and replace process: Only values that are known to exist in the document are searched and replaced for.

After that, the list of values to be used for this docyment is determined: the global values list, and the values in the current record of the dataset. After this, the values are replaced in the document using the `ReplaceValues` method. After this, depending on user settings, the document is printed and/or saved.

The `ReplaceValues` does the actual work of searching placeholders and replacing them with their contents. This work is delegated to the driver:

```
procedure TWordDriver.ReplaceValues(Value: TStrings);
begin
  FDriver.ReplaceValues(Value);
end;

procedure TWord97Driver.ReplaceValues(Value: TStrings);

Var
  I,J : Integer;
  R,S : String;
  D1,D2,D3 : OleVariant;

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);
      With FDocument.Content.Find do
        begin
        Text:=S;
        ClearFormatting;
        Replacement.Text:=R;
        ReplaceMent.ClearFormatting;
        D1:=True;
        D2:=wdfindcontinue;
        D3:=wdReplaceAll;
        Execute(EmptyParam,EmptyParam,EmptyParam,
                EmptyParam,EmptyParam,EmptyParam,
                D1,D2,EMptyParam,emptyParam,D3);
        end;
      end;
    Inc(I);
    end;
end;
```

Again, this is a simple loop: all values are searched and replaced. The `FDocument` field of the driver contains a reference to the currently open document. The `Content` property of the document interface is of type 'Range' - and a 'Range' contains the find functionality: the 'find' functionality will search the `Range` - in this case, since the range is 'Content', the whole document will be searched. The `Find` property of the Range is again an interface, in which several properties must be set:

Text The text to search.

Replacement The text with which to replace.

The `Execute` method will then do the actual search and replace. The options passed (note that they should all be of type OleVAriant) tell MS-Word that all occurrences must be replaced, and that it should continue searching when the end of the selection is reached. The call to `ClearFormatting` tells MS-Word that it should not touch the formatting when searching, or when replacing.

The exact semantics for the Execute call can be found in the MS-Word Visual Basic reference help. It is also used to get the list of placeholders for a document in the `GetMergeFieldNames` call:

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

Var
  W,F : Boolean;

begin
  W:=False;
  F:=not HaveDocument;
  If F then
    begin
    W:=not Assigned(FDriver);
    If W then
      OpenWord;
```

```
    OpenDocument(FileName);
    end;
  Try
    FDriver.GetMergeFieldNames(List);
  finally
    If F then
      CloseDocument(False);
    if W then
      CloseWord(True);
  end;
end;
```

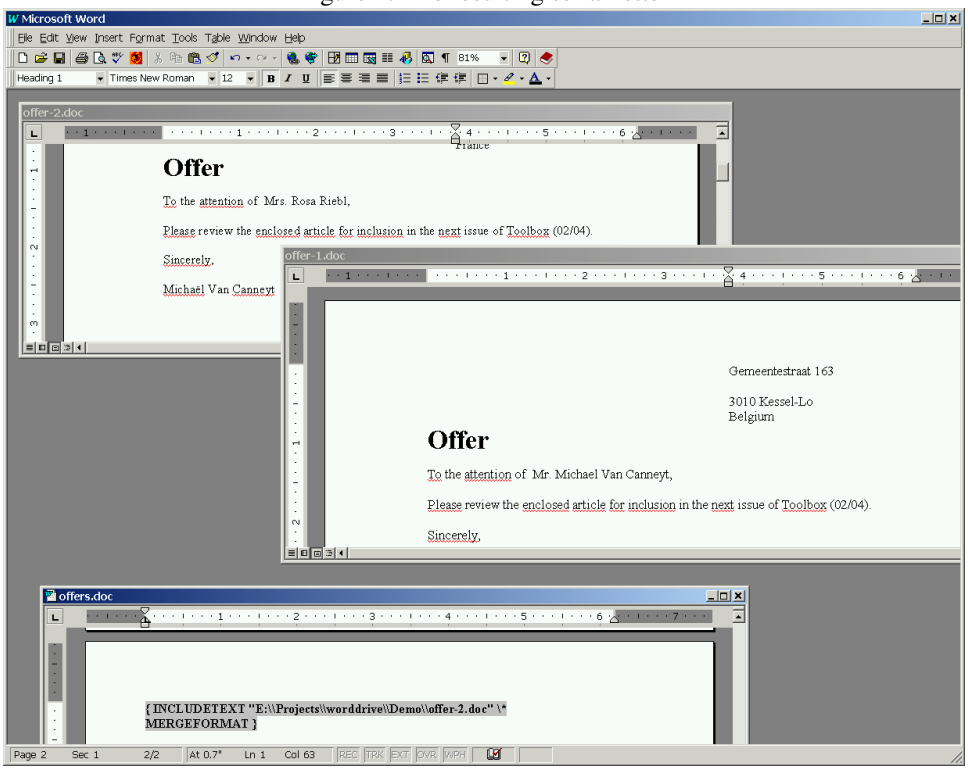
Most of this procedure is code to see whether the document is loaded. The actual work is again done by the driver class:

```
procedure TWord97Driver.GetMergeFieldNames(List : TStrings);

Var
  R : Word97.Range;
  S : String;

begin
  R:=FDocument.Content;
  With R.Find do
    begin
    Text:='\{[!\{]@\}';
    ClearFormatting;
    MatchWildcards:=True;
    MatchCase:=False;
    Repeat
      If Execute(emptyParam,emptyParam,emptyParam,
                 emptyParam,emptyparam,EmptyParam,
                 EmptyParam,EmptyParam,EmptyParam,
                 EmptyParam,EmptyParam) then
        begin
        S:=R.Text;
        Delete(S,1,1);
        SetLength(S,Length(S)-1);
        If (Pos(' ',S)=0) and (length(S)<32) then
          List.Add(S);
        end;
    Until not Found;
    end;
end;
```

Here the `Find` interface is used with a regular expression to find all placeholders in the document: The `MatchWildCards` tells MS-Word to use regular expressions. The `Found` property of the `Find` interface is used to determine whether a match was found or not: When no match is found, the loop is exited.

The above code also demonstrates an important feature in the OLE interface of MS-Word: Ranges. A `Range` can be any piece of text in MS-Word. In the above example, the range `R` is initially defined as the whole text of the document: this defines the scope of the search process. When a match is found, the range `R` is redefined as the piece of text that was found: in the above code, it is used to save the found text in a stringlist.

Figure 4: The resulting serial letter



The result of all this code can be viewed in figure 4 on page 14.

The other methods, to create or fill a table, are equally simple as the code for a serial letter. The code to create a new table looks as follows:

```
Procedure TWordDriver.CreateNewTable;

Var
  TheValues : TStrings;

begin
  If Not (Assigned(FDataSource) and
          Assigned(DataSource.Dataset)) then
    DriverError(SErrNoDatasetAvailableForTable);
  OpenDocument(FFileName);
  Try
    Inc(FDocumentCount);
    TheValues:=TStringList.Create;
    Try
      GetValues(TheValues);
      ReplaceValues(TheValues);
      If Not FCancelled then
        InsertTable(Datasource.Dataset,
                    TableTag,FColumnList);
      If Not FCancelled then
        begin
        If Print then
```

```
          PrintDocument;
        If Save then
          SaveDocument(SaveFileName);
        end;
    finally
      TheValues.Free;
    end;
  finally
    CloseDocument(FKeepOpen);
  end;
end;
```

The design pattern is followed again: the real work is done by the driver class:

```
procedure TWord97Driver.InsertTable(Dataset : TDataset;
                                    NameTag : String;
                                    FieldList : TStrings);

Var
  R : Word97.Range;
  T : Word97.Table;
  TR : Word97.Row;
  I : Integer;

begin
  R:=FDocument.Content;
  With R.Find do
    begin
    Text:='{'+NameTag+'}';
    ClearFormatting;
    MatchWildcards:=False;
    MatchCase:=False;
    If Execute(emptyParam,emptyParam,emptyParam,
               emptyParam,emptyparam,
               EmptyParam,EmptyParam,EmptyParam,
               EmptyParam,EmptyParam,EmptyParam) then
      begin
      R.Text:='';
      If FieldList=Nil 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]);
      T:=R.Tables.Add(R,1,FieldList.Count);
      TR:=T.Rows.Item(1);
      For I:=0 to FieldList.Count-1 do
        With TR.Cells.Item(I+1).Range do
          begin
```

```
          Bold:=1;
          Text:=Tfield(FieldList.Objects[i]).DisplayName;
          end;
      Dataset.First;
      While Not Dataset.Eof do
        begin
        TR:=T.Rows.Add(EmptyParam);
        For I:=0 to FieldList.Count-1 do
          With TR.Cells.Item(I+1).Range do
            begin
            Text:=Tfield(FieldList.Objects[i]).AsString;
            Bold:=0;
            end;
        Dataset.Next;
        end;
      end
    else
      Raise EWordDriver.CreateFmt(SErrNoSuchTable,[NameTag])
    end;
end;
```

The beginning of this method is familiar: code to look for the table placeholder. When it is found, the actual work is done: The Range found by the `Find` interface is used to contain the text of the table. First a fieldlist is created (if it wasn't passed on by the calling procedure), and filled with references to the field objects - for performance reasons: this avoids excessively calling the dataset's `FieldByName` method in later loops.

Then the following statement is used to create a table:

```
T:=R.Tables.Add(R,1,FieldList.Count);
```

Each range has a `Tables` property: This is a collection interface which represents the tables inside the range. The `Add` method of this collection will create a new table inside the range, with 1 row and as many columns as there are fields, and it spans the whole range as per the the first argument to the call.

The result of the `Add` call is a `Table` object, stored in the `T` variable. The rest of the code is then child's play:

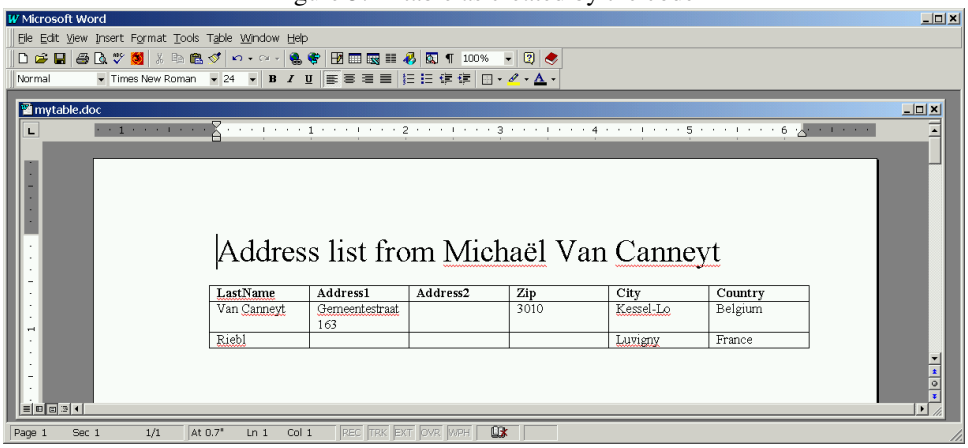
- The `Rows` collection of the table gives a reference to the first row, which is stored in `TR`.
- The cells of the first row are accessible through the `Cells` collection. Individual cells are accessible through the `Item` property.
- Each cell has again a `Range` property, which represents the contents of the cell: Setting the `Text` property of this range fills the cell with text.

So the first thing to do is create a header row for the table. Each cell is filled with the `DisplayName` of the fields in our fieldlist. The `Bold` attribute of the cell's range is set (other properties can be set).

After that, the dataset is browsed, adding a row to the tables for each row in the dataset, and filling the columns with the field contents. The result is shown in figure 5 on page 17.

Filling an existing table is done in a similar manner:

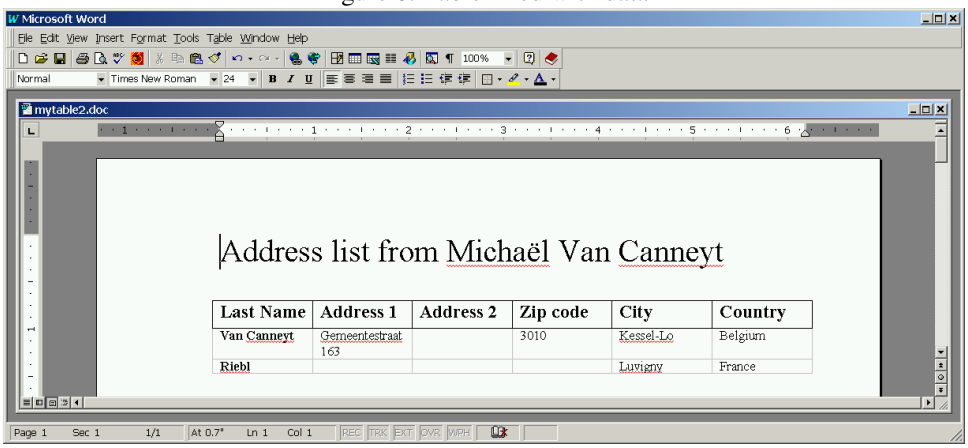
Figure 5: A table as created by the code




```
procedure TWord97Driver.InsertTableFromTemplate(Dataset : TDataset);

Var
  R : Word97.Range;
  TR : Word97.Row;
  T : Word97.Table;
  I : Integer;
  FieldList : TStrings;

begin
  R:=FDocument.Content;
  If R.Tables.Count>0 then
    begin
    FieldList:=TStringList.Create;
    GetTableFieldNames(FieldList,Dataset);
    T:=R.Tables.Item(1);
    TR:=T.Rows.Last;
    Dataset.First;
    While Not Dataset.Eof do
      begin
      For I:=0 to FieldList.Count-1 do
        With TR.Cells.Item(I+1).Range do
          If Assigned(FieldList.Objects[i]) then
            Text:=Tfield(FieldList.Objects[i]).AsString
          else
            Text:='';
      Dataset.Next;
      If Not Dataset.EOF then
        TR:=T.Rows.Add(EmptyParam);
      end;
    end;
end;
```

The first thing to do is to get the list of fields that should be inserted in the table. This is done in the `GetTableFieldNames` method - the interested reader can find the code on the CD-ROM accompagnying this article. The rest of the code is a simple set of nested loops, similar to the one in the previous method: The first loop loops through the dataset,

Figure 6: Table filled with data



and creates a new row as it goes along, and the second loop goes through the fields and fills the table cells with the fields contents. The result of this code can be seen in figure 6 on page 18

The rest of the methods of the driver are very simple indeed, and they will not be presented here. The interested reader is referred to the code of the component in the WordDriver.pas file. The above code was presented with the MS-Word 97 driver. The code for the other drivers is almost exactly the same: Only the actual calls to the interfaces may be different when the number of parameters is different.

6 An end-user export dialog

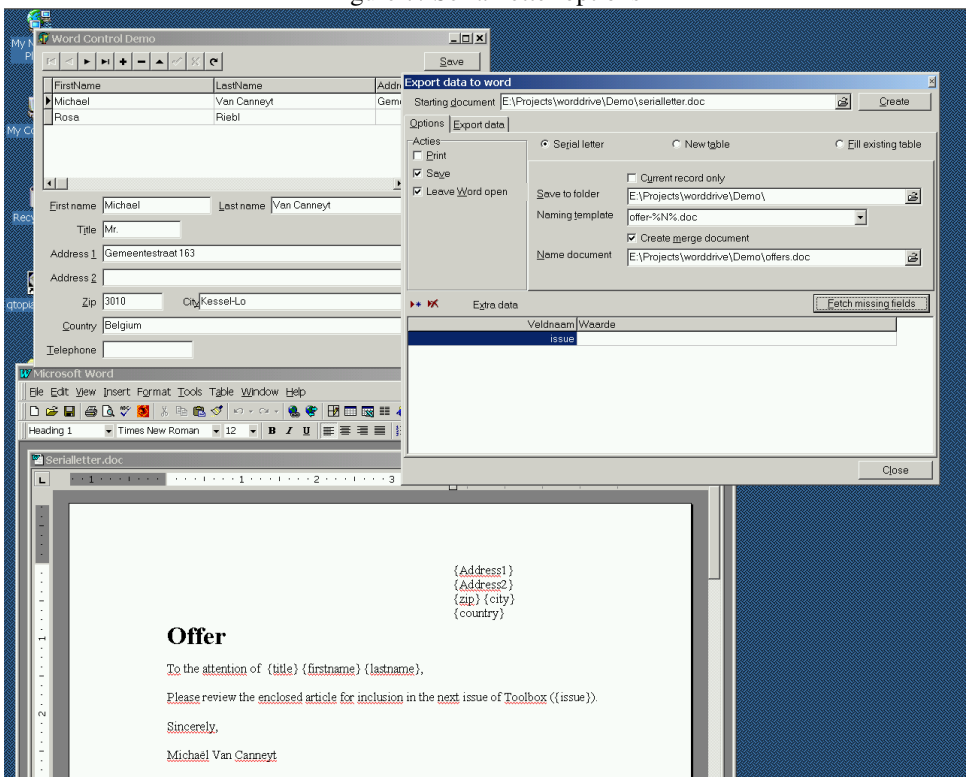
The component presented here is used in the compagny of the author to add an `export-to-word` capability to all forms with datasets on them: Through a menu entry, the user activates a wizard-like dialog which allows him to export the data of the current form to any arbitrary prepared MS-Word document.

All the dialog does is collect values for the various properties of the `TWordDriver` component, and when the user hits the 'Create' button, the component is put to work. The dialog even allows to edit (in a limited way) a copy of the form's data before it is exported. The form is called `TWordExportForm`, and can be used as follows:

```
procedure TMainForm.AExportExecute(Sender: TObject);
begin
  With TWordExportForm.Create(Self) do
    try
      Dataset:=CDSAddress;
      ShowModal;
    Finally
      Free;
    end;
end;
```

All it needs is the dataset which should be exported to MS-Word. The dialog handles the rest. Figures 7 (serial letter), 8 (create table) and 9 (fill table) show the various options which can be set in the dialog. When `Fetch missing fields` button is pressed, the document is scanned and placeholders are retrieved: The placeholders which match fields

Figure 7: Serial letter options



in the dataset are removed from the list, and the remaining names are placed in the 'Extra Data' grid: Here the user can enter extra content for the document. (In the example shown, the text for the 'Issue' placeholder)

The 'Naming template' combobox can be filled with pre-defined templates, to make it easier for the user to set a naming scheme.

In order to use this form, the RXLib suite must be installed for the grid and the filename edit component. This suite can be obtained from Torry's pages. Also, to be able to edit the data in the form, MIDAS (Datasnap) support in Delphi is needed. It should not be hard, however, to remove these two limitations from the form: The MIDAS limitation can be removed by using a memory dataset (one is provided in RXLib), RXLib can be removed by resorting to a regular drawgrid and a simple edit with a speedbutton to select a file.

7 Conclusion

Using the component presented here, it is not hard to export arbitrary data to MS-Word: It is by no means a finished product, but shows that it is relatively easy to control MS-Word from inside a Delphi application. There are some issues with Word versions, but they can be dealt with, at the price of loosing some of the compile-time checking mechanisms of Delphi.

The `TWordDriver` component can be used as is, but can easily be extended to include more options controlling, for example, the formatting of the text. It would also be possible to change it so the templates are not filled by a search-and-replace mechanism, but by using existing MS-Word mechanisms such as form fields. In general, a study of the MS-Word visual basic reference is very helpful in deciding how to control MS-Word from inside a

Figure 8: Create table options

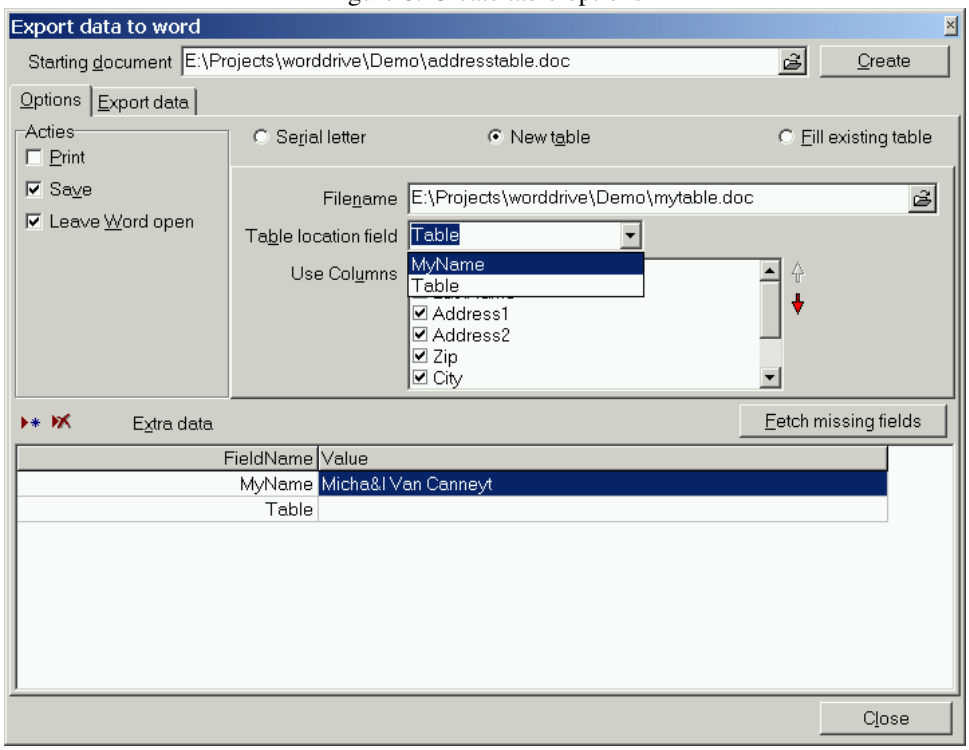
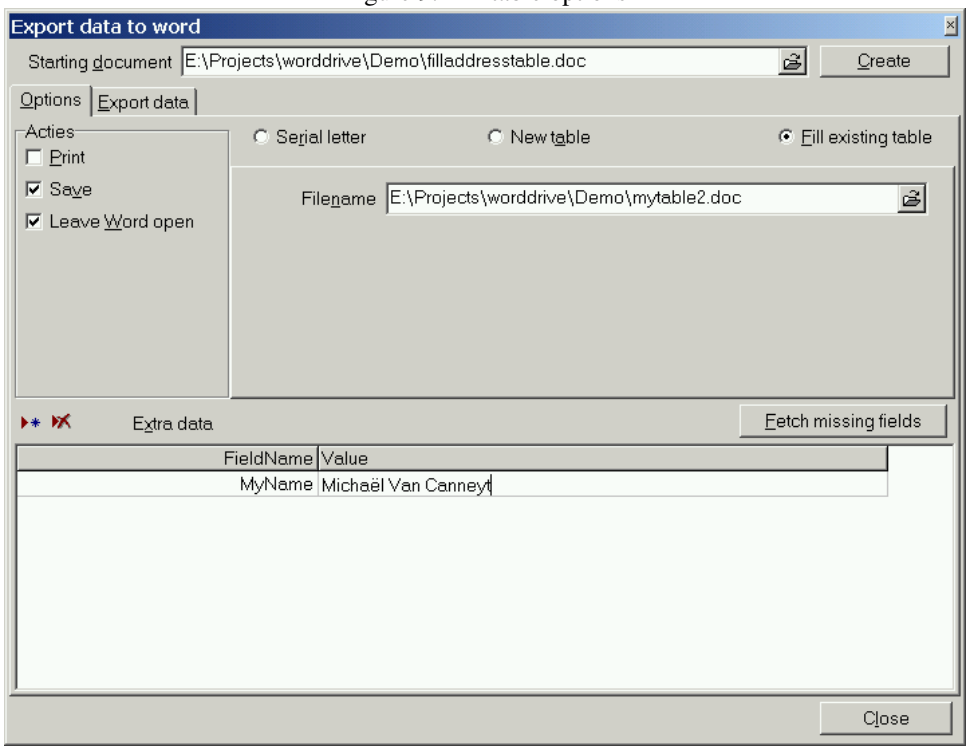


Figure 9: Fill table options



Delphi application: There are many examples of how to use the various properties and objects presented by the MS-Word interface. It may well be that the examples present a way that is better suited for the application that should interface to MS-Word than the code presented here: In that case the code is still useful in showing how one goes about when the development tool is not Visual Basic, but Delphi.

Of course, if a client's PC doesn't run MS-Word, but runs e.g. OpenOffice, then the code presented here is useless. But no fear: The same things can be done in OpenOffice as well, as will be shown in a future contribution.