Chapter 2: Using OCX Controls

Overview

The following sections provide guidance on using the SAP Automation Server objects. All code examples are provided in Visual Basic 4.0.

Contents

Summary of OCX Controls

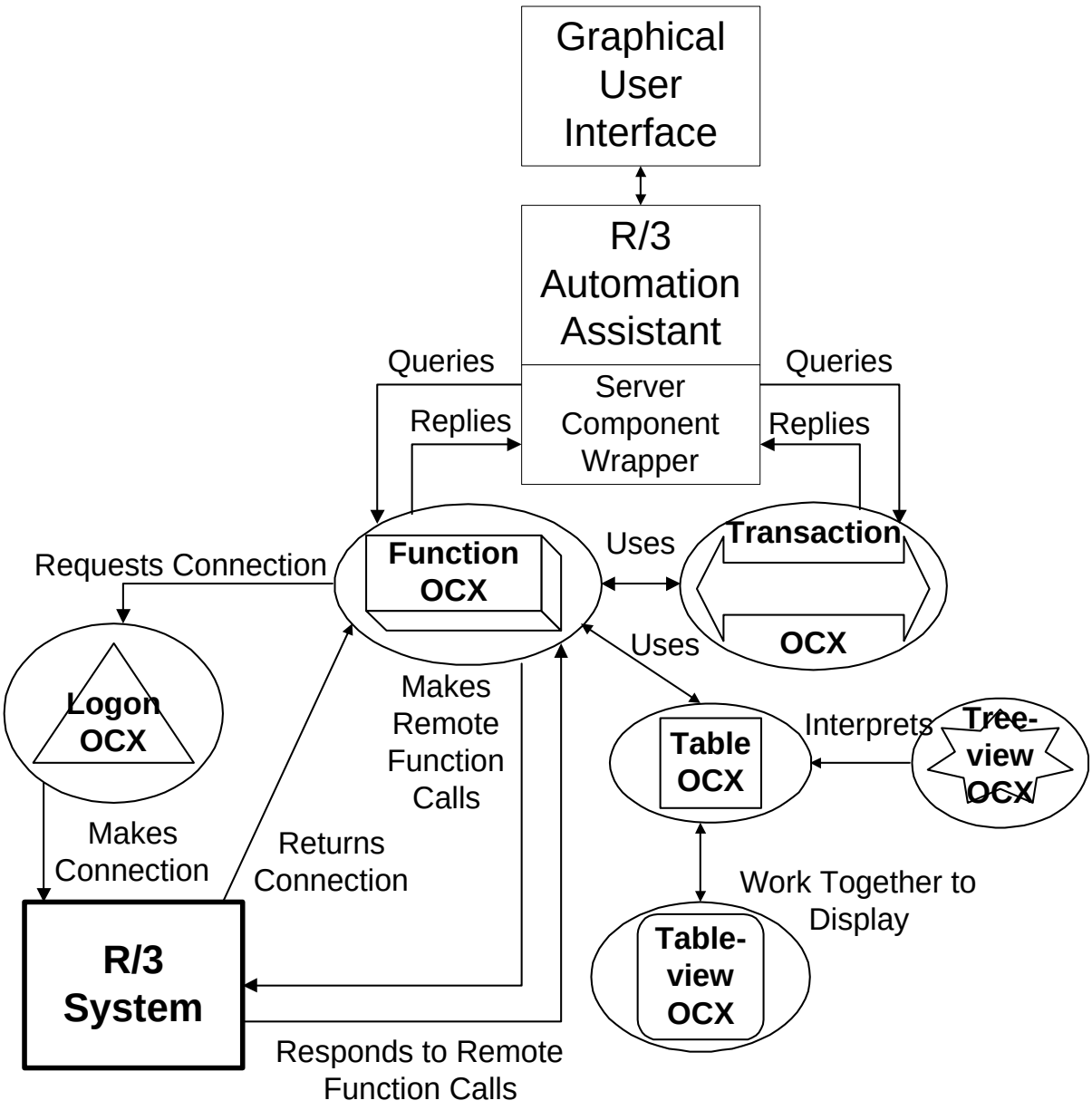
External applications access R/3 by making remote calls to R/3 functions. The SAP Assistant and the OCX components use **RFC** (the Remote Function Call API) to execute calls to the R/3 System. The SAP Assistant and OCX controls provide objects that allow the programmer to manage function call details from desktop applications.

The following are the available SAP OCX controls. Two of these, (the Function and Transaction controls), are available from SAP as separate component DLLs.

- The **Function control** provides objects (functions collection, function, logon, tables, table, exports, export, imports, import, parameter types, and so on) to log on to the R/3 System, manage the connection, request functionality, receive back data and direct data streams to client programs or the user interface. (See *The Function Control*.)
- The **Logon control** provides a connection object that makes it possible to log onto R/3. (See *The Logon control*)
- The **Transaction control** provides screen and field object management so that a user can remotely call R/3 transactions or use them in programs. (See *The Transaction Control*.)

  Note that the Transaction control only allows transaction calls in batch-input mode. That is, the external program can send input field values to an R/3 screen, but output field values are not returned. For complete transaction execution, with data transfer in both directions, use the *BAPI Component*.
- The **Table View control** allows an internal table from R/3's RFC library to be viewed on screen like a spreadsheet. (See *The Table View control*)
- The **Table Factory control** works with the Function control to manage tables attached to Function objects. In addition, the Table Factory encapsulates Table objects for easier access by the client application. (See *The Table Factory control*)
- The **Table Tree control** makes visible those tables that contain hierarchically structured data (trees of parent nodes and their children). Table Trees allow the programmer and user to manage tables containing directory trees. (See *The Table Tree Control* .
- A **Server Object Wrapper** that provides compatibility for OLE automation programs written for client applications of previous R/3 versions.

All of these objects are invoked via OLE Automation, and they in turn communicate with R/3. The objects manage all the necessary details: using RFC libraries, packing tables, making calls, receiving responses and displaying information. The following diagram illustrates how the controls and SAP Assistant components perform these functions:



OCX Interoperation Details

Possible Uses for OCX Controls

Here are some examples of how programs can use OCX controls.

Program	Possible Use	Server	Client
MS Excel 5	Upload Planning Data, Download Report Data	√	√
MS Project 4	Control Purchasing Schedules	√	√
Visio	View Business Process Flows	√	√
MS Word 6	Use Spellcheck	√	
Powerbuilder MS Access 2 Borland Delphi	Build Database Front Ends, Use Programming Language of Choice		√ √ √

Possible Applications of OCX Controls for R/3 RFCs

Summary of Programming Tasks

Your program's overall goal is to call a function in the R/3 System, sending data as input to the function and receiving data as return values. In the Function Control, sending and return parameters are presented as further objects contained in the Function object.

Any application using SAP Automation must perform the tasks listed below. Perform the first three steps for both methods of remote call access:

1. Create the base-object; i.e. the component itself.
2. Supply connection and logon information.
3. Open a connection to the R/3 System and log the user on.

If you are accessing remote function calls **non-dynamically**:

4. Request a Function object for the R/3 function you want.
5. Set the Export and Table parameter values.
6. Make the remote call.
7. Get the return values from the Import and Tables.

For accessing remote functions **dynamically** (in other words, you do not add the function to the functions collection, but write the function call like a native (local) VB function):

4. Create Table or Structure objects to place data into to pass data with the table parameters or structure parameters.
5. Invoke the function from the functions collection object.
6. Parameters can be either simple variables or object variables.
   Pass parameters to the named argument, for example:

*X*Func (X_1: = 5, X_2: = nVar, X_3: = objVar)

where X_1, X_2, and X_3 are the argument names in the function interface.

The following sections show how to program these tasks.

Example Application with the Function Control

In this example, we show an application that will use almost all the objects we discussed and several of their properties and methods. The application gets a list of customers from the R/3 System and prints attributesfor each customer (for example, name and ZIP-code). The function interface is shown here:

RFC_CUSTOMER_GET	IMPORT	*Name (NAME1)*
	IMPORT	*Customer-Number (KUNNR)*
	TABLES	*CUSTOMER_T (RFCKNA1)*

This function takes selection criteria (name and customer number) and retrieves the customers matching that description. Essentially, this function performs an SQL query:

```
select * fromCUSTOMER_T whereNAME1=Name
                        and  KUNNR=Customer Number
```

The table has the following structure, where all fields are of type RFC_CHAR:

Customer Table Structure:

KUNNR

ANRED

NAME1

PFACH

STRAS

PSTLZ

ORT01

TELF1

TELFX

The following example accesses the remote function by adding Function objects to the Functions collection object:

Declare object variables.

```
Dim Functions as Object
Dim GetCustomers as Object
Dim Customers as Object
```

Create the Function control (that is, the high-level Functions collection).

```
set Functions = CreateObject ("SAP.Functions")
```

Indicate what R/3 System you want to log on to.

```
Functions.Connection.Destination = "B20"
```

Set the rest of Connection object values.

```
......
```

Log on to the R/3 System.

```
Functions.Connection.Logon
if Functions.Connection.Logon (0, True) <> True then
  MsgBox "Cannot logon!"
End If
```

Retrieve the Function object. The Connection object must have been set up first before any Function objects can be created.

```
set GetCustomers = Functions.Add("RFC_CUSTOMER_GET")
```

Set the export parameters., Here, get all customers whose names start with J.

```
GetCustomers.Exports("NAME1") = "J*"
GetCustomers.Exports("KUNNR") = "*"
```

Call the function. If the result is not true, then display a message.

```
If GetCustomers.Call = True then
  There are two ways of accessing the table:
  Set Customers = GetCustomers.Tables(1)
  set Customers = GetCustomers.Tables("Customers")
  Print Customers (Customers.rowcount, "KUNNR")
  Print Customers (Customers.rowcount, "NAME1")
Else
  MsgBox "Call Failed! error: " + GetCustomers.Exception
End If
Functions.Connection.Logoff
```

A variation of this code can be used for dynamic calling. See *Variation using the Dynamic Calling Convention* .

Variation using the Dynamic Calling Convention

The dynamic calling technique looks different from a non-dynamic call, but performs the same function. The code is the same as that shown in *Example Application with the Function Control* , until the logon process is complete. Then, the following line is used:

Call the function with parameter "NAME1"

```
Functions.RFC_CUSTOMER_GET (Exception, NAME1:="J*",
KUNNR:="*", CUSTOMER_T:=Customers)
```

The customers table retrieved by the function has been copied into Customers variable. This way of calling makes the code a little easier to read, but takes slightly more time to process.

Creating the Base-level Control

Controls are usually made up of collections of objects. For example, the Function control contains Function objects, and the Transaction control contains Transaction objects. The highest (base) level of a control is either an actual collection object (like the Functions collection object) or a single control object (like the Table Factory object).

Controls also maintain some properties or objects common to all their sub-objects. Examples are the R/3 Connection (an object shared by all Functions or Transactions in a collection) or the Count property (the number of objects in a collection).

To create the base-level control, you use the *CreateObject* function and a fixed request string. This string specifies the kind of control (i.e. "SAP.TableFactory.1") and causes creation of the relevant base-level object. For example:

```
transactionsOCX = CreateObject("SAP.Transactions")
```

For the SAP OCX controls, the required fixed strings are:

CreateObject String

OCX control	Highest (base-level) object	Fixed String
Function control	Functions collection object	"SAP.Functions"
Transaction control	Transactions collection object	"SAP.Transactions"
Logon control	Logon object	"SAP.LogonControl"
Table Factory control	Table Factory object	"SAP.TableFactory"
Table View control	Table View object	"SAP.TableViewControl"
Table Tree control	Table Tree object	"SAP.TableTreeControl"

For more information, see *SAP Control Base Classes* .

Connecting to the R/3 System

The Connection object is a property of the both the Function and the Transaction controls. A Connection object is created automatically when you request the relevant collection for either the Function or Transaction control. The Logon component creates connections. If you have a Connection object obtained from another control or directly from the Logon OCX, you can set it in the Function or Transaction control.

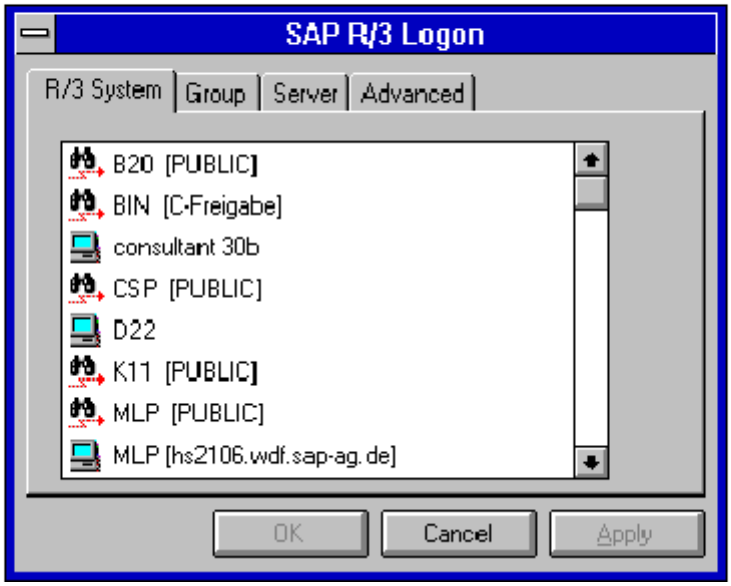
The Connection object's Logon method establishes the connection to the R/3 System. This method has a parameter that can suppress the dialog-box when the user logs in. This parameter allows you to automatically log the user into a fixed account or provide your own logon dialogs.

To establish a connection, you must call the Logon method for your Connection object. See *Using the Logon Control to Connect to R/3* .

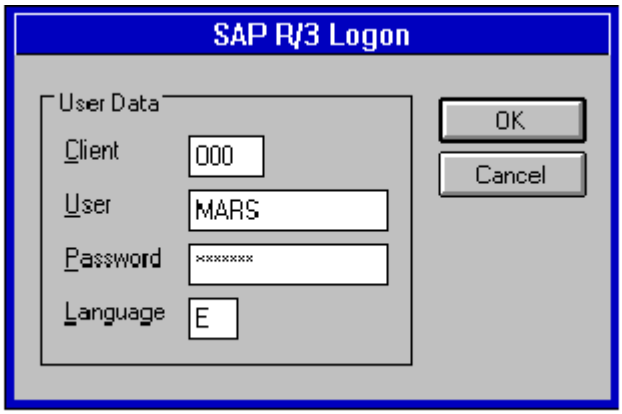
Using the Logon Control to Connect to R/3

Most components deal directly with R/3 and therefore need a connection to the application server. The basic steps needed to get a connection include creating a Logon object and then calling *NewConnection* method on that object. The result of this call is the connection object. To log on to the R/3 System you invoke the *Logon* method. The example code brings up the window shown:

```
 ‘ Create the Control.
Dim LogonControl As Object
set LogonControl = CreateObject (“SAP.LogonControl.1”)

‘ Create the connection.
Dim conn As Object
set conn  = LogonControl.NewConnection

‘ Log on.
if conn.Logon (0, True) <> True then
   MsgBox “Cannot log on!”
End If
```



You can set parameters for the logon process such as username and password. Consult the <LOGON control SECTION> for details. Depending on the parameters you set in the Logon control, one or two dialog boxes will be displayed that ask for the input needed to make the connection. One of these is:



Using the Function Control

After creating the Function control, you have to tell it about its connection to the R/3 System. Then you can start adding Function objects and calling RFCs. The Function control base-class is the Functions collection object. You can add and remove functions by using the *Add* and *Remove* methods.

Once a function has been created (added), it can be set up with parameter values and then called. The *Call* method of the Function object returns a boolean value that tells you whether the call executed with no problems. If there were problems, use the *Exception* property of the Function object to get more information on the error. If you have problems calling RFCs, consult the LogFileName and LogLevel properties of the Function control's base-class (the Functions collection object). These properties provide more information for trouble-shooting.

The following code creates the Function object, sets a connection, sets parameters, and calls the function.

```
' Create the component.
Dim functions As Object
Set functions = CreateObject ("SAP.Functions")

' Set the connection.
Set functions.Connection = conn

' Add (retrieve) function from R/3.
func = Functions.Add ("RFC_CUSTOMER_GET")

' Set a parameter.
func.Exports ("NAME1") = "ACME Steel"

' Call the function.
If Func.Call <> True then
   MsgBox "Call Failed. Exception" + func.exception
End If
```

For more information, see:

Requesting Functions

You must get a Functions collection object in order to access all other objects. Each Functions collection contains one Connection object. If your application needs to access data from multiple R/3 Systems, you must create a separate Function control for each system. Each of these components will have its own Connection object.

Adding a Function

Both ABAP/4 function modules and their separate parameters are represented by the object types Function, Parameter, Structure, and Table. You must get the Function object explicitly in order to be able to access the other object types. The Function control adds each function requested to the Functions collection.

Setting Parameter Values

Before calling an RFC function, you must set export and table parameter values. Depending upon the ABAP/4 function definition, you may not have to use all the formal variables specified in the interface. If you don't specify a value for a parameter, it is not sent with the call.

- To set values for import/export parameters defined as simple fields, use the Parameter object and its property functions.
- For import/export parameters defined as structures, use the Structure object and its properties to access individual fields.
- To set values for table parameters, use the Table, Rows and Row objects (and their properties) to access individual table rows and fields.

If the control cannot convert between ABAP/4 data types and your variable data types, it sends you an error message as soon as you fill the parameter.

See *Using Parameter and Structure Objects* for information on how to handle export and import parameters.

See *Viewing Table Objects* for information on how to handle table parameters.

Handling parameter objects

RFC objects "depend" on the parent object that contains them. If you assign the object to an object variable, the variable shares the object with the containing parent object. For example, after the statement:

```
Set ObjVar = FunctionsOCX ("RFC_PING")
```

`ObjVar` and **`FunctionsOCX ("RFC_PING")`** refer to (that is, point to) the same object. This is important because if you remove the object (or its containing parent) from the relevant collection, you simultaneously invalidate the contents of the object variable. (In the example, if you remove **`Funct1`** from its Functions collection, the variable **`ObjVar`** becomes invalid.)

There are two exceptions to this rule:

- Table objects that have been detached ("unloaded") from their containing Function objects, making the Table control the Table's new parent. (See *Viewing Table Objects* .)
- Collection objects that have been assigned to object variables.

Both of these become independent of the containing Function object. By contrast, you cannot detach Export/Import objects in this way. In addition, Table objects that are simply assigned (rather than unloaded) to a variable also remain shared.

Table and Structure objects are always independent of the Function objects when a remote function is called directly because there is no Function object created externally.

Viewing Table Objects

To set values for table parameters:

- To access fields in a row, use the Table.Item and Row.Value properties
- To process table rows as units, use the Table.Rows property to loop through all rows, or the Table.Item property for direct access to a single row.

The Row object is used to retrieve column information in the Table object. Do not attempt to use the Structure object methods on table rows; the Structure object is only for use with objects from the Exports and Imports collections.

Remember that if you assign a row to a object variable, and then delete either the Table object from the Tables collection object, or the row from the table (using the any of the RemoveRow, DeleteTable, or FreeTable methods), you invalidate the contents of the target object variable.

You can avoid this problem by detaching Table objects from the Function object. The table component provides the methods to unload and set for detaching and reattaching tables. A Table object unloaded to an object variable is no longer shared with the Function object. However, Table objects that are simply assigned (rather than unloaded) to a variable remain shared.

Using Parameter and Structure Objects

When you want to access fields in an export/import parameter defined as a structure, use the following:

1. Assign the parameter to an object variable:

   ```
   set StructObj = MyFunct.Exports("Employee_Struct")    OR
   Set FieldObj  = MyFunct.Exports("Company_Number")
   ```

2. Use either Parameter or Structure functions on the variable, depending on whether the parameter is defined as a field or structure.

   ```
   StructObj.Value("Name") = "Smith"  OR
   FieldObj.Value = "1234"
   ```

Structure objects are provided to perform operations on export and import parameters only. Do not attempt to use the methods and properties for this object type with table rows.

Structure and Parameter objects are fundamentally different from the other object types. They are not maintained in their own collection lists, and do not occur in any other object type as a property or method. As a result, expressions of the form:

```
MyFunction.Structure.Value("field-name") OR
MyFunction.Parameter.Value
```

are not valid and result in runtime errors.

To get information on a parameter's structure definition, use the online Assistant.

Using Named–Argument Calling Conventions

The named-argument calling conventions do not have qualifiers to distinguish the importing parameters from the exporting parameters like the calling conventions used in the ABAP language. You must be aware which parameter is the importing parameter and which parameter is the exporting parameter. From a caller's point of view, you can use either variables or constants for exporting parameters. However, you can only use variables for importing parameters so that the variables can store the returned data.

From the caller's point of view, if the exporting parameter is a structure and you want to pass data to this parameter, you must first create a Structure object using the CreateStructure method of the Functions collection and fill data in the Structure object. If the importing parameter is a structure, you can pass any variable to receive the returned Structure object. There is no need to create Structure objects yourself.

If you only want to retrieve data in table parameters, you can use any variable for the table parameters. There is no need to create Table objects yourself. The remote Function object will create the Table objects and store them into your variables. If you want to pass data to the table parameters, you must first create a Table object in the table component and fill data to the able object. Then you can pass the table object into the table parameter.

The following example illustrates these two scenarios.

Example

For the remote function interface:

```
xFunc
   Importing   IP     LIKE   TP-IP
               SIP    LIKE   SX STRUCTURE SX
   Exporting   EP     LIKE   TP-EP
               SEP    LIKE   SY STRUCTURE SY
   Tables      TP     STRUCTURESZ
```

In the first scenario, when you only want to retrieve data, the following VBA code illustrates the calling statement:

Call the xFunc remote function.

```
R3.xFunc IP:= 1, SEP:= objStruct, EP:= nVar,  TP:=objTable
```

where **objStruct**, **nVar**, and **objTable** can be uninitialized.

In the second scenario, when you want to pass data to the table parameter and the exporting parameter with the structure type, the following VBA code illustrates the calling statement:

Create a Structure object with structure type “SX”.

```
set objMyStruct = R3.CreateStructure(“SX”)
```

Create a Table object with table structure type “SZ”.

```
set objTable = R3.CreateTable(“SZ”)
```

Fill in data for **objMyStruct** and **objTable** here.

```
...
```

Call the xFunc remote function.

```
R3.xFunc IP:=1, SIP:= objMyStruct, EP:= nVar, SEP:= objStruct,
TP:=objTable
```

Using the Table Factory Control

Each Function object owns a collection of R/3 tables. The Table Factory control gives you easy access to these SAP internal tables. If used from within the Function control, Table Factory is invisible to the user. The Table Factory control provides a Tables collection object that is used as a property in the Function object. When you access the function’s Tables collection, Visual Basic™ gets a “dispatch pointer” to the Table control. In the previous example, the *RFC_CUSTOMER_GET* call returns an internal table with the name “CUSTOMER_T”. To get the last row of that table and display the ZIP code, for example, use code as shown:

```
‘ Get table.
Set customers = func.Tables (“CUSTOMER_T”)

‘ Get zipcode of last row.
zipCode = customers (customers.RowCount, "PSTLZ")
```

Using the Transaction Control

Most of the day, when communicating with users, the R/3 System executes transactions. These are made up of a sequence of screens that contain fields. The online user fills in the fields, presses a button and the next screen appears. The Transaction control allows you to programmatically fill in these fields and proceed through the screens. No GUI is displayed.

You create a Transactions collection object just like a Functions collection object:

```
Set transOCX = CreateObject("SAP.Transactions.1")
```

Add a Transaction object:

```
Set trans = transOCX.Add("SE11", "DOMAIN")
```

Then you add screens and fields:

```
  Set Screen = trans.Screens.Add
  Screen.Program = "SAPMSRD0"
  Screen.Number = "0100"
  Set Fields = Screen.Fields
  Fields.Add "RSRD1-OBJNAME", "ZTST"
```

```
Fields.Add "RSRD1-DOMA", "X"
Fields.Add "BDC_OKCODE", "=ADD"

Set Screen = trans.Screens.Add
Screen.Program = "SAPMSD01"
Screen.Number = "0100"
Set Fields = Screen.Fields

Text = "Testing at " + Str$(Hour(Now)) + ":" + Str$(Minute(Now))
Fields.Add "DD01V-DDTEXT", Text
Fields.Add "DD01V-DATATYPE", "CHAR"
Fields.Add "DD01V-LENG", "10"
Fields.Add "BDC_OKCODE", "/11"

Set Screen = trans.Screens.Add
Screen.Program = "SAPLSTRW"
Screen.Number = "0100"
Screen.Fields.Add "BDC_OKCODE", "/9"
```

Don’t forget to set the Connection! Here we’ll just reuse the one from above:

```
set transOCX.Connection = conn
```

Finally we call the transaction:

```
if trans.Call <> true then
   MsgBox “Call failed…”
End If

‘ Display the message generated.
MsgBox "Transaction message: " + trans.Message.Value("MSGTX")
```

Remember that with the Transaction control, you are always calling the transaction in batch-input mode. That is, the external program can send input field values to an R/3 screen, but output field values are not returned. For complete transaction execution, with data transfer in both directions, use the *BAPI Component*.

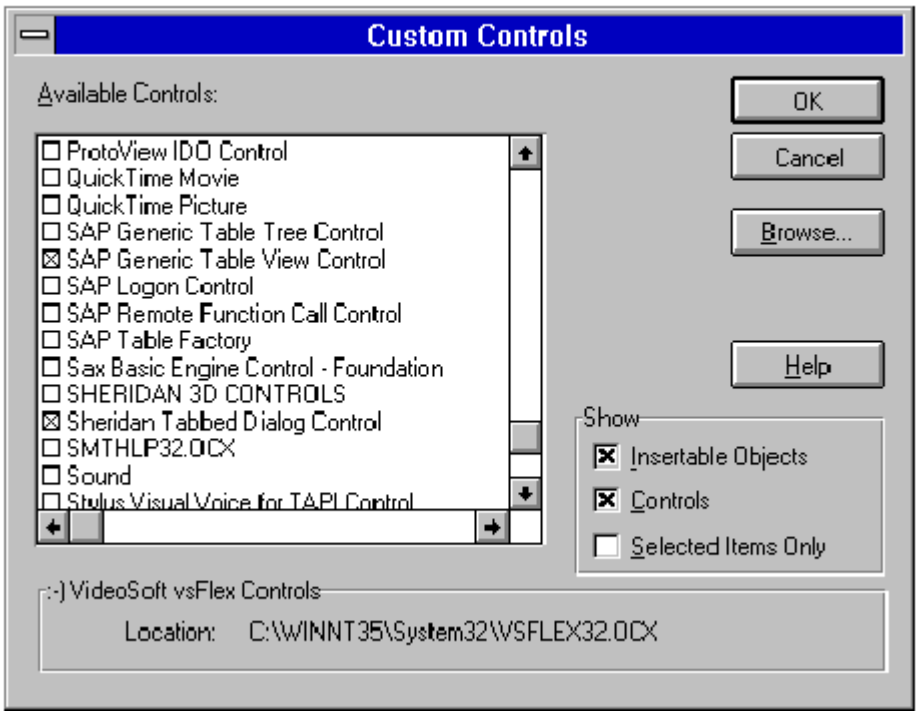
Using the Table View Control

The Table View control provides views of tables that have been retrieved from R/3. It also allows editing of the data shown in the view and automatically gets the table contents updated. You get access to a Table View control by using a variable. Visual Basic (Version >= 4.0) allows this most easily.

To bind a table view to a table, you must get the table (e.g. customers), the view (e.g. SAPTableView), and then notify the table that it now has a view. For example:

```
‘ Establish view-table connection.
Customers.Views.Add SAPTableView.Object
Customers.Refresh
```

A complete recipe follows for displaying a table from Visual Basic. The first step is to drop a table view onto a form and set its name.



Then, after you have retrieved the table via an RFC, you connect it to the view. In Visual Basic, create a form, then select *SAP Generic Table View Control* in the *Custom Control* dialog.

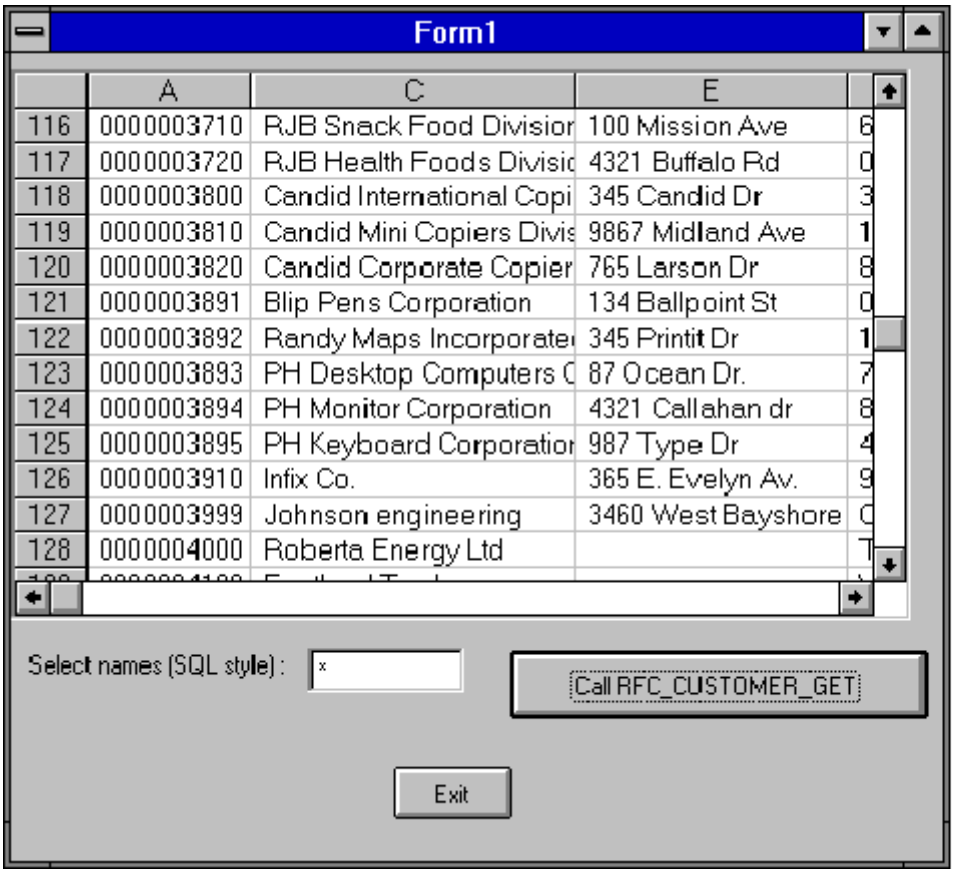
An icon will be placed in the toolbox  . Click on this icon, and then create the control in the area on the form where you want the table data displayed. Change the name of the new object (in the *Properties* window) to SAPTableView. (The name is optional, as long as you use the same name in the script.)

Create a text-entry field, and name it (for example) NameInput. Then add a button and add the following code to it (the click callback function):

```
Private Sub Command1_Click()

' Create function component.
Dim fns As Object
Set fns = CreateObject("SAP.Functions.1")
fns.logfilename = "c:\tmp\table+viewlog.txt"
fns.loglevel = 8

Dim conn As Object
Set conn = fns.Connection
conn.Client = "000"
conn.Language = "E"
conn.tracelevel = 6
```

```
if conn.logon(0, 0) <> True then
   MsgBox "Could not logon!"
End If

Dim Customers, Customer As Object
Dim Result As Boolean

' Get the name to display from NameInput and call function …
Result = fns.RFC_CUSTOMER_GET(Exception, NAME1:=NameInput, KUNNR:="*",
CUSTOMER_T:=Customers)

If Result <> True Then
  MsgBox ("Call error: " + Exception)
  Exit Sub
Else
  ' try to display table view.
  Customers.views.Add TheTableView.object
  Customers.Refresh
  MsgBox "Got " + Str$(Customers.RowCount) + " rows."
End If

Set fns = Nothing
Set conn = Nothing
End Sub
```

The resulting form (after the script shown in this section has run) should resemble the following:



Using Collection Objects

The SAP Assistant defines several *collection* object types. Collection objects gather all objects of a given type into a list. (For example, all Function objects for a given control are gathered into the Functions collection object representing that control.)

A collection object provides list–oriented functions for accessing list objects, adding and removing from the list, looping through the list, and so on.

Example

To loop through a Rows collection object:

```
For Each Customer in Customers.Rows
  print Customer ("NAME")
Next Customer
```

Performance and Debugging Tips

The following sections describe techniques you can use to improve the performance and reduce debugging time for your applications.

Avoiding Unnecessary Object Creation

Application objects and all collection objects are created dynamically when your code makes a reference to one of them. These objects are temporary and are destroyed when no more references to them exist. As a result, multiple accesses to one of these objects can hurt performance. To avoid this problem, assign the object to an object variable of your own and then make the accesses. For example, you can improve the following code:

```
GetFunct.Exports("P1")
GetFunct.Exports("P2")
GetFunct.Exports("P3")
GetFunct.Exports("P4")
GetFunct.Exports("P5")
```

by changing it to:

```
set MyExports = GetFunct.Exports
MyExports("P1")
MyExports("P2")
MyExports("P3")
MyExports("P4")
MyExports("P5")
```

In the first code example, five temporary collection objects are created, each destroyed after a single statement. In the second, only one temporary object is created. (This object is not destroyed after the first statement, because the other statements still refer to it.)

Tracing RFC Calls

You can request a trace of connection activity as the RFC call executes. The TraceLevel property in the Connection object lets you specify tracing. Possible values are 0 (tracing not requested) and 1 (tracing requested).

The activity information is logged in a file "RFC<random number>.TRC" located in the active default directory.

The Functions collection and Transactions collection objects also provide logging functionality with the LogFileName and LogLevel properties.

Note on Embedded Property Calls

Bear in mind while coding your application that client languages may execute certain kinds of statements differently. Of particular importance is whether or not your language performs cascaded evaluation. An example of cascaded evaluation is the evaluation of statements like:

```
MyFunct.Exports("P1").Value("F1")
```

This statement requires the language to first call the Exports property for MyFunct, and when a Structure object is returned, call the Value property on the Structure object.

Some languages do not perform full evaluation of statements like this. They evaluate the first call (the Exports property), but do not evaluate the returned value (a Parameter or Structure object) to call the next property (Value method) on it. As a result, you get the wrong object returned, and eventually a runtime error.

The Excel macro language executes the above statement correctly, while Visual Basic 3.0 does not. However, almost all interpreters fail to evaluate statements correctly when a default function is left implicit:

```
GetFunct.Exports("P1")("F1")
```

Note on Default Property Calls

Some languages do not evaluate the default value property. In this case, the default value property must be explicitly specified:

```
MyFunct.Exports("P1")        ' Does not work.'
MyFunct.Exports.Item ("P1")  ' OK'
```

SAP Control Base Classes

The following reference topics on base classes are available:

SAP Standard Collection

SAP Named Collection

Using Collection Objects

SAP Data Object

Safe Arrays and Values

Font Objects

Data Types

SAP Standard Collection

All active OLE controls containing collections within an SAP System are implemented according to common conventions. As long as a collection is not explicitly declared as a non-standard collection, the following description applies to the collection. Not all properties and methods mentioned below are available with every collection.

The lower bound index for all collections is 1.

For hints on using collection objects, see *Using Collection Objects* .

Standard Collection Properties

Name	Parameter	Type	Description
Count	void	Long	Returns the number of objects stored in the collection.
Item	Variant vaIndex	Object	Returns an object according to *vaIndex.* Item is always the default property.

Standard Collection Methods

Name	Parameter	Return Type	Description
Add	Object dependent	Object	Adds a new object and returns the new object.
Insert	Variant vaIndex Obj	Object	Inserts a new object at position *vaIndex* and returns the new object.

	ect dep end ent		
Re mo ve	Var iant vaI nde x	B o o le a n	Removes the object at position *vaIndex.*
Re mo ve All	voi d	B o o le a n	Removes all objects from the collection.
Un Loa d	Var iant vaI nde x	O b je ct	Unloads the object at position *vaIndex.*

Detailed Description

Object Item(Variant vaIndex)

Item returns an object from the collection. The parameter *vaIndex* identifies the position of the object to be returned. The type of this parameter depends on the object. It may describe the position where the object can be found, either as a simple integer value, or as a string value (as described in *Named Collections*), or in any object-dependent variant data type. In the following sections, the valid types are described for each collection.

Object Add (…)

The parameters for the Add method depend on the object. These parameters are used to initialize the new object. Add always returns the new object.

Object Insert(Variant vaIndex,…)

The parameters for the Insert method depend on the object. These parameters are used to initialize the new object. Insert always returns the new object. The first parameter of the Insert methods always describes the position where to insert the new object (the new object is always inserted in front of the position described by vaIndex). The type of this parameter is object-dependent. It may describe the position where to insert the new object, either as a simple integer value, or as a string value (as described in *Named Collections*), or as an Object which is already part of the collection. Nevertheless, the indexing parameter always has the same meaning as the indexing parameter of the default property *Item.*

Boolean Remove(Variant vaIndex)

Removes an object from its collection. The parameter vaIndex identifies the position of the object to be returned. The type of this parameter depends on the object. It may describe the position where to insert the new object, either as a simple integer value, or as a string value (as described in *Named Collections*). Nevertheless, the indexing parameter always has the same meaning as the indexing parameter of the default property *Item.*

Caution

When removing an object from the collection, the object becomes **invalid**. Any further attempts to work on the object return an Invalid Object Exception. Use UnLoad if the object should be removed from the collection for further use.

Object UnLoad(Variant *vaIndex*)

Unloads an object from its collection. The parameter vaIndex identifies the position of the object to be returned. The type of this parameter depends on the object. It may describe the position where to unload the object, either as a simple integer value, or as a string value (as described in *Named Collections*). Nevertheless, the indexing parameter always has the same meaning as the indexing parameter of the default property *Item.*

SAP Named Collection

Named collections are derived from *SAP Standard Collections*. In addition, a named collection may always work with a string as indexing parameter for methods like Item, Insert, Remove and UnLoad. Objects within a named collection always have a Name property which stores the indexing name. The name describing an object in a named collection need not to be unique. If a name is used frequently, Item, Insert, Remove and UnLoad always use the first object with the given name.

Further features of named collections are dynamic properties, created as a result of the objects' names. Instead of invoking the Item property, an object may also be returned if the name of the object is used as property.

For further hints on using collection objects, see *Using Collection Objects* .

Example

```
Dim oObj as Object
' Add a new empty object
Set oObj = NamedCollectionObject.Add ()
' Assign name
oObj.Name = "ItemB"
' Add object an pass name as parameter.
Set oObj = NamedCollectionObject.Add "ItemC")
' Insert  object in prior to object "ItemB"
```

```
Set oObj = NamedCollectionObject.Insert("ItemB","ItemA")

' Accessing the object
' Retrieve second object through index
Set oObj  = NamedCollectionObject.Item(2)
' Retrieve second object through name
Set oObj  = NamedCollectionObject.Item("ItemB")
' Retrieve second object through dynamic property
Set oObj  = NamedCollectionObject.ItemB
```

SAP Data Object

The SAP Data Object is a special object for purposes of data transport. This object is used in *drag and drop* and clipboard operations. The SAP Data Object implements an automation and an IDataObject interface. The IDataObject interface is a standard OLE interface for data object manipulation. The automation interface displays the following methods:

Name	Parameters		Return Type	Description
GetData	Long Variant	cfFormat vaData	void	Retrieves data from the data object in the specified format
SetData	Long Variant	cfFormat vaData	void	Stores data in the data object in the specified format
IsFormatAvailable	Long	cfFormat	Boolean	Returns TRUE if the data object contains data in the specified format

Caution

SetData may specify a format which is not initially cached in the data object. Since the data object does not interpret the data in any way, the data may be of any clipboard format. The variant data type must be any data type which is transportable to other processes. Therefore, object may not be stored in the data object.

Valid formats are: char, short, long, String, Date, Time, Boolean and safe arrays of these types.

Example

```
Sub DragSourceFill(DataObject As Object)
     Dim cfFormat As Long
     Dim Data As String

     cfFormat = RegisterClipboardFormat("MyClipFormat");
     Data = "This is my personal string"
     DataObject.SetData(cfFormat,Data)
Sub End

Sub Drop(DataObject As Object)
     Dim cfFormat As Long
     Dim Data As String

     cfFormat = RegisterClipboardFormat("MyClipFormat");
     if  DataObject.IsFormatAvailable(cfFormat) then
          DataObject.GetData(cfFormat,Data)
          MsgBox(Data)

     end if
Sub End
```

Safe Arrays and Values

If an SAP object returns data as a safe array, the object always has the property Data. This property usually returns a two-dimensional safe array with a lower bound index of 1 for each dimension. An example of a Data property would be the entire content of a table or the entire content of a row or column in a table. Single date like the content of a cell in a table is always returned as a Variant. The corresponding property is always called Value. If an object implements a Value property, this property is the default property.

Font Objects

Font Properties:

Table Caption

Name	**Type**
Name	String
Size	Currency

Bold	Boolean
Italic	Boolean
Underline	Boolean
StrikeThrough	Boolean
Weight	Short
CharSet	Short

For more information, see the VBA help on font objects.

Data Types

The following are the available data types:

Table Caption

Types in Help File	VBA Data Type	C++ Data Type
Char	Byte (By Val)	unsigned char
Short	Integer (By Val)	short
Long	Long (By Val)	long
String	String (By Val)	BSTR
Boolean	Bool (By Val)	VT_BOOL
Object	Object	IDispatch *
Short*	Integer	short*
Long*	Long	long*
String*	String	BSTR*
Boolean*	Bool	VT_BOOL*
void	Method does not return a value	void
Array of *type*	(1, n) of *Type*	VT_ARRAY | VT_*Type*