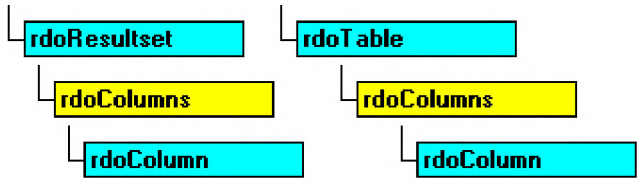
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoColumn Object

See Also Example Properties Methods Events

An **rdoColumn** object represents a column of data with a common data type and a common set of properties.



Remarks

The **rdoTable**, or **rdoResultset** object's **rdoColumns** collection represents the **rdoColumn** object in a row of data. You can use the **rdoColumn** object in an **rdoResultset** to read and set values for the data columns in the current row of the object. However, in most cases, references to the **rdoColumn** object is only implied because the **rdoColumns** collection is the **rdoResultset** object's default collection.

An **rdoColumn** object's name is determined by the name used to define the column in the data source table or by the name assigned to it in an SQL query. For example, if an SQL query aliases the column, this name is assigned to the **Name** property; otherwise, the column's name is used.

You manipulate database columns using an **rdoColumn** object and its methods and properties. For example, you can:

- Use the **Value** property of an **rdoColumn** to extract data from a specified column.
- Use the **Type** and **Size** property settings to determine the data type and size of the data.
- Use the **Updatable** property to see if the column can be changed.
- Use the **SourceColumn** and **SourceTable** property settings to locate the original source of the data.
- Use the **OrdinalPosition** property to get presentation order of the **rdoColumn** objects in an **rdoColumns** collection.
- Use the **Attributes** and **Required** property settings to determine optional characteristics and if Nulls are permitted in the column.
- Use the **AllowZeroLength** property to determine how zero-length strings are handled.
- Use the **BatchConflictValue,** and **OriginalValue** properties to resolve optimistic batch update conflicts.
- Use the **KeyColumn** to determine if this column is part of the primary key.
- Use the **Status** property to determine if the column has been modified.
- Use the **AppendChunk, ColumnSize**, and **GetChunk** methods to manipulate columns that require the use of these methods, as determined by the **ChunkRequired** property.

When you need to reference data from an **rdoResultset** column, you can refer to the **Value** property of an **rdoColumn** object by:

- Referencing the **Name** property setting using this syntax:

```
' Refers to the Au_Fname column rdoColumns("Au_Fname")
rs.rdoColumns("Au_Fname")
```

-Or-

```
' Refers to the Au_Fname column
rs.rdoColumns!Au_Lname
```

- Referencing its ordinal position in the **rdoColumns** collection using this syntax:

```
rs.rdoColumns(0)
```

The **rdoTable** object's **rdoColumns** collection contains specifications for the data columns. You can use the **rdoColumn** object of an **rdoTable** object to map a base table's column structure. However, you cannot directly alter the structure of a database table using RDO properties and methods. You can, however, use data definition language (DDL) action queries to modify database schema.

When the **rdoColumn** object is accessed as part of an **rdoResultset** object, data from the current row is visible in the **rdoColumn** object's **Value** property. To manipulate data in the **rdoResultset**, you don't usually reference the **rdoColumns** collection directly. Instead, use syntax that references the **rdoColumns** collection as the default collection of the **rdoResultset**.

```
dim rs As rdoResultset
Set rs = cn.OpenResultset("Select * from Authors" _
   & "Where Au_Lname = 'White'",rdOpenForwardOnly)
debug.print rs!Au_Fname
   'Refers to rdoRecordset object's rdoColumns collection.
```

Visual Basic: RDO Data Control

rdoColumn Object, rdoColumns Collection Example

The following example opens a connection against an SQL Server database and creates an **rdoResultset** that returns two columns: one normal column, and one derived from an expression. Next, the example maps the **rdoColumn** objects returned from the result set.

```
Private Sub rdoColumnButton_Click()
Dim cl As rdoColumn
Dim rs As rdoResultset
Dim sSQL As String
Dim cn As rdoConnection
Dim connect As String

connect = "uid=;pwd=;database=pubs;"

Set cn = rdoEnvironments(0).OpenConnection(workdb, _
             rdDriverNoPrompt, False, connect)

sSQL = "Select Pub_ID, Max(Price) BestPrice " _
    & " from Titles Group by Pub_ID"

Set rs = cn.OpenResultset(sSQL, rdOpenForwardOnly, _
    rdConcurReadOnly)

With rs
    For Each cl In .rdoColumns
        Print cl.Name; "-"; cl.Type; ":"; cl.Size, _
         cl.SourceTable, cl.SourceColumn
    Next cl
    Print
    Do Until .EOF
        For Each cl In .rdoColumns
            Print cl.Value,
        Next cl
        Print
        .MoveNext
    Loop
End With
End Sub
```

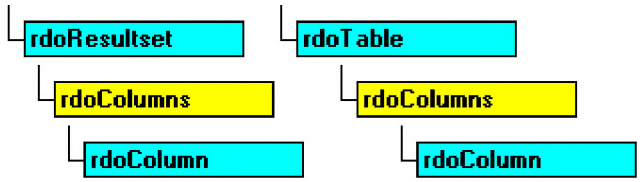
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoColumns Collection

See Also Example Properties Methods Events

An **rdoColumns** collection contains all **rdoColumn** objects of an **rdoResultset**, or **rdoTable** object.



Remarks

The **rdoTable**, or **rdoResultset** object's **rdoColumns** collection represents the **rdoColumn** objects in a row of data. You use the **rdoColumn** object in an **rdoResultset** to read and set values for the data columns in the current row of the object.

The **rdoColumn** object is either created automatically by RDO when

- An **rdoTable**, or **rdoResultset** object is created.
- An **rdoTable** object is referenced.
- An **rdoResultset** is created via OpenResultset.

Visual Basic: RDO Data Control

rdoColumn Object, rdoColumns Collection Example

The following example opens a connection against an SQL Server database and creates an **rdoResultset** that returns two columns: one normal column, and one derived from an expression. Next, the example maps the **rdoColumn** objects returned from the result set.

```
Private Sub rdoColumnButton_Click()
Dim cl As rdoColumn
Dim rs As rdoResultset
Dim sSQL As String
Dim cn As rdoConnection
Dim connect As String

connect = "uid=;pwd=;database=pubs;"

Set cn = rdoEnvironments(0).OpenConnection(workdb, _
             rdDriverNoPrompt, False, connect)

sSQL = "Select Pub_ID, Max(Price) BestPrice " _
    & " from Titles Group by Pub_ID"

Set rs = cn.OpenResultset(sSQL, rdOpenForwardOnly, _
    rdConcurReadOnly)

With rs
    For Each cl In .rdoColumns
        Print cl.Name; "-"; cl.Type; ":"; cl.Size, _
         cl.SourceTable, cl.SourceColumn
    Next cl
    Print
    Do Until .EOF
        For Each cl In .rdoColumns
            Print cl.Value,
        Next cl
        Print
        .MoveNext
    Loop
End With
End Sub
```

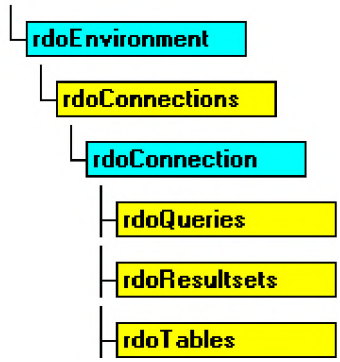
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoConnection Object

See Also Example Properties Methods Events

An **rdoConnection** object represents an open connection to a remote data source and a specific database on that data source, or an allocated but as yet unconnected object, which can be used to subsequently establish a connection.



Remarks

Generally, an **rdoConnection** object represents a physical connection to the remote data source and corresponds to a single ODBC **hDbc** handle. A connection to a remote data source is required before you can access its data. You can open connections to remote ODBC data sources and create **rdoConnection** objects with either the **RemoteData** control or the **OpenConnection** method of an **rdoEnvironment** object.

To establish a connection to a remote server using the **rdoConnection** object, you can use the **OpenConnection** method to gather the ***connect***, ***dsname***, ***readonly*** and ***prompt*** arguments and open the connection. These arguments are then applied to the newly created **rdoConnection** object. You can also establish connections using the **RemoteData** control.

Creating Stand Alone rdoConnection Objects

You can also create a new **rdoConnection** object that is *not* immediately linked with a specific physical connection to a data source. For example, the following code creates a new stand-alone **rdoConnection** object:

```
Dim X as new rdoConnection.
```

Once created, you can set the properties of a stand-alone **rdoConnection** object and subsequently use the **EstablishConnection** method. This method determines how users are prompted based on the ***prompt*** argument, and sets the read-only status of the connection based on the ***readonly*** argument.

When using this technique, RDO sets the following properties based on **rdoEngine** default values: **CursorDriver**, **LoginTimeout**, **UserName**, **Password** and **ErrorThreshold**. The **CursorDriver** and **LoginTimeout** properties can be set in the **rdoConnection** object itself and the **UserName** and **Password** can be set through arguments in the connect string. Once the connection is open, all of these properties are read-only.

When you declare a stand-alone **rdoConnection** object or use the **EstablishConnection** method, the object is not automatically appended to the **rdoConnections** collection. Use the **Add** or **Remove** methods to add or delete stand-alone

rdoConnection objects to or from the **rdoConnections** collection. It is not necessary, however to add an **rdoConnection** object to the **rdoConnections** collection before it can be used to establish a connection.

Note RDO 1.0 collections behave differently than Data Access Object (DAO) collections. When you **Set** a variable containing a reference to a RDO object like **rdoResultset**, the existing **rdoResultset** is *not* closed and removed from the **rdoResultsets** collection. The existing object remains open and a member of its respective collection.

In contrast, RDO 2.0 collections do not behave in this manner. When you use the Set statement to assign a variable containing a reference to an RDO object, the existing object *is* closed and removed from the associated collection. This change is designed to make RDO more compatible with DAO.

Asynchronous Operations

Both the **EstablishConnection** and **OpenConnection** methods support synchronous, asynchronous, and event-managed operations. By setting the **rdAsyncEnable** option, control returns to your application *before* the connection is established. Once the **StillConnecting** property returns **False**, and the **Connect** event fires, the connection has either been made or failed to complete. You can check the success or failure of this operation by examining errors returned through the **rdoErrors** collection.

Opening Connections without Data Source Names

In many situations, it is difficult to ensure that a registered Data Source Name (DSN) exists on the target system, and in some cases it is not advisable to create one. Actually, a DSN is not needed to establish a connection if you are using the default network protocol (named pipes) and you know the name of the server and ODBC driver. If this is the case, you can establish a *DSN-less* connection by following these steps:

1. Set the ***DSN*** argument of the connect string to an empty string (DSN='').
2. Include the server name in the connect string.
3. Include the ODBC driver name in the connect string. Since many driver names have more than one word, enclose the name in curly braces { }.

Note This option is not available if you need to use other than the named pipes network protocol or one of the other DSN-set options such as OEMTOANSI conversion.

For example, the following code opens a read-only ODBC cursor connection against the SQL Server "SEQUEL" and includes a simple error handler:

```
Sub MakeConnection()
Dim rdoCn As New rdoConnection
On Error GoTo CnEh
With rdoCn
    .Connect = "UID=;PWD=;Database=WorkDB;" _
        & "Server=SEQUEL;Driver={SQL Server}" _
        & "DSN='';"
    .LoginTimeout = 5
    .CursorDriver = rdUseODBC
    .EstablishConnection rdDriverNoPrompt, True
End With
Exit Sub
CnEh:
Dim er As rdoError
    Debug.Print Err, Error
    For Each er In rdoErrors
        Debug.Print er.Description, er.Number
    Next er
    Resume Next
End Sub
```

Choosing a Specific Database

Once a connection is established, you can manipulate a database associated with the **rdoConnection** using the **rdoConnection** object and its methods and properties. For servers that support more than one database per connection, the default database is:

- Assigned to the user name by the database system administrator
- Specified with the DATABASE connect argument used when the **rdoConnection** is created.
- Specified in the registered ODBC data source entry.
- Selected by using an SQL statement such as USE <database> submitted with an action query.

All queries executed against the server assume this default database unless another database is specifically referenced in your SQL query.

Preparing for Errors when Connecting

There are a variety of reasons why you might be unable to connect to your remote database. Consider the following conditions that can typically prevent connections from completing:

- Your server might not have sufficient connection resources due to administrative settings or licensing restrictions.
- Your user might not have permission to access the network, server, or database with the password provided.
- The server, network or WAN bridges might be down or simply running slower than expected.

Closing the rdoConnection

When you use the **Close** method against an **rdoConnection** object, any open **rdoResultset,** or **rdoQuery** objects are closed. However, if the **rdoConnection** object simply loses scope, these objects remain open until the **rdoConnection** or the objects are explicitly closed. Closing a connection is not recommended when there are incomplete queries or uncommitted transactions pending.

Closing a connection also removes it from the **rdoConnections** collection. However, the **rdoConnection** object itself is not destroyed. If needed, you can use the **EstablishConnection** method to re-connect to the same server using the same settings, or change the **rdoConnection** object's properties and then use **EstablishConnection** to connect to another server.

Closing a connection also instructs the remote server to discard any instance-specific objects associated with the connection. For example, server-side cursors, temporary tables or any other objects created in the *TempDB* database on SQL Server are all dropped.

Working with rdoConnection Methods and Properties

You can manipulate the connection, databases, and queries associated with them using the methods and properties of the **rdoConnection** object. For example, you can:

- Use the **CursorDriver** property to determine the type of cursor requested by result sets created against the connection.
- Use the **OpenResultset** method to create a new **rdoResultset** object.
- Use the **LastQueryResults** to reference the last **rdoResultset** created against this connection.
- Use the **QueryTimeout** or **LoginTimeout** properties to specify how long the ODBC driver manager should wait before abandoning a query or connection attempt.

- Use the **RowsAffected** property to determine how many rows were affected by the last action query.
- Use the **Execute** method to run an action query or pass an SQL statement to a database for execution.
- Use the **CreateQuery** method to create a new **rdoQuery** object.
- Use the **Close** method to close an open connection, remove the **rdoConnection** object from the **rdoConnections** collection, deallocate the connection handle, and terminate the connection.
- Use the **Transactions** property to determine if the connection supports transactions, which you can implement using the **BeginTrans**, **CommitTrans**, and **RollbackTrans** methods.
- Use the **AsyncCheckInterval** property to determine how often RDO should poll for a completed asynchronous operation.
- Use the ODBC API with the **hDbc** property to set connection options.
- Use the **Connect** property to determine the ***connect*** argument used in the **OpenConnection** method, or the **Connect** property of the **RemoteData** control.

rdoConnection Events

The following events are fired as the **rdoConnection** object is manipulated. These can be used to micro-manage the process of connecting and disconnecting and provide additional retry handling in query timeout situations.

Event Name	Description
BeforeConnect	Fired before ODBC is called to establish the connection.
Connect	Fired after a connection is established.
Disconnect	Fired after a connection has been closed
QueryComplete	Fired after a query run against this connection is complete
QueryTimeout	Fired after the QueryTimeout period is exhausted.

Addressing the rdoConnection Object

The **Name** property setting of an **rdoConnection** specifies the data source name (DSN) parameter used to open the connection. This property is often empty as it is not used when making a DSN-less connection. In cases where you specify a different DSN to open each connection, you can refer to any **rdoConnection** object by its **Name** property setting using the following syntax. This code Refers to the connection opened against the *Accounting* DSN:

```
rdoConnections("Accounting")
```

You can also refer to the object by its ordinal number using this syntax (which refers to the first member of the **rdoConnections** collection):

```
rdoConnections(0)
```

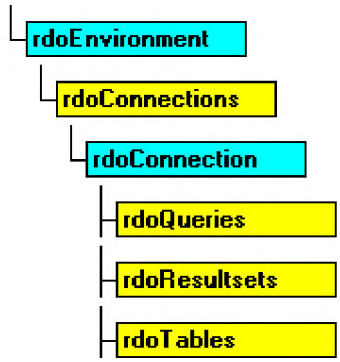
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoConnections Collection

See Also Example Properties Methods Events

An **rdoConnections** collection contains all **rdoConnection** objects opened or created in an **rdoEnvironment** object of the remote database engine, or allocated and appended to the **rdoConnections** collection using the **Add** method.



Remarks

The **rdoConnections** collection is used to manage your **rdoConnection** objects. However, only **rdoConnection** objects created using the **OpenConnection** method, or using the **RemoteData** control are automatically appended to the collection. When you allocate a stand-alone **rdoConnection** object, it is not appended to the **rdoConnections** collection until you use the **Add** method.

Note RDO 1.0 collections behave differently than Data Access Object (DAO) collections. When you **Set** a variable containing a reference to a RDO object like **rdoResultset**, the existing **rdoResultset** is *not* closed and removed from the **rdoResultsets** collection. The existing object remains open and a member of its respective collection.

In contrast, RDO 2.0 collections do not behave in this manner. When you use the Set statement to assign a variable containing a reference to an RDO object, the existing object *is* closed and removed from the associated collection. This change is designed to make RDO more compatible with DAO.

Closing rdoConnection Objects

When you use the **Close** method against an **rdoConnection** object, any open **rdoResultset,** or **rdoQuery** objects are closed and the **rdoConnection** object is removed from the **rdoConnections** collection. However, if the **rdoConnection** object simply loses scope, these objects remain open until the **rdoConnection** or the objects are explicitly closed.

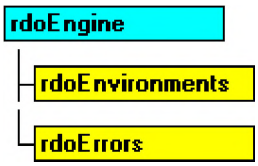
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoEngine Object

See Also Example Properties Methods Events

The **rdoEngine** object represents the remote data source. As the top-level object, it contains all other objects in the hierarchy of Remote Data Objects (RDO).



Remarks

The **rdoEngine** object can represent a remote database engine or another data source managed by the ODBC driver manager as a database. The **rdoEngine** object is a predefined object, therefore you can't create additional **rdoEngine** objects and it isn't a member of any collection.

The **rdoEngine** object is used to reference the **rdoEnvironments** collection, or establish default values for newly created **rdoEnvironment** objects. When an **rdoEnvironment** object is created, its properties are initialized based on the default values set in the **rdoEngine**. A default **rdoEnvironments**(0) object is created automatically when it is first referenced.

The **rdoEngine** object fires the InfoMessage event when an informational message is returned from the remote data source. Informational messages are indicated by an ODBC SQL_SUCCESS_WITH_INFO return code. These messages are placed in the **rdoErrors** collection. In cases where several messages arrive at once, only a single InfoMessage event is fired after the last message arrives and has been added to the **rdoErrors** collection.

Setting Default rdoEnvironment Properties

The following properties establish default settings for all newly-created **rdoEnvironment** objects. They are also used when instantiating stand-alone **rdoConnection** objects.

- Use the **rdoDefaultLoginTimeout** property to determine the **rdoEnvironment** object's default **LoginTimeout** property used in connection timeout management.
- Use the **rdoDefaultCursorDriver** property to determine the **rdoEnvironment** object's default **CursorDriver** value. This property determines if the ODBC driver manager creates client batch, local, server-side, or no cursors.
- Use the **rdoDefaultUser** and **rdoDefaultPassword** properties to determine the default **rdoEnvironment** object's **UserName** and **Password** properties. These determine the user name and password when opening connections if no specific values are supplied.

Working with other rdoEngine Properties and Methods

You can establish the default configuration of new **rdoEnvironment** objects and create new ODBC data source entries using the properties and methods of the **rdoEngine** object. For example, you can:

- Use the **rdoEnvironments** collection to examine **rdoEnvironment** objects that have been appended to the collection. Note that **rdoEnvironment** objects can be allocated as stand-alone objects.
- Use the **rdoLocaleID** property to determine which language-localized DLLs are loaded.
- Use the **Version** property to examine the version of RDO in use.
- Use the **rdoErrors** collection to examine information about errors generated by the ODBC interface. Errors generated by Visual Basic are maintained in a separate **Errors** collection.
- Use the **rdoRegisterDataSource** method to create a new data source entry in the Windows System Registry.
- Use the **rdoCreateEnvironment** method to create a new **rdoEnvironment** object. You can also allocate a new **rdoEnvironment** object by coding

```
Dim MyEnv as New rdoEnvironment
```

Visual Basic: RDO Data Control

rdoEngine Object Example

This example sets a number of **rdoEngine** properties and creates a customized **rdoEnvironment** object based on these new default settings. Note that while your code can *set* a password in an **rdoEnvironment** object, it cannot be read once it is set.

```
Dim en As rdoEnvironment
Private Sub Form_Load()
With rdoEngine
    .rdoDefaultLoginTimeout = 20
    .rdoDefaultCursorDriver = rdUseOdbc
    .rdoDefaultUser = "Fred"
    .rdoDefaultPassword = ""
End With
Set en = rdoEnvironments(0)
'
'   Dump current rdoEnvironments collection
'   and display current properties where
'   possible.
'
For Each en In rdoEnvironments
    Debug.Print "LoginTimeout:" & en.LoginTimeout
    Debug.Print "CursorDriver:" & en.CursorDriver
    Debug.Print "User:" & en.UserName
   ' (Write-only) Debug.Print "Password:" & en.Password
Next
End Sub
```

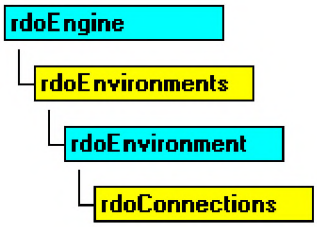
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoEnvironment Object

See Also Example Properties Methods Events

An **rdoEnvironment** object defines a logical set of connections and transaction scope for a particular user name. It contains both open and allocated but unopened connections, provides mechanisms for simultaneous transactions, and provides a security context for data manipulation language (DML) operations on the database.



Remarks

Generally, an **rdoEnvironment** object corresponds to an ODBC environment that can be referred to by the **rdoEnvironment** object's **hEnv** property. However, if the *Name* argument is *not* provided when the **rdoEnvironment** object is created by the **rdoCreateEnvironment** method, a stand-alone **rdoEnvironment** is created that is not added to the **rdoEnvironments** collection. Stand-alone **rdoEnvironment** objects are not exposed to other in-process DLLs unless specifically designated as public. If the reference count for any private **rdoEnvironment** is reduced to zero, all **rdoConnections** associated with the **rdoEnvironment** are closed.

Once you set the properties of an **rdoEnvironment** object, you can use the **Add** method to append it to the **rdoEnvironments** collection or the **Remove** method to detach and deallocate the object. The **Name** property is read-only and is determined by the specific remote data object.

The default **rdoEnvironment** is created automatically when the **RemoteData** control is initialized, or the first remote data object is referenced in code. The **Name** property of **rdoEnvironments**(0) is "Default_Environment". The user name and password for **rdoEnvironments**(0) are both "".

rdoEnvironment objects can be created with the **rdoCreateEnvironment** method of the **rdoEngine** object which automatically appends the new object to the **rdoEnvironments** collection. All **rdoEnvironment** objects created in this manner are assigned properties based on the default properties set in the **rdoEngine** object.

The user name and password information from the **rdoEnvironment** is used to establish the connection if these values are not supplied in the ***connect*** argument of the **OpenConnection** method, or in the **Connect** property of the RemoteData control.

All **rdoEnvironment** objects share a common **hEnv** value that is created on an application basis. Use the **rdoEnvironment** object to manage the current ODBC environment, or to start an additional connection. In an **rdoEnvironment**, you can open multiple connections, manage transactions, and establish security based on user names and passwords. For example, you can:

- Create an **rdoEnvironment** object using the **Name**, **Password**, and **UserName** properties to establish a named, password-protected environment. The environment creates a scope in which you can open multiple connections and

conduct one instance of coordinated transactions.

- Use the **CursorDriver** property to determine which cursor driver library is used to build **rdoResultset** objects. You can choose one of four types of cursors, or set the **CursorDriver** property to **rdUseNone** to indicate that no cursor is to be used to manage result sets.
- Use the **OpenConnection** method to open one or more existing connections in that **rdoEnvironment**.
- Use the **LoginTimeout** property to determine how long the ODBC drivers should wait before abandoning the connection attempt.
- Use the **BeginTrans**, **CommitTrans**, and **RollbackTrans** methods to manage transaction processing within an **rdoEnvironment** across several connections.
- Use several **rdoEnvironment** objects to conduct multiple, simultaneous, independent, and overlapping transactions.
- Use the **Close** method to terminate an environment and the connection and remove the **rdoEnvironment** object from the **rdoEnvironments** collection. This also closes all connections associated with the object.

Managing Transactions

The **rdoEnvironment** also determines transaction scope. Committing an **rdoEnvironment** transaction commits all open **rdoConnection** databases and their corresponding open **rdoResultset** objects. This does not imply a two-phase commit operation simply that individual **rdoConnection** objects are instructed to commit any pending operations one at a time.

For Microsoft SQL Server databases, the Distributed Transaction Coordinator (DTC) can be used to manage blocks of transactions simply by introducing the SQL query with the BEGIN DISTRIBUTED TRANSACTION statement. DTC facilitates the creation of network-wide database updates through its own two-phase commit protocol. Whenever SQL Server commits a transaction, the DTC ensures all related resources also commit the transaction. If any part of the transaction fails, the DTC ensures that the entire transaction is rolled back across all enlisted servers.

When you use transactions, all databases in the specified **rdoEnvironment** are affected even if multiple **rdoConnection** objects are opened in the **rdoEnvironment**. For example, suppose you use a **BeginTrans** method against one of the databases visible from the connection, update several rows in the database, and then delete rows in another **rdoConnection** object's database. When you use the **RollbackTrans** method, both the update and delete operations are rolled back. To avoid this problem, you can create additional **rdoEnvironment** objects to manage transactions independently across **rdoConnection** objects. Note that transactions executed by multiple **rdoEnvironment** objects are serialized and are not atomic operations. Because of this, their success or failure is not interdependent. This is an example of batched transactions.

You can execute nested transactions *only* if your data source supports them. For example, on a single connection, you can execute a BEGIN TRANS SQL statement, execute several UPDATE queries, and another BEGIN TRANS statement. Any operations executed after the second BEGIN TRANS SQL statement can be rolled back independently of the statements executed after the first BEGIN TRANS. This is an example of nested transactions. To commit the first set of UPDATE statements, you must execute a COMMIT TRANS statement, or a ROLLBACK TRANS statement for each BEGIN TRANS executed.

rdoEnvironment Events

The following events are fired as the **rdoEnvironment** object is manipulated. These can be used to micro-manage RDO transactions associated with the **rdoEnvironment** or to synchronize some other process with the transaction.

Event Name	Description
BeginTrans	Fired after the BeginTrans method has completed.
CommitTrans	Fired after the CommitTrans method has completed.
RollbackTrans	Fired after the RollbackTrans method has completed.

Addressing rdoEnvironment Objects

The **Name** property of **rdoEnvironment** objects is set from the *name* argument passed to the **rdoCreateEnvironment** method. You can refer to any other **rdoEnvironment** object by specifying its **Name** property setting using this syntax:

rdoEnvironments("MyEnvName")

or simply:

rdoEnvironments!*MyEnvName*

You can also refer to **rdoEnvironment** objects by their position in the **rdoEnvironments** collection using this syntax (where *n* is the *n*th member of the zero-based **rdoEnvironments** collection):

rdoEngine.rdoEnvironments(*n*)

or simply:

rdoEnvironments(*n*)

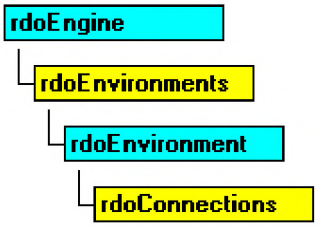
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoEnvironments Collection

See Also Example Properties Methods Events

The **rdoEnvironments** collection contains all active **rdoEnvironment** objects of the **rdoEngine** object.



Remarks

rdoEnvironment objects are created with the **rdoCreateEnvironment** method of the **rdoEngine** object. Newly created **rdoEnvironment** objects are automatically appended to the **rdoEnvironments** collection unless you do not provide a name for the new object when using the **rdoCreateEnvironment** method or simply declare a new **rdoEnvironment** object in code.

The **rdoEnvironments** collection is automatically initialized with a default **rdoEnvironment** object based on the default properties set in the **rdoEngine** object.

If you use the **Close** method against an **rdoEnvironment** object, all **rdoConnections** it contains are closed and the object is removed from the **rdoEnvironments** collection.

Visual Basic: RDO Data Control

rdoEnvironment Object, rdoEnvironments Collection Example

The following example illustrates creation of the **rdoEnvironment** object and its subsequent use to open an **rdoConnection** object.

```
Private Sub rdoEnvironmentButton_Click()
Dim en As rdoEnvironment
Dim cn As rdoConnection
Set en = rdoEngine.rdoEvironments(0)
With en
    en.CursorDriver = rdUseOdbc
    en.LoginTimeout = 5
    en.Name = "TransOp1"
    Set cn = en.OpenConnection(dsname:="", _
        prompt:=rdDriverNoPrompt, _
        Connect:="UID=;PWD=;" _
        driver={SQL Server};Server=SEQUEL;", _
        Options:=rdAsyncEnable)
End With
Print "Connecting ";
While cn.StillConnecting
    Print ".";
    DoEvents
Wend
Print "done."

End Sub
```

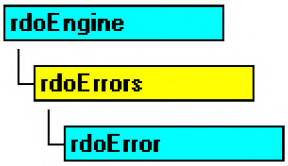
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoError Object

See Also Example Properties Methods Events

Contains details about remote data access errors.



Remarks

Any operation involving remote data objects can potentially generate one or more ODBC errors or informational messages. As each error occurs or as messages are generated, one or more **rdoError** objects are placed in the **rdoErrors** collection of the **rdoEngine** object. When a subsequent RDO operation generates an error, the **rdoErrors** collection is cleared, and the new set of **rdoError** objects is placed in the **rdoErrors** collection. RDO operations that don't generate an error have no effect on the **rdoErrors** collection. To make error handling easier. you can use the **Clear** method to purge the **rdoErrors** collection between operations.

Generally, all ODBC errors generate a trappable Visual Basic error of some kind. This is your cue to check the contents of the **rdoErrors** collection for any and all errors resulting from the last operation which provide specific details on the cause of the error.

Not all errors generated by ODBC are fatal. In the normal course of working with connections, default databases, stored procedure print statements and other operations, the remote server often returns warnings or messages that are usually safe to ignore. When an informational message arrives, the **rdoEngine** InfoMessage event is fired. You should examine the **rdoErrors** collection in this event procedure.

If the severity of the error number is below the error threshold as specified in either the **rdoDefaultErrorThreshold** or **ErrorThreshold** property, then a trappable error is triggered when the error is detected. Otherwise, an **rdoError** object is simply appended to the **rdoErrors** collection. To control trappable errors in Microsoft SQL Server, you should use the Transact SQL RAISERROR statement coupled with an appropriate *Severity* argument to indicate the error or other information.

Use the **rdoError** object to determine the type and severity of any errors generated by the RemoteData control or RDO operations. For example, you can:

- Use the **Description** property to display a text message describing the error.
- Use the **Number** property to determine the native data source error number.
- Use the **Source** property to determine the source of the error and the object class causing the error.
- Use the **SQLRetCode** and **SQLState** properties to determine the ODBC return code and **SQLState** flags.

- Use the **Clear** method on the **rdoErrors** collection to remove all **rdoError** objects. In most cases, it is not necessary to use the **Clear** method because the **rdoErrors** collection is cleared automatically when a new error occurs.

Members of the **rdoErrors** collection aren't appended as is typical with other collections. The most general errors are placed at the end of the collection (**Count** -1), and the most detailed errors are placed at index 0. Because of this implementation, you can often determine the root cause of the failure by examining **rdoErrors**(0).

The set of **rdoError** objects in the **rdoErrors** collection describes one error. The first **rdoError** object is the lowest level error, the second is the next higher level, and so forth. For example, if an ODBC error occurs while the **RemoteData** control tries to create an **rdoResultset** object, the last **rdoError** object contains the RDO error indicating the object couldn't be opened. The first error object contains the lowest level ODBC error. Subsequent errors contain the ODBC errors returned by the various layers of ODBC. In this case, the driver manager, and possibly the driver itself, returns separate errors which generate **rdoError** objects.

The **rdoErrors** collection is also used to manage informational messages returned by the data source. For example, messages returned back from PRINT statements, showplan requests, or DBCC operations in SQL Server are returned as **rdoError** objects in the **rdoErrors** collection. This type of message causes the InfoMessage event to fire, but does not trip a trappable error. Because of this, you must check the **rdoErrors** collection's **Count** property to see if any new errors have arrived.

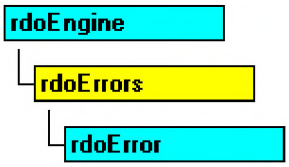
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoErrors Collection

See Also Example Properties Methods Events

Contains all stored **rdoError** objects which pertain to a single operation involving Remote Data Objects (RDO).



Remarks

Any operation involving remote data objects can generate one or more errors. As each error occurs, one or more **rdoError** objects are placed in the **rdoErrors** collection of the **rdoEngine** object. When another RDO operation generates an error, the **rdoErrors** collection is cleared, and the new set of **rdoError** objects is placed in the **rdoErrors** collection. RDO operations that don't generate an error have no effect on the **rdoErrors** collection.

- Use the **Clear** method on the **rdoErrors** collection to remove all **rdoError** objects. In most cases, it is not necessary to use the **Clear** method because the **rdoErrors** collection is cleared automatically when a new error occurs.

Members of the **rdoErrors** collection aren't appended as is typical with other collections. The most general errors are placed at the end of the collection (**Count** -1), and the most detailed errors are placed at index 0. Because of this implementation, you can determine the root cause of the failure by examining **rdoErrors**(0).

The set of **rdoError** objects in the **rdoErrors** collection describes one error. The first **rdoError** object is the lowest level error, the second is the next higher level, and so forth. For example, if an ODBC error occurs while the **RemoteData** control tries to create an **rdoResultset** object, the last **rdoError** object contains the RDO error indicating the object couldn't be opened. The first error object contains the lowest level ODBC error. Subsequent errors contain the ODBC errors returned by the various layers of ODBC. In this case, the driver manager, and possibly the driver itself, returns separate errors which generate **rdoError** objects.

Visual Basic: RDO Data Control

rdoError Object, rdoErrors Collection Example

The following code illustrates a simple design-time RDO error handler. Note that the handler simply displays the errors in the **rdoErrors** collection in the Immediate window.

```
Dim er as rdoError
On Error GoTo CnEh
.
.
.

CnEh:
Dim er As rdoError
    Debug.Print Err, Error
    For Each er In rdoErrors
        Debug.Print er.Description, er.Number
    Next er
    Resume Next
```

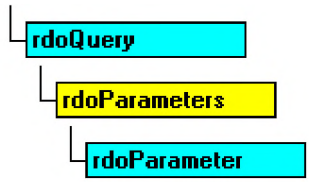
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoParameter Object

See Also Example Properties Methods Events

An **rdoParameter** object represents a parameter associated with an **rdoQuery** object.



Remarks

When working with stored procedures or SQL queries that require use of arguments that change from execution to execution, you should create an **rdoQuery** object to manage the query and its parameters. For example, if you submit a query that includes information provided by the user such as a date range, or part number, RDO and the ODBC interface can insert these values automatically into the SQL statement at specific positions in the query.

Providing Parameters

Your query's parameters can be provided in a number of ways:

- As hard-coded arguments in the SQL query string.

```
"Select Name from Animals Where ID = 'Cat'"
```

- As concatenated text or numeric values extracted from **TextBox**, **Label** or other controls.

```
"Select Name from Animals Where ID = '"  _
        & IDWanted.Text & "'"
```

- As the question mark (?) parameter placeholders.

```
"Select Name from Animals Where ID = ?"
```

- As the question mark (?) parameter placeholders in a stored procedure call that accepts input, output and/or return status arguments.

```
"{ ? = Call MySP (?, ?, ?) }"
```

Note Stored procedure invocations that use the Call syntax (as shown above) are executed in their "native" format so they do not require parsing and data conversion by the ODBC Driver Manager. Because of this the Call syntax can be executed somewhat faster than other syntaxes.

Using Parameter Markers

The only time you *must* use parameter markers is when executing stored procedures that require input, output or return status arguments. If the stored procedure only requires input arguments, these can be provided in-line as imbedded values

concatenated into the query (as shown below).

When the **rdoParameter** collection is first referenced (but not before) RDO and the ODBC interface pre-processes the query, and creates an **rdoParameter** object for each *marked* parameter. You can also create queries with multiple parameters, and in this case you can mark some parameters and provide the others by hard-coding or concatenation in any combination. However, all marked parameters must appear to the left of all other parameters. If you don't, a trappable error occurs indicating "Wrong number of parameters".

Note Due to the extra overhead involved in creating and managing **rdoQuery** objects and their **rdoParameters** collection, you should not use parameter queries for SQL statements that do not change from execution to execution especially those that are executed only once or infrequently.

Marking Parameters

Each query parameter that you want to have RDO manage must be indicated by a question mark (?) in the text of the SQL statement, and correspond to an **rdoParameter** object referenced by its ordinal number counting from zero left to right. For example, to execute a query that takes a single input parameter, your SQL statement would look something like this:

```
SQL$ = "Select Au_Lname, Au_Fname where Au_ID Like ? "
Dim qd as rdoQuery, rd as rdoResultset
Set qd = CreateQuery ("SeekAUID", SQL$)
qd(0) = "236-66-%"
set rd = qd.OpenResultset(rdOpenForwardOnly)
```

Note You can also create an **rdoQuery** object using the Query Connection designer and name and set the data type and direction of individual parameters.

Acceptable Parameters

Not all types of data are passable as parameters. For example you cannot always use a TEXT or IMAGE data type as an OUTPUT parameter. In addition, if your query does not require parameters or has no parameters in a specific invocation of the query, you cannot use parenthesis in the query. For example, for a stored procedure that does not require parameters could be coded as follows:

```
"{ ? = Call MySP }"
```

When submitting queries that return output parameters, these parameters must be submitted at the end of the list of your query's parameters. While it is possible to provide both marked and unmarked (in-line) parameters, your output parameters must still appear at the end of the list of parameters.

All in-line parameters must be provided to the right of marked parameters. If this is not the case, RDO returns an error indicating "Wrong number of parameters".

RDO 2.0 supports BLOB data types as parameters and you also can use the **AppendChunk** method against the **rdoParameter** object to pass TEXT or IMAGE data types as parameters into a procedure.

Identifying the Parameter's Data Type

When your parameter query is processed by ODBC, it attempts to identify the data type of each parameter by executing ODBC functions that query the remote server for specific information about the query. In some cases, the data type cannot be correctly determined. In these cases, use the **Type** property to set the correct data type or create a custom query using the User Connection Designer.

For example, in the following query, the parameter passed to the TSQL **Charindex** function is typed as an integer. While this is correct for the function itself, the parameter is referencing a string argument of the TSQL function, so it must be set to an ODBC character type to work properly.

```
Dim SQL as string, qd as rdoQuery
SQL = "Select * From Titles " _
```

```
    & "Where Charindex( ?,  Title) > 0
Set qd = cn.CreateQuery("FindTitle", SQL)
qd(0).Type = rdTypeChar
```

Note You do not have to surround text parameters with quotes as this is handled automatically by the ODBC API interface.

Handling Output and Return Status Arguments

In some cases, a stored procedure returns an output or return status argument instead of or in addition to any rows returned by a SELECT statement. Each of these parameters must also be marked in the SQL statement with a question mark. Using this technique, you can mark the position of any number of parameters in your SQL query including input, output or input/output.

Whenever your query returns output or return status arguments, you *must* use the ODBC CALL syntax when setting the SQL property of the **rdoQuery** object. In this case, a typical stored procedure call would look like this:

```
Dim qd as rdoQuery, rd as rdoResultset, SQL as String
SQL = "{ ? = Call master..sp_password (?, ?) }"
Set qd = CreateQuery ("SetPassword", SQL)
qd.rdoParameters(0).Direction = rdParamReturnValue
qd(1) = "Fred"      ' the old password
qd(2) = "George"    ' the new password
set rd = qd.Execute
if qd(0) <> 0 then _
   MsgBox "Operation failed"
```

Tip Be sure to specifically address stored procedures that do not reside in the current (default) database. In this example, the default database is *not* Master where the sp_password procedure is maintained, so this procedure is specifically addressed.

When control returns to your application after the procedure is executed, the **rdoParameter** objects designated as **rdParamReturnValue**, **rdParamOutput** or **rdParamInputOutput** contain the returned argument values. In the example shown above, the return status is available by examining `qd(0)` after the query is executed.

Using Other Properties

Using the properties of an **rdoParameter** object, you can set a query parameter that can be changed before the query is run. You can:

- Use the **Direction** property setting to determine if the parameter is an input, output, or input/output parameter, or a return value. In RDO 2.0, the **Direction** property is usually set automatically, so it is unnecessary to set this value. It is also unnecessary to set it for input parameters which is the default value.
- Use the **Type** property setting to determine the data type of the **rdoParameter**. Data types are identical to those specified by the **rdoColumn.Type** property. In some cases, RDO might not be able to determine the correct parameter data type. In these cases, you can force a specific data type by setting the **Type** property.
- Use the **Value** property (the default property of an **rdoParameter**) to pass values to the SQL queries containing parameter markers used in **rdoQuery.Execute** or **rdoQuery.OpenResultset** methods. For example:

  ```
  MyQuery(0) = 5
  ```

Note RDO requires that your ODBC driver support a number of Level II compliant options and support the **SQLNumParams, SQLProcedureColumns** and **SQLDescribeParam** ODBC API functions in order to be able to create the **rdoParameters** collection and parse parameter markers in SQL statements. While some drivers can be used to create and execute queries, if your driver does not support creation of the **rdoParameters** collection, RDO fails quietly and simply does not create the collection. As a result, any reference to the collection results in a trappable error.

Addressing the Parameters

By default, members of the **rdoParameters** collection are named "Parameter*n*" where *n* is the **rdoParameter** object's ordinal number. For example, if an **rdoParameters** collection has two members, they are named "Parameter0" and "Parameter1". However, if you use the User Connection Designer, you can specify names for specific parameters.

Because the **rdoParameters** collection is the default collection for the **rdoQuery** object, addressing parameters is easy. Assuming you have created an **rdoQuery** object referenced by `rdoQo`, you can refer to the **Value** property of its **rdoParameter** objects by:

- Referencing the **Name** property setting using this syntax:

```
' Refers to PubDate parameter
rdoQo("PubDate")
```

-Or-

```
' Refers to PubDate parameter
rdoQo!PubDate
```

- Referencing its ordinal position in the **rdoParameters** collection using this syntax:

```
' Refers to the first parameter marker
rdoQo(0)
```

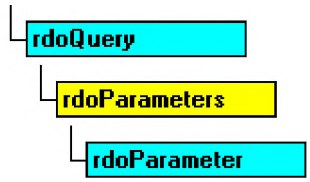
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoParameters Collection

See Also Example Properties Methods Events

An **rdoParameters** collection contains all the **rdoParameter** objects of an **rdoQuery** object.



Remarks

The **rdoParameters** collection provides information only about *marked* parameters in an **rdoQuery** object or stored procedure. You can't append objects to or delete objects from the **rdoParameters** collection.

When the **rdoParameters** collection is first referenced, RDO and the ODBC interface parse the query searching for parameter markers the question mark (?). For each marker found, RDO creates an **rdoParameter** object and places it in the **rdoParameters** collection. However, if the query cannot be compiled or otherwise processed, the **rdoParameters** collection is *not* created and your code will trigger a trappable error indicating that the object does not exist. In this case, check the query for improper syntax, permissions on underlying objects, and proper placement of parameter markers.

Visual Basic: RDO Data Control

rdoParameter Object, rdoParameters Collection, Direction Property Example

This example executes a stored procedure against the SQL Server 'Pubs database. The procedure text is also included here so you can setup this example on your own machine. The stored procedure expects your code to provide three input arguments: A string to use in an expression to choose the title, and two numbers used to choose a price range for the books. The procedure returns the number of books that fall in the range, and the maximum price of the books. It also returns a set of rows containing detailed information about the books.

To establish the connection, we assume the name of the server is "SEQUEL" and it is a Microsoft SQL Server this is a DSN-less connection. Next, we use the ODBC CALL syntax to prepare the query. Notice that each parameter is marked with a question mark. Once, marked, the **rdoParameters** collection is used to set the direction for the output and return value parameters and the initial values for the input parameters. While you don't see the **rdoParameters** collection called out specifically, understand that it is the default collection of the **rdoQuery** object so references are made simpler by *not* including a reference to the **rdoParameters** collection itself.

```
Sub RunQuery_Click()
Dim rs As rdoResultset
Dim cn As New rdoConnection
Dim qd As New rdoQuery
Dim cl As rdoColumn
Const None As String = ""

cn.Connect = "uid=;pwd=;server=SEQUEL;" _
    & "driver={SQL Server};database=pubs;" _
    & "DSN='';"
cn.CursorDriver = rdUseOdbc
cn.EstablishConnection rdDriverNoPrompt

Set qd.ActiveConnection = cn
qd.SQL = "{ ? = Call ShowOutputRS (?,?,?,?,?) }"
qd(0).Direction = rdParamReturnValue
qd(4).Direction = rdParamOutput
qd(5).Direction = rdParamOutput
qd(1) = "c"
qd(2) = 5
qd(3) = 50

Set rs = qd.OpenResultset(rdOpenForwardOnly, _
    rdConcurReadOnly)

For Each cl In rs.rdoColumns
    Debug.Print cl.Name,
Next
Debug.Print

Do Until rs.EOF
    For Each cl In rs.rdoColumns
        Debug.Print cl.Value,
    Next
    rs.MoveNext
```

```
Debug.Print
Loop

Debug.Print "Output from SP="; qd(3)
Debug.Print "Return Status from SP="; qd(0)

rs.Close
qd.Close
cn.Close

End Sub
```

This is the stored procedure that is executed by the example shown above.

```
CREATE PROCEDURE ShowOutputRS
(
   @Ser varChar(128),
   @PriceLow Integer,
   @PriceHigh Integer,
   @Hits Integer OUTPUT,
   @MaxPrice integer OUTPUT
)
AS
Select @MaxPrice = Max(Price) from Titles
where Charindex(@Ser, title) > 0
and price between @priceLow and @priceHigh

Select * from Titles
where Charindex(@Ser, title) > 0
and price between @priceLow and @PriceHigh

Select @Hits = @@RowCount

return @@ROWCOUNT
```

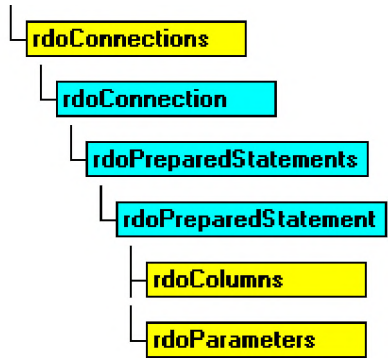
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoPreparedStatement Object

See Also Example Properties Methods Events

An **rdoPreparedStatement** object is a prepared query definition.



Remarks

Note The **rdoPreparedStatement** object is outdated and only maintained for backward compatibility. It should be replaced with the **rdoQuery** object. The **rdoQuery** object supports all of the **rdoPreparedStatement** object's properties and methods. In contrast, the **rdoPreparedStatement** only a subset of the **rdoQuery** object's properties and methods and none of its events.

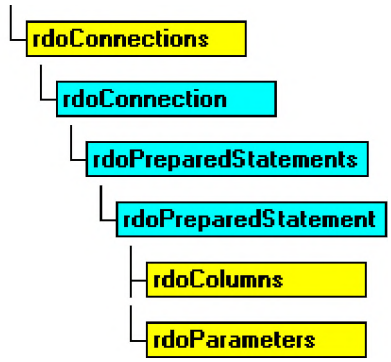
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoPreparedStatements Collection

See Also Example Properties Methods Events

An **rdoPreparedStatements** collection contains all **rdoPreparedStatement** objects in an **rdoConnection**.



Remarks

Note The **rdoPreparedStatements** collection is outdated and maintained for compatibility. It should be replaced with the **rdoQueries** collection. The **rdoQuery** object and **rdoQueries** collection supports all of the **rdoPreparedStatement** object's properties and methods. In contrast, the **rdoPreparedStatement** supports only a subset of the **rdoQuery** object's properties and methods and none of its events.

Note RDO requires that your ODBC driver support a number of Level II options and support the **SQLNumParams, SQLProcedureColumns** and **SQLDescribeParam** ODBC API functions in order to be able to create the **rdoParameters** collection and parse SQL statement parameter markers. While some drivers can be used to create and execute queries, if your driver does not support creation of the **rdoParameters** collection, RDO fails quietly and simply does not create the collection.

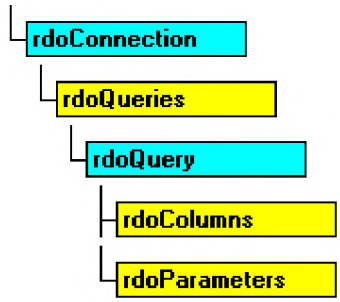
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoQueries Collection

See Also Example Properties Methods Events

Contains **rdoQuery** objects that have been added to the **rdoQueries** collection either automatically via the **CreateQuery** method, or with the **Add** method.



Remarks

An **rdoQuery** object is automatically appended to the **rdoQueries** collection when you use the **CreateQuery** method of the **rdoConnection** object. You can also use the **Add** method against the rdoQueries collection supplying a stand-alone **rdoQuery** object as the argument.

When you use the **Close** method against and **rdoQuery** object, it is removed from the **rdoQueries** collection, but the object remains instantiated. By resetting the **ActiveConnection** property, you can associate the **rdoQuery** object with another connection and use the **Add** method to append it to the **rdoQueries** collection.

An **rdoQuery** object need not be a member of the **rdoQueries** collection before it can be associated with an **rdoConnection** object and used with the **Execute** or **OpenResultset** methods.

Visual Basic: RDO Data Control

rdoQuery Object, rdoQueries Collection Example

This example leverages RDO's ability to set the data type of individual arguments of a query. In this case, a CHARINDEX function argument is passed as a parameter. Since the ODBC driver does not recognize this data type correctly, we simply change it to CHAR before assigning a value to the parameter. The query itself uses TSQL syntax it does not need to use the ODBC CALL syntax as it does not execute a parameter-based stored procedure. This example also creates a DSN-less connection to a Microsoft SQL Server and uses the sample Pubs database.

```
Private Sub Query1_Click()
Dim rs As rdoResultset
Dim cn As New rdoConnection
Dim qd As New rdoQuery
Dim cl As rdoColumn
Const None As String = ""

cn.Connect = "uid=;pwd=;server=SEQUEL;" _
    & "driver={SQL Server};database=pubs;" _
    & "DSN='';"
cn.CursorDriver = rdUseOdbc
cn.EstablishConnection rdDriverNoPrompt

Set qd.ActiveConnection = cn
qd.SQL = "Select * From Titles" _
    & " Where CharIndex( ?, Title) > 0"

qd(0).Type = rdTypeCHAR
qd(0) = InputBox("Enter search string", , "C")

Set rs = qd.OpenResultset(rdOpenForwardOnly, rdConcurReadOnly)

For Each cl In rs.rdoColumns
    Debug.Print cl.Name,
Next
Debug.Print

Do Until rs.EOF
    For Each cl In rs.rdoColumns
        Debug.Print cl.Value,
    Next
    rs.MoveNext
Debug.Print
Loop
End Sub
```

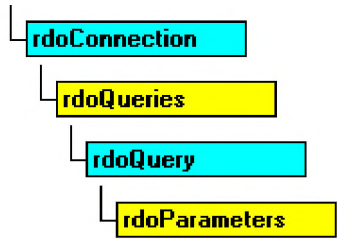
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoQuery Object

See Also Example Properties Methods Events

An **rdoQuery** object is a query definition that can include zero or more parameters.



Remarks

The **rdoQuery** object is used to manage SQL queries requiring the use of input, output or input/output parameters. Basically, an **rdoQuery** functions as a compiled SQL statement. When working with stored procedures or queries that require use of arguments that change from execution to execution, you can create an **rdoQuery** object to manage the query parameters. If your stored procedure returns output parameters or a return value, or you wish to use **rdoParameter** objects to handle the parameters, you *must* use an **rdoQuery** object to manage it. For example, if you submit a query that includes information provided by the user such as a date range or part number, RDO can substitute these values automatically into the SQL statement when the query is executed.

Note The **rdoQuery** object replaces the outdated **rdoPreparedStatement** object. The **rdoQuery** object remains similar to the **rdoPreparedStatement** in its interface, but adds the ability to be persisted into a Visual Basic project, allowing you to create and manipulate it at design time. Additionally, the **rdoQuery** objects can be prepared or not, allowing the you to choose the most appropriate use of the query.

Creating rdoQuery Objects

To create an **rdoQuery** object, use the **CreateQuery** method which associates the **rdoQuery** with a specific **rdoConnection** object and adds it to the **rdoQueries** collection. Once created, you must fill in required parameters using the **rdoParameters** collection, and then use the **OpenResultset** method to create resultsets from the query, or the **Execute** method to simply run the query if it does not return rows.

You can also use the User Connection Designer (CQD) to create **rdoQuery** objects in your project. The CQD takes your SQL query and permits you to specify the data types for each parameter. It then inserts appropriate code in your application to expose these procedures very much like methods off of the **rdoQuery** object.

Note Due to the extra overhead involved in creating and managing **rdoQuery** objects and the **rdoParameters** collection, you should not use parameter queries for SQL statements that do not change from execution to execution especially those that are executed only once or infrequently.

Stand Alone rdoQuery Objects

You can declare a stand-alone **rdoQuery** object using the **Dim** statement as follows:

```
Dim MyQuery as New rdoQuery
```

Stand-alone **rdoQuery** objects are not assigned to a specific **rdoConnection** object, so you must set the **ActiveConnection** property before attempting to execute the query, or to use the **OpenResultset** object against it. The **CursorType** and **ErrorThreshold** properties are set from default values established by the **rdoEngine** default settings. In addition, new **rdoQuery** objects are not automatically appended to the **rdoQueries** collection until you use the **Add** method.

For example, the code shown below creates an **rdoQuery** object, associates it with a connection, and executes it. Next, the **rdoQuery** object is associated with a different connection and executed again. The query object becomes more of an encapsulation of any kind of query, and thus can be executed against any kind of connection, provided the SQL statement would be appropriate for the connection.

```
Dim MyQuery As rdoQuery '
MyQuery.SQL = "Update customers " _
   & " Set LastTouched = GetDate()"
MyQuery.Prepared = False    'don't prepare it,
                           'just SQLExecDirect
'assume that cnSomeConnection
'is an rdoConnection or stand-alone object
MyQuery.ActiveConnection = cnSomeConnection
MyQuery.Execute

MyQuery.ActiveConnection = cnOtherConnection
'the cnOtherConnection is over a WAN, so I can increase
'my query timeout to compensate
MYQuery.QueryTimeout = 120
MyQuery.Execute
```

Choosing the right SQL Syntax

When coding the SQL property of an **rdoQuery** object, you can choose between one of three syntax styles to code your parameter query:

- Concatenated Strings: Your code builds up the SQL statement and its parameters using the Visual Basic concatenation (&) operator. This statement can be passed to the ***SQL*** argument of the **OpenResultset** method or the **rdoQuery** object's **SQL** property. In this case, a parameter query might look like this:

```
sSQL = "Select Name, Age From Animals " _
& " Where Weight > " & WeightWanted.Text _
& " and Type = ' & TypeWanted.Text & "'"
```

- Native SQL syntax: The SQL syntax used by the remote server. In this case you can execute your own query or stored procedure, and pass in parameters by concatenation, or using placeholders, or both. The parameters marked with placeholders are managed by RDO as **rdoParameter** objects. A parameter query might look like this:

  ```
  sSQL = "Select Au_LName from Authors" _
      & " Where Au_Fname = ?"
  ```

  Or

  ```
  sSQL = "Execute MyStoredProc 'Arg1', 450, '"  _
      & Text1
  ```

  Or

  ```
  sSQL = "Execute MyStoredProc ?, ?, ?"
  ```

- ODBC CALL syntax: Designed to call stored procedures that return a return status or output parameters. In this case, a placeholder can be defined for each input, output, or input/output parameter which is automatically mapped to

rdoParameter objects. You can also mix in concatenated operators as needed. In this case, a parameter query might look like this:

```
sSQL = "{call ParameterTest (?,?,?) }"
```

Or

```
sSQL = "{? = call ParameterTest (?,?,?) }"
```

Or

```
sSQL = "{? = call CountAnimals (?, ?, 14, 'Pig')}
```

The **rdoQuery** object is managed by setting the following properties and methods.

- Use the **SQL** property to specify a parameterized SQL statement to execute. The ***name*** argument of the **CreateQuery** method can also be used to provide the SQL query string.
- Set query parameters by using the **rdoQuery** object's **rdoParameters** collection.
- Use the **Prepared** property to indicate if the **rdoQuery** object should be prepared by the ODBC **SQLPrepare** function. If **False**, the query is executed using the **SQLExecDirect** function.
- Use the **Type** property to determine whether the query selects rows from an existing table (select query), performs an action (an action query), contains both action and select operations, or represents a stored procedure.
- Use the **RowsetSize** property setting to determine how many rows are buffered internally when building a cursor and locked when using pessimistic locking.
- Use the **KeysetSize** property to indicate the size of the keyset buffer when creating cursors.
- Use the **MaxRows** property to indicate the maximum number of rows to be returned by a query.
- Use the **RowsAffected** property to indicate how many rows are affected by an action query.
- Use the **QueryTimeout** property to indicate how long the driver manager waits before pausing a query and firing the QueryTimeout event.
- Use the **BindThreshold** property to indicate the largest column to be automatically bound.
- Use the **ErrorThreshold** property to indicate the error level that constitutes a trappable error.
- Use the **Updatable** property to see if the result set generated by an **rdoQuery** can be updated.
- Use the **OpenResultset** method to create an **rdoResultset** based on the **OpenResultset** arguments and properties of the **rdoQuery**.
- Use the **Execute** method to run an action query using **SQL** and other **rdoQuery** properties, including any values specified in the **rdoParameters** collection.
- Use the **LogMessages** property to activate ODBC tracing.

rdoQuery Object Events

The following events are fired as the **rdoQuery** object is manipulated. These can be used to micro-manage queries associated with the **rdoQuery** or coordinate other processes in your application.

Event Name	Description

QueryComplete	Fired when a query has completed.
QueryTimeout	Fired when the **QueryTimeout** period has elapsed and the query has not begun to return rows.
WillExecute	Fired before the query is executed permitting last-minute changes to the SQL, or to prevent the query from executing.

Closing the rdoQuery Object

Use the **Close** method to close an **rdoQuery** object, set its **ActiveConnection** property to **Nothing**, and remove it from the **rdoQueries** collection. However, you can still re-associate the **rdoQuery** object with another **rdoConnection** object by setting its **ActiveConnection** property to another **rdoConnection** object. Using the **Execute** method or **OpenResultset** method against an **rdoQuery** object that has its **ActiveConnection** property set to **Nothing** or an invalid **rdoConnection** causes a trappable error.

Addressing rdoQuery Objects

rdoQuery objects are the preferred way to submit parameter queries to the external server. For example, you can create a parameterized Transact SQL query (as used on Microsoft SQL Server) and store it in an **rdoQuery** object.

You refer to an **rdoQuery** object by its **Name** property setting using the following syntax. Since the **rdoQuery** object's default collection is the **rdoParameters** collection, all unqualified references to the **rdoQuery** object refer to the **rdoParameters** collection. In these examples, assume we have created an **rdoQuery** object named `rdoQo` using the syntax `Dim rdoQo as rdoQueries`. The first two examples refer to the **rdoQuery** object named "MyQuery".

rdoQo("MyQuery")

Or

rdoQo!*MyQuery*

You can also refer to **rdoQuery** objects (and the **rdoPreparedStatements** collection) by their position in the **rdoQueries** collection using this syntax (where *n* is the *n*th member of the zero-based **rdoQueries** collection):

rdoQo(*n*)

Visual Basic: RDO Data Control

rdoQuery Object, rdoQueries Collection Example

This example leverages RDO's ability to set the data type of individual arguments of a query. In this case, a CHARINDEX function argument is passed as a parameter. Since the ODBC driver does not recognize this data type correctly, we simply change it to CHAR before assigning a value to the parameter. The query itself uses TSQL syntax it does not need to use the ODBC CALL syntax as it does not execute a parameter-based stored procedure. This example also creates a DSN-less connection to a Microsoft SQL Server and uses the sample Pubs database.

```
Private Sub Query1_Click()
Dim rs As rdoResultset
Dim cn As New rdoConnection
Dim qd As New rdoQuery
Dim cl As rdoColumn
Const None As String = ""

cn.Connect = "uid=;pwd=;server=SEQUEL;" _
    & "driver={SQL Server};database=pubs;" _
    & "DSN='';"
cn.CursorDriver = rdUseOdbc
cn.EstablishConnection rdDriverNoPrompt

Set qd.ActiveConnection = cn
qd.SQL = "Select * From Titles" _
    & " Where CharIndex( ?, Title) > 0"

qd(0).Type = rdTypeCHAR
qd(0) = InputBox("Enter search string", , "C")

Set rs = qd.OpenResultset(rdOpenForwardOnly, rdConcurReadOnly)

For Each cl In rs.rdoColumns
    Debug.Print cl.Name,
Next
Debug.Print

Do Until rs.EOF
    For Each cl In rs.rdoColumns
        Debug.Print cl.Value,
    Next
    rs.MoveNext
Debug.Print
Loop
End Sub
```

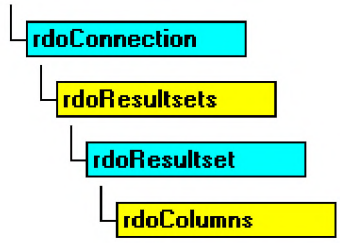
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoResultset Object

See Also Example Properties Methods Events

An **rdoResultset** object represents the rows that result from running a query



Remarks

When you use remote data objects, you interact with data almost entirely using **rdoResultset** objects. **rdoResultset** objects are created using the RemoteData control, or the **OpenResultset** method of the **rdoQuery**, **rdoTable,** or **rdoConnection** object.

When you execute a query that contains one or more SQL SELECT statements, the data source returns zero or more rows in an **rdoResultset** object. All **rdoResultset** objects are constructed using rows and columns.

A single **rdoResultset** can contain zero or any number of result sets so-called "multiple" result sets. Once you have completed processing the first result set in an **rdoResultset** object, use the **MoreResults** method to discard the current **rdoResultset** rows and activate the next **rdoResultset**. You can process individual rows of the new result set just as you processed the first **rdoResultset**. You can repeat this until the **MoreResults** method returns **False**.

A new **rdoResultset** is automatically added to the **rdoResultsets** collection when you open the object, and it's automatically removed when you close it.

Note RDO 1.0 collections behave differently than Data Access Object (DAO) collections. When you **Set** a variable containing a reference to a RDO object like **rdoResultset**, the existing **rdoResultset** is *not* closed and removed from the **rdoResultsets** collection. The existing object remains open and a member of its respective collection.

In contrast, RDO 2.0 collections do not behave in this manner. When you use the **Set** statement to assign a variable containing a reference to an RDO object, the existing object *is* closed and removed from the associated collection. This change is designed to make RDO more compatible with DAO.

Processing Multiple Result Sets

When you execute a query that contains more than one SELECT statement, you must use the **MoreResults** method to discard the current **rdoResultset** rows and activate each subsequent **rdoResultset**. Each of the **rdoResultset** rows *must* be processed or discarded before you can process subsequent result sets. To process result set rows, use the *Move* methods to position to individual rows, or the **MoveLast** method to position to the last row of the **rdoResultset**. You can use the **Cancel** or **Close** methods against **rdoResultset** objects that have not been fully processed.

Choosing a Cursor Type

You can choose the type of **rdoResultset** object you want to create using the *type* argument of the **OpenResultset** method the default **Type** is **rdOpenForwardOnly** for RDO and **rdOpenKeyset** for the **RemoteData** control. If you specify **rdUseNone** as the **CursorDriver** property, a forward-only, read-only result set is created. Each type of **rdoResultset** can contain columns from one or more tables in a database.

There are four types of **rdoResultset** objects based on the type of cursor that is created to access the data:

- Forward-only type **rdoResultset** individual rows in the result set can be accessed and can be updatable (when using server-side cursors), but the current row pointer can only be moved toward the end of the **rdoResultset** using the **MoveNext** method no other method is supported.
- Static-type **rdoResultset** a static copy of a set of rows that you can use to find data or generate reports. Static cursors might be updatable when using either the ODBC cursor library or server-side cursors, depending on which drivers are supported and whether the source data can be updated.
- Keyset-type **rdoResultset** the result of a query that can have updatable rows. Movement within the keyset is unrestricted. A keyset-type **rdoResultset** is a dynamic set of rows that you can use to add, change, or delete rows from an underlying database table or tables. Membership of a keyset **rdoResultset** is fixed.
- Dynamic-type **rdoResultset** the result of a query that can have updatable rows. A dynamic-type **rdoResultset** is a dynamic set of rows that you can use to add, change, or delete rows from an underlying database table or tables. Membership of a dynamic-type **rdoResultset** is not fixed.

Dissociate rdoResultset objects

When using the client batch cursor library, RDO permits you to disconnect an **rdoResultset** object from the **rdoConnection** object used to populate its rows by setting the **ActiveConnection** property to **Nothing**. While dissociated, the **rdoResultset** object becomes a temporary static snapshot of a local cursor. It can be updated, new rows can be added and rows can be removed from this **rdoResultset**. You can re-associate the **rdoResultset** by setting the **ActiveConnection** property to another (or the same) **rdoConnection** object. Once reconnected, you can use the **BatchUpdate** method to synchronize the **rdoResultset** with a remote database.

To perform this type of dissociated update operation, you should open the **rdoResultset** using an **rdOpenStatic** cursor, and use the **rdConcurBatch** as the concurrency option.

Managing rdoResultset Object Properties and Methods

You can use the methods and properties of the **rdoResultset** object to manipulate data and navigate the rows of a result set. For example, you can:

- Use the **Type** property to indicate the type of **rdoResultset** created, and the **Updatable** property indicates whether or not you can change the object's rows.
- Use the **BOF** and **EOF** properties to see if the current row pointer is positioned beyond either end of the **rdoResultset** or it contains no rows.
- Use the **MoveNext** method to reposition the current row in forward-only type **rdoResultset** objects.
- Use the **Bookmarkable, Transactions**, and **Restartable** properties to determine if the **rdoResultset** supports bookmarks or transactions, or can be restarted.
- Use the **LockEdits** property to determine the type of locking used to update the **rdoResultset**.
- Use the **RowCount** property to determine how many rows in the **rdoResultset** are available. If the **RowCount** property returns -1, RDO cannot determine how many rows have been processed. Only when you move to **EOF** does the **RowCount** property reflect the number of rows returned by the query. Not all cursor types support this functionality. The **RowCount** property returns -1 if it is not available.

- Use the **AddNew**, **Edit**, **Update**, and **Delete** methods to add new rows or otherwise modify updatable **rdoResultset** objects. Use the **CancelUpdate** method to cancel pending edits.

- Use the **Requery** method to restart the query used to create an **rdoResultset** object. This method can be used to re-execute a parameterized query.

- Use the **MoreResults** method to complete processing of the current **rdoResultset** and begin processing the next result set generated from a query. Use the **Cancel** method to terminate processing of all pending queries when the query contains more than one SQL operation. When you use the **Close** method against an **rdoResultset**, all pending queries are flushed and the **rdoResultset** is automatically dropped from the **rdoResultsets** collection.

- Use the **Close** method to terminate and deallocate the **rdoResultset** object and remove it from the **rdoResultsets** collection.

rdoResultset Events

The following events are fired as the **rdoResultset** object is manipulated. These can be used to micro-manage result sets or to synchronize other processes with the operations performed on the **rdoResultset** object.

Event Name	Description
Associate	Fired after a new connection is associated with the object.
ResultsChange	Fired after current rowset is changed (multiple result sets).
Dissociate	Fired after the connection is set to nothing.
QueryComplete	Fired after a query has completed.
RowStatusChange	Fired after the state of the current row has changed (edit, delete, insert).
RowCurrencyChange	Fired after the current row pointer is repositioned.
WillAssociate	Fired before a new connection is associated with the object.
WillDissociate	Fired before the connection is set to nothing.
WillUpdateRows	Fired before an update to the server occurs.

Executing Multiple Operations on a Connection

If there is an unpopulated **rdoResultset** pending on a data source that can only support a single operation on an **rdoConnection** object, you cannot create additional **rdoQuery** or **rdoResultset** objects**,** or use the **Refresh** method on the **rdoTable** object until the **rdoResultset** is flushed, closed, or fully populated. For example, when using SQL Server 4.2 as a data source, you cannot create an additional **rdoResultset** object until you move to the last row of the last result set of the current **rdoResultset** object. To populate the result set, use the **MoreResults** method to move through all pending result sets, or use the **Cancel** or **Close** method on the **rdoResultset** to flush all pending result sets.

Handing Beginning and End of File Conditions

When you create an **rdoResultset**, the current row is positioned to the first row if there are any rows. If there are no rows, the **RowCount** property setting is 0, and the **BOF** and **EOF** property settings are both **True**.

Note An **rdoResultset** may not be updatable even if you request an updatable **rdoResultset**. If the underlying database, table, or column isn't updatable, or if your user does not have update permission, all or portions of your **rdoResultset** may

be read-only. Examine the **rdoConnection**, **rdoResultset,** and **rdoColumn** objects' **Updatable** property to determine if your code can change the rows.

Closing rdoResultset objects

Use the **Close** method to remove an **rdoResultset** object from the **rdoResultsets** collection, disassociate it from its connection, and free all associated resources. No events are fired when you use the **Close** method.

Setting the **ActiveConnection** property to **Nothing** removes the **rdoResultset** object from the **rdoResultsets** collection and fires events, but does not deallocate the object resources. Setting the **rdoResultset** object's **ActiveConnection** property to a valid **rdoConnection** object causes the **rdoResultset** object to be re-appended to the **rdoResultsets** collection of the **rdoConnection** object.

Addressing rdoResultset Objects

The default collection of an **rdoResultset** is the **rdoColumns** collection, and the default property of an **rdoColumn** object is the **Value** property. You can simplify your code by taking advantage of these defaults. For example, the following lines of code all set the value of the `PubID` column in the current row of an **rdoResultset**:

```
MyRs.rdoColumns("PubID").Value = 99
MyRs("PubID") = 99
MyRs!PubID = 99
' This is the first column
' returned by the SELECT statement...
MyRs(0) = 99
```

The **Name** property of an **rdoResultset** object contains the first 255 characters of the query used to create the resultset, so it is often unsuitable as an index into the **rdoResultsets** collection especially since several queries might be created with the same SQL query.

You can refer to **rdoResultset** objects by their position in the **rdoResultsets** collection using this syntax (where *n* is the *n*th member of the zero-based **rdoResultsets** collection):

```
rdoResultsets(n)
```

```
    Do Until rs.EOF
        For Each cl In rs.rdoColumns
            Debug.Print cl.Value,
        Next
        rs.MoveNext
    Debug.Print
    Loop
    Debug.Print "Row count="; rs.RowCount

Loop Until rs.MoreResults = False
End Sub
```

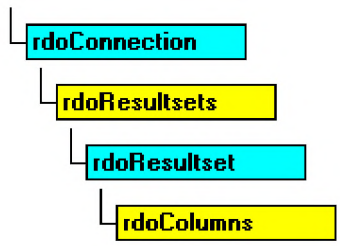
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoResultsets Collection

See Also Example Properties Methods Events

The **rdoResultsets** collection contains all open **rdoResultset** objects in an **rdoConnection**.



Remarks

A new **rdoResultset** is automatically added to the **rdoResultsets** collection when you open the object, and it's automatically removed when you close it. Several **rdoResultset** objects might be active at any one time.

Use the **Close** method to remove an **rdoResultset** object from the **rdoResultsets** collection, disassociate it from its connection, and free all associated resources. No events are fired when you use the **Close** method.

Setting the **ActiveConnection** property to **Nothing** removes the **rdoResultset** object from the **rdoResultsets** collection and fires events, but does not deallocate the object resources. Setting the **rdoResultset** object's **ActiveConnection** property to a valid **rdoConnection** object causes the **rdoResultset** object to be re-appended to the **rdoResultsets** collection.

Note RDO 1.0 collections behave differently than Data Access Object (DAO) collections. When you **Set** a variable containing a reference to a RDO object like **rdoResultset**, the existing **rdoResultset** is *not* closed and removed from the **rdoResultsets** collection. The existing object remains open and a member of its respective collection.

In contrast, RDO 2.0 collections do not behave in this manner. When you use the Set statement to assign a variable containing a reference to an RDO object, the existing object *is* closed and removed from the associated collection. This change is designed to make RDO more compatible with DAO.

Managing the rdoResultsets Collection

When you use the **OpenResultset** method against an **rdoConnection** or **rdoQuery**, and assign the result to an existing **rdoResultset** object, the existing object is maintained and a new **rdoResultset** object is appended to the **rdoResultsets** collection. When performing similar operations using the Microsoft Jet database engine and Data Access Objects (DAO), existing recordset objects are automatically closed when the variable is assigned, and no two **Recordsets** collection members can have the same name. For example, using RDO:

```
Dim rs as rdoResultset
Dim cn as rdoConnection
Set cn = OpenConnection....
Set rs = cn.OpenResultset("Select * from Authors", _
    rdOpenStatic)
```

Visual Basic: RDO Data Control

rdoResultset Object, rdoResultsets Collection Example

The following example illustrates execution of a multiple result set query. While this query uses three SELECT statements, only two return rows to your application. The subquery used instead of a join does not pass rows outside the scope of the query itself. This is also an example of a simple parameter query that concatenates the arguments instead of using an **rdoQuery** to manage the query. The **OpenResultset** also runs asynchronously the code checks for completion of the operation by polling the **StillExecuting** property.

```
Private Sub ShowResultset_Click()
Dim rs As rdoResultset
Dim cn As New rdoConnection
Dim cl As rdoColumn
Dim SQL As String
Const None As String = ""

cn.Connect = "uid=;pwd=;server=SEQUEL;" _
    & "driver={SQL Server};database=pubs;" _
    & "DSN='';"

cn.CursorDriver = rdUseOdbc
cn.EstablishConnection rdDriverNoPrompt

SQL = "Select Au_Lname, Au_Fname" _
    & " From Authors A" _
    & " Where Au_ID in " _
    & " (Select Au_ID" _
    & "     from TitleAuthor TA, Titles T" _
    & "     Where TA.Au_ID = A.Au_ID" _
    & "     And TA.Title_ID = T.Title_ID " _
    & "     And T.Title Like '" _
    & InputBox("Enter search string", , "C") & "%')" _
    & "Select * From Titles Where price > 10"

Set rs = cn.OpenResultset(SQL, rdOpenKeyset, _
    rdConcurReadOnly, rdAsyncEnable + rdExecDirect)

Debug.Print "Executing ";
While rs.StillExecuting
    Debug.Print ".";
    DoEvents
Wend

Do
    Debug.Print String(50, "-")  _
   & "Processing Result Set " & String(50, "-")
    For Each cl In rs.rdoColumns
        Debug.Print cl.Name,
    Next
    Debug.Print
```

```
Set rs = cn.OpenResultset("Select * from Titles", _
    rdOpenDynamic)
```

This code opens two separate **rdoResultset** objects; both are stored in the **rdoResultsets** collection. After this code runs, the second query, which is stored in **rdoResultsets**(1), is assigned to the **rdoResultset** variable `rs`. The first query is available and its cursor is still available by referencing **rdoResultsets**(0). Because of this implementation, more than one member of the **rdoResultsets** collection can have the same name.

This behavior permits you to maintain existing **rdoResultset** objects, which are maintained in the **rdoResultsets** collection, or close them as needed. In other words, you must explicitly close any **rdoResultset** objects that are no longer needed. Simply assigning another **rdoResultset** to a **rdoResultset**-type variable has no affect on the existing **rdoResultset** formerly referenced by the variable. Note that the procedures and other temporary objects created to manage the **rdoResultset** are maintained on the remote server as long as the **rdoResultset** remains open.

If you write an application that does not close each **rdoResultset** before opening additional **rdoResultset** objects, the number of procedures maintained in *TempDB* or elsewhere on the server increases each time another **rdoResultset** object is opened. In addition those resultsets might require significant client or server resources to store keysets or row values. Over time, this behavior can overflow the capacity of the server or workstation resources.

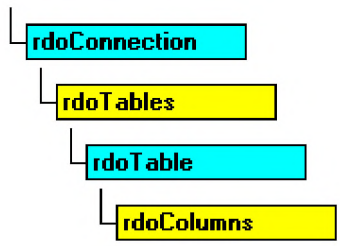
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoTable Object

See Also Example Properties Methods Events

An **rdoTable** object represents the stored definition of a base table or an SQL view.



Remarks

Note You are discouraged from using the **rdoTable** object and **rdoTables** collection to manage or inspect the structure of your database tables. This object is maintained for backward compatibility and might not be supported in future versions of Visual Basic or RDO.

You can map a table definition using an **rdoTable** object and determine the characteristics of an **rdoTable** object by using its methods and properties. For example, you can:

- Examine the column properties of any table in an ODBC database. (Note that all **rdoTable** object properties are read-only.)
- Use the **OpenResultset** method to create an **rdoResultset** object based on all of the rows of the base table.
- Use the **Name** property to determine the name of the table or view.
- Use the **RowCount** property to determine the number of rows in the table or view. Referencing the **RowCount** property causes the query to be completed just as if you had used the **MoveLast** method.
- Use the **Type** property to determine the type of table. The ODBC data source driver determines the supported table types.
- Use the **Updatable** property to determine if the table supports changes to its data.

You cannot reference the **rdoTable** objects until you have populated the **rdoTables** collection because it is not automatically populated when you connect to a data source. To populate the **rdoTables** collection, use the **Refresh** method or reference individual members of the collection by their ordinal number.

When you use the **OpenResultset** method against an **rdoTable** object, RDO executes a "SELECT * FROM *table*" query that returns *all* rows of the table using the cursor type specified. By default, a forward-only cursor is created.

You cannot define new tables or change the structure of existing tables using RDO or the RemoteData control. To change the structure of a database or perform other administrative functions, use SQL queries or the administrative tools that are provided with the database.

The default collection of an **rdoTable** object is the **rdoColumns** collection. The default property of an **rdoTable** is the **Name** property. You can simplify your code by using these defaults. For example, the following statements are identical in that they both print the number corresponding to the column data type of a column in an **rdoTable** using a RemoteData control:

```
Print RemoteData1.Connection.rdoTables _
   ("Publishers").rdoColumns("PubID").Type
Print RemoteData1.Connection("Publishers"). _
   ("PubID").Type
```

The **Name** property of an **rdoTable** object isn't the same as the name of an object variable to which it's assigned it is derived from the name of the base table in the database.

You refer to an **rdoTable** object by its **Name** property setting using this syntax:

```
rdoTables("Authors")   'Refers to the Authors table
```

Or

```
rdoTables!Authors      'Refers to the Authors table
```

You can also refer to **rdoTable** objects by their position in the **rdoTables** collection using this syntax (where *n* is the *n*th member of the zero-based **rdoTables** collection):

```
rdoTables(n)
```

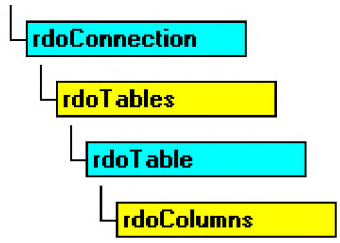
Visual Basic: RDO Data Control

Visual Studio 6.0

rdoTables Collection

See Also Example Properties Methods Events

The **rdoTables** collection contains all stored **rdoTable** objects in a database.



Remarks

Note You are discouraged from using the **rdoTable** object and **rdoTables** collection to manage or inspect the structure of your database tables. This object is maintained for backward compatibility and might not be supported in future versions of Visual Basic.

For performance reasons, you cannot reference an **rdoTable** object until you have first populated the **rdoTables** collection because it is not automatically populated when you connect to a data source. To populate the **rdoTables** collection, use the **Refresh** method or reference individual members of the collection by their ordinal number. Depending on the number of tables in your database, this can take quite some time.

Visual Studio 6.0

Visual Basic: MSChart Control

Rect Object

See Also Example Properties Methods Events

Defines a coordinate location.

Syntax

Rect

Visual Basic: MSChart Control

LocationRect Property, Rect Object Example

The example increases the size of the chart plot using the **LocationRect** property and the x and y properties of the **Rect** object.

```
Private Sub Command1_Click()
   ' Increase the size of the chart plot.
   MSChart1.Plot.AutoLayout = False
   With MSChart1.Plot.LocationRect
      .Min.x = .Min.x * 1.2
      .Min.y = .Min.y * 1.2
      .Max.x = .Max.x * 1.2
      .Max.y = .Max.y * 1.2
   End With
End Sub
```

Visual Basic Extensibility Reference

Visual Studio 6.0

Reference Object

See Also Example Properties Methods Events Specifics

Represents a reference to a type library or a project.

Remarks

Use the **Reference** object to verify whether a reference is still valid.

The **IsBroken** property returns **True** if the reference no longer points to a valid reference. The **BuiltIn** property returns **True** if the reference is a default reference that can't be moved or removed. Use the **Name** property to determine if the reference you want to add or remove is the correct one.

Visual Basic Extensibility Reference

Visual Studio 6.0

References Collection

See Also Example Properties Methods Events Specifics

Represents the set of references in the project.

Remarks

Use the **References** collection to add or remove references. The **References** collection is the same as the set of references selected in the **References** dialog box.

Visual Basic Extensibility Reference

Visual Studio 6.0

ReferencesEvents Object

See Also Example Properties Methods Events Specifics

Returned by the **ReferencesEvents** property.

Remarks

The **ReferencesEvents** object is the source of events that occur when a reference is added to or removed from a project. The ItemAdded event is triggered after a reference is added to a project. The ItemRemoved event is triggered after a reference is removed from a project.

Visual Basic: RDO Data Control

Visual Studio 6.0

RemoteData Control

See Also Example Properties Methods Events

Provides access to data stored in a remote ODBC data source through bound controls. The **RemoteData** control enables you to move from row to row in a result set and to display and manipulate data from the rows in bound controls.

Syntax

RemoteData

Remarks

The **RemoteData** control provides an interface between Remote Data Objects (RDO) and data-aware bound controls. With the **RemoteData** control, you can:

- Establish a connection to a data source based on its properties.
- Create an **rdoResultset**.
- Pass the current row's data to corresponding bound controls.
- Permit the user to position the current row pointer.
- Pass any changes made to the bound controls back to the data source.

Overview

Without a **RemoteData** control, a **Data** control or its equivalent, data-aware (bound) controls on a form can't automatically access data. The RemoteData and Data controls are examples of *DataSource Controls*. You can perform most remote data access operations using the DataSource controls without writing any code at all. Data-aware controls bound to a DataSource control automatically display data from one or more columns for the current row or, in some cases, for a set of rows on either side of the current row. DataSource controls perform all operations on the current row.

The RemoteData DataSource Control

If the **RemoteData** control is instructed to move to a different row, all bound controls automatically pass any changes to the **RemoteData** control to be saved to the ODBC data source. The **RemoteData** control then moves to the requested row and passes back data from the current row to the bound controls where it's displayed.

The **RemoteData** control automatically handles a number of contingencies including empty result sets, adding new rows, editing and updating existing rows, converting and displaying complex data types, and handling some types of errors. However, in more sophisticated applications, you must trap some error conditions that the **RemoteData** control can't handle. For example, if the remote server has a problem accessing the data source, the user doesn't have permission, or the query can't be executed as coded, a trappable error results. If the error occurs *before* your application procedures start, or as a result of some internal errors, the Error event is triggered.

Operation

Use the **RemoteData** control properties to describe the data source, establish a connection, and specify the type of cursor to create. If you alter these properties once the result set is created, use the **Refresh** method to rebuild the underlying **rdoResultset** based on the new property settings.

The **RemoteData** control behaves like the Jet-driven **Data** control in most respects. The following guidelines illustrate a few differences that apply when setting the **SQL** property.

You can treat the **RemoteData** control's **SQL** property like the **Data** control's **RecordSource** property except that it cannot accept the name of a table by itself, unless you populate the **rdoTables** collection first. Generally, the **SQL** property specifies an SQL query. For example, instead of just "Authors", you would code "SELECT * FROM AUTHORS" which provides the same functionality. However, specifying a table in this manner is not a good programming practice as it tends to return too many rows and can easily exhaust workstation resources or lock large segments of the database.

The result set created by the **RemoteData** control might not be in the same order as the **Recordset** created by the **Data** control. For example, if the **Data** control's **RecordSource** property is set to "Authors" and the **RemoteData** control's **SQL** property is set to "SELECT * FROM AUTHORS", the first record returned by Jet to the **Data** control is based on the first available index on the Authors table. The **RemoteData** control, however, returns the first row returned by the remote database engine based on the physical sequence of the rows in the database, regardless of any indexes. In some cases, the order of the records could be identical, but not always.

This difference in behavior can affect how bound controls handle the resulting rows especially multiple-row bound controls like the **DataGrid** control. You can manipulate the **RemoteData** control with the mouse to move the current row pointer from row to row, or to the beginning or end of the **rdoResultset** by clicking the control. As you manipulate the **RemoteData** control buttons, the current row pointer is repositioned in the **rdoResultset**. You cannot move off either end of the **rdoResultset** using the mouse. You also can't set focus to the **RemoteData** control.

Other Features

You can use the objects created by the **RemoteData** control to create additional **rdoConnection, rdoResultset**, or **rdoQuery**objects.

You can set the **RemoteData** control **Resultset** property to an **rdoResultset** created independently of the control. If this is done, the **RemoteData** control properties are reset based on the new **rdoResultset** and **rdoConnection**.

You can set the **Options** property to enable asynchronous creation of the **rdoResultset** (**rdAsyncEnable**) or to execute the query without creating a temporary stored procedure (**rdExecDirect**).

The Validate event is triggered before each reposition of the current row pointer. You can choose to accept the changes made to bound controls or cancel the operation using the Validate event's *action* argument.

The **RemoteData** control can also manage what happens when you encounter an **rdoResultset** with no rows. By changing the **EOFAction** property, you can program the **RemoteData** control to enter **AddNew** mode automatically.

Note If you have an **Image** control bound to an image-containing field in a **RemoteData** control, and the **RemoteData** control uses batch cursors (that is, *CursorDriver* = *rdUseClientBatch*), the **Image** control doesn't display the image.

Programmatic Operation

To create an **rdoResultset** programmatically with the **RemoteData** control:

- Set the **RemoteData** control properties to describe the desired characteristics of the **rdoResultset**.
- Use the **Refresh** method to begin the automated process or to create the new **rdoResultset**. Any existing **rdoResultset** is discarded.

All of the **RemoteData** control properties and the new **rdoResultset** object may be manipulated independently of the **RemoteData** control—with or without bound controls. The **rdoConnection** and **rdoResultset** objects each have properties and methods of their own that can be used with procedures that you write.

For example, the **MoveNext** method of an **rdoResultset** object moves the current row to the next row in the **rdoResultset**. To invoke this method with an **rdoResultset** created by a **RemoteData** control, you could use this code:

```
RemoteData1.Resultset.MoveNext
```

Resultset Does Not Automatically Update Bound Controls

Assigning a resultset to a **RemoteData** Control (**RDC**) doesn't update bound controls. When you bind a control to the resultset of an **RDC**, the resultset doesn't automatically display in the control. To illustrate this:

1. Start Visual Basic and open a Standard EXE project.
2. Reference the **RDC**.
3. Place an **RDC** on the form.
4. Place a **TextBox** control on the form.
5. Set the following **TextBox** properties:
   DataSource: MSRDC1

   DataField: au_lname
6. Place a **CommandButton** control on the form and add the following code to its Click event:

```
Dim cn As New rdoConnection
cn.Connect = _
"dsn=pinkpearl;database=rdobugs;uid=rdo;pwd="
cn.EstablishConnection
Set MSRDC1.Resultset = cn.OpenResultset("select * _
from authors]")
```

7. Run the project (F5).
8. Click the **CommandButton**.

Notice that the bound control does not populate with data as you would expect. You must issue the command `MSRDC1.Refresh` for the bound control to populate, which causes the server to send the entire resultset again. (Note that this can take a long time in some situations.)

To work around this problem, set any bound control's datafield after setting the resultset in code. For example, after the line:

```
Set MSRDC1.Resultset = cn.OpenResultset("select * _
from authors]")
```

you would add:

```
Text1.DataField = "au_lname"
```

which forces the binding manager to set and update the bindings, which populates the bound control with data.

Bound Image or PictureBox Control Doesn't Display Picture When RDC Uses Batch Cursors

When you are using an **Image** or **PictureBox** control bound to an image-containing field in an **RDC**, and the **RDC** uses batch cursors (**CursorDriver** = *rdUseClientBatch*), be aware that the **Image** or **PictureBox** control doesn't display the image. To correctly display the image, either set the **RDC**'s **Options** property to 128 (*rdFetchLongColumns*), or use a different cursor.

Do Not Use Forward-only Resultsets

When you attempt to assign a forward-only resultset to an **RDC**, you get an "invalid object" error. To illustrate this situation:

1. Start Visual Basic.
2. Place a **RemoteData** control on Form1.
3. Add a reference to **RDO** through the References command on the Project menu.
4. Add the following code to the Form_Load event:

```
Dim x as new rdoConnection
Dim y as rdoQuery
x.Connect = "DSN=Union;UID=rdo;PWD="
x.EstablishConnection
Set y = x.CreateQuery("Query1", "SELECT * FROM _
authors")
x.Query1
' invalid object error occurs on next line
Set MSRDC1.Resultset = x.LastQueryResults
```

5. Press F5.

The reason this error occurs is that it uses a forward-only resultset which cannot be assigned to the **RDC**. In order to assign a resultset to an **RDC**, it must be either keyset or static. For example:

```
Dim x As New rdoConnection
Dim y As rdoQuery
x.Connect = "DSN=Union;database=rdobugs;UID=rdo;PWD="
  x.EstablishConnection
Set y = x.CreateQuery("Query1", "SELECT * FROM _
  authors")
y.CursorType = rdOpenKeyset
y.LockType = rdConcurRowVer
x.Query1
Set MSRDC1.Resultset = x.LastQueryResults
```

Visual Basic: DataRepeater Control

Visual Studio 6.0

RepeaterBinding Object

See Also Example Properties Methods Events

The **RepeaterBinding** object represents a bindable property of a component.

Syntax

RepeaterBinding

Remarks

Use the **RepeaterBinding** object at run time to change the contents of a **DataRepeater** control by changing how fields are bound.

Visual Basic: DataRepeater Control

RepeaterBinding Object Example

The example below first prints the existing property names of the **RepeaterBindings** collection, then adds a **DataBinding** object to the collection, and finally changes the format of a **DataBinding** object.

```
Private Sub DataRepeater1_RepeatedControlLoaded()
   Dim rb As RepeaterBinding

   For Each rb In DataRepeater1.RepeaterBindings
      Debug.Print rb.PropertyName ' Print all property names.
   Next

   With DataRepeater1
      ' Add a new RepeaterBinding object.
      .RepeaterBindings.Add "cleared", "cleared"
      ' Change the Format to all caps.
      .RepeaterBindings(2).DataFormat.Format = ">"
   End With
End Sub
```

Visual Basic: RichTextBox Control

Visual Studio 6.0

RichTextBox Control

See Also Example Properties Methods Events

The **RichTextBox** control allows the user to enter and edit text while also providing more advanced formatting features than the conventional **TextBox** control.

Syntax

RichTextBox

Remarks

The **RichTextBox** control provides a number of properties you can use to apply formatting to any portion of text within the control. To change the formatting of text, it must first be selected. Only selected text can be assigned character and paragraph formatting. Using these properties, you can make text bold or italic, change the color, and create superscripts and subscripts. You can also adjust paragraph formatting by setting both left and right indents, as well as hanging indents.

The **RichTextBox** control opens and saves files in both the RTF format and regular ASCII text format. You can use methods of the control (**LoadFile** and **SaveFile**) to directly read and write files, or use properties of the control such as **SelRTF** and **TextRTF** in conjunction with Visual Basic's file input/output statements.

The **RichTextBox** control supports object embedding by using the **OLEObjects** collection. Each object inserted into the control is represented by an **OLEObject** object. This allows you to create documents with the control that contain other documents or objects. For example, you can create a document that has an embedded Microsoft Excel spreadsheet or a Microsoft Word document or any other OLE object registered on your system. To insert objects into the **RichTextBox** control, you simply drag a file (from the Windows 95 Explorer for example), or a highlighted portion of a file used in another application (such as Microsoft Word), and drop the contents directly onto the control.

The **RichTextBox** control supports both clipboard and OLE drag/drop of OLE objects. When an object is pasted in from the clipboard, it is inserted at the current insertion point. When an object is dragged and dropped into the control, the insertion point will track the mouse cursor until the mouse button is released, causing the object to be inserted. This behavior is the same as Microsoft Word.

To print all or part of the text in a **RichTextBox** control use the **SelPrint** method.

Because the **RichTextBox** is a data-bound control, you can bind it with a **Data** control to a Binary or Memo field in a Microsoft Access database or a similar large capacity field in other databases (such as a TEXT data type field in SQL Server).

The **RichTextBox** control supports almost all of the properties, events and methods used with the standard **TextBox** control, such as **MaxLength**, **MultiLine**, **ScrollBars**, **SelLength**, **SelStart**, and **SelText**. Applications that already use **TextBox** controls can easily be adapted to make use of **RichTextBox** controls. However, the **RichTextBox** control doesn't have the same 64K character capacity limit of the conventional **TextBox** control.

Distribution Note To use the **RichTextBox** control in your application, you must add the Richtx32.ocx file to the project. When distributing your application, install the Richtx32.ocx file in the user's Microsoft Windows SYSTEM directory. For more information on how to add a custom control to a project, see the *Programmer's Guide*.