Quick Start Guide

A.1 Introduction

This document is intended for Delphi Developers new to NexusDB. Generally speaking these new users can be split into 3 groups:

1. Developers who have used only TTable components, usually with a Paradox (PX) backend,
2. Developers who are upgrading from FlashFiler, and,
3. Those who are coming from a different BDE replacement DBMS.

The majority of this manual is aimed at those developers in groups 1 and 3 above. However it is recommended that current FF users should also at least browse the entire manual as the section on upgrading from FF2 to NexusDB is rather brief and to the point.

A.1.1 Layout of this Guide

Section A.2 takes you through the installation and compilation of the packages and tools that make up NexusDB.
Section A.3 provides a quick overview of the NexusDB components.
Section A.4 is about getting up to speed with the "simplest" NexusDB application – a single executable (sometimes referred to as a SingleExe app). This is using an embedded server engine: it is the NexusDB way of compiling the database engine directly into an application, resulting in a single exe for distribution. No DLL's to distribute - no server to install on-site and configure. Just the application and tables (if you also auto-create the tables, you need only distribute the exe!).
Section A.5 will help to get you quickly up and running with the full multi-user C/S version of NexusDB. Instructions are given on how to set up a generic data module to enable connection to a remote server (remote server = you have a NexusDB Server.exe running separately to the main application, usually on a PC acting as the designated NexusDB file-server on your network).
Section A.6 shows how easy it is to use both the server engines in the one application at the same time.
Section A.7 introduces you to database conversion, batch mode (blockread) access and in-memory tables.
Section A.8 provides a very cook-book approach to quickly upgrading existing FF2 applications to NexusDB. One of the elegant features of Nexus is the ability for FF2 and NexusDB components to co-exist in harmony on the same form in a project. This allows you to "incrementally" convert your existing apps from FF2 to NexusDB – something that is explored in this section.
Section A.9 details the main differences and improvements of NexusDB over FF2.
Section A.10 explains Transaction and their isolation levels. It also deals with deadlocks in multi-user environments and techniques to resolve these issues.
Section A.11 gives some general hints on how to improve you NexusDB code and application runtime speed.
And the document ends with **Section A.12** on how to get quick support and the use of the newsgroups.

A.2 Installation and initial compilation

NexusDB is delivered as a source only installation to keep the download size for our customers as small as possible. The installation and initial compilation is straightforward as shown in the following steps:

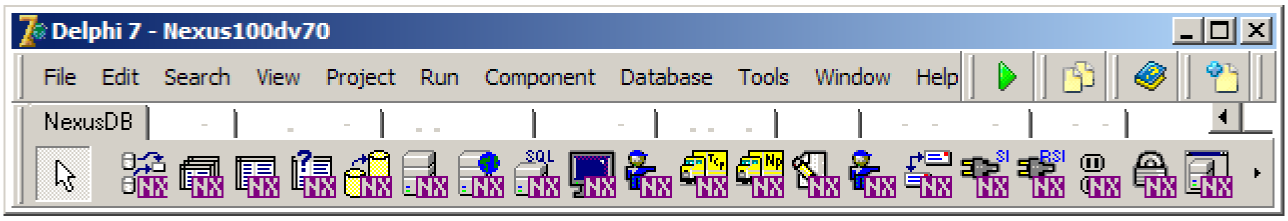
1. **Uninstall any previously installed NexusDB packages from any and all Delphi Versions on your machine**
2. Ensure that you have closed down any running instances of Delphi. The installer detects this in order to be able to properly install the components into your IDE
3. Execute either the NexusDBSetup.exe or NexusTrialSetup.exe file
4. The install wizard will then lead you through the installation in the usual fashion
5. If you're installing the trial or dcu/bpl beta version you are now finished.
6. Start Delphi, close whatever project that might be loaded, and go to Open Project
7. Select the Nexus1PackageX0.bpg package from the correct DelphiX subdirectory (x stands for your version of Delphi, for eg., Nexus1Package70.bpg from the \Delphi7 subdirectory)
8. After the project group is loaded please go to the Project menu and do a "Build all projects" - this will compile all the NexusDB packages
9. After compilation open the Project Manager (CTRL-ALT-F11) and select the Nexus100dv70.dpk project
10. Right mouse click over this project to bring up the popup menu and select "Install"
11. A message box should appear to inform you that NexusDB components have been added to the NexusDB palette tab
12. Select File/Close All (save changes)
13. Due to a bug in Delphi, you should now close down and restart Delphi. If you do not do this, then the executables in the next step will be built with runtime packages, even though the project options specify otherwise.
14. Open Project Nexus1ApplicationsX0.bpg from the same directory as step 7 above
15. Again do a Project/Build All to create Server, Enterprise Manager and Tools executables which should end up in the <install directory>\Bin subfolder

Table 1: NexusDB Installation Steps

If you have made it this far, you should now have successfully installed NexusDB into your version of Delphi and be ready to use it in your projects!

A.3 Components on the VCL Palette

When just starting out, and for the purposes of what's in this document, the following components are used. From the NexusDB tab:



Screenshot 1: Component Palette

Here's a short desription of what each component is and what its place is in NexusDB:

Component	Short Description
TnxSession	The NexusDB session component. Required component to connect your database/table/query components to a remote or embedded server engine component.
TnxDatabase	The NexusDB equivalent of TDatabase. Provides alias and database path handling. Not required, although for manageability reasons it's recommended to use one.
TnxTable	NexusDB equivalent of TTable. Use for navigational access to tables by a NexusDB server.
TnxQuery	NexusDB TQuery equivalent. Extensive support for ANSI SQL/DDL provides strong access and data manipulating capabilities.
TnxBackupController	The backup controller allows backups with full data integrity at any time, even if the server is running 24/7.
TnxServerEngine	The server engine contains the core NexusDB engine. Drop one on a form or data module in your application to create a single user database application. Connect with transports, SQLEngine etc to create custom stand-alone database servers.
TnxRemoteServerEngine	Client-side server engine. Together with a transport, it provides remote access to data on a stand-alone server.
TnxSQLEngine	SQL engine. Required for single-user applications that use SQL statements (TnxQuery components), or in standalone servers.
TnxSimpleMonitor	Used to develop server-side modules that monitors events (table open, record update, etc) and performs actions based on those events. Use to extend the server functionality.
TnxServerCommandHandler	The server command handler is responsible for decoding client requests, translation and forwarding of this request to the server engine and issuing the results back to the client.
TnxWinsockTransport	This component connects two machines via TCP/IP (Winsock). At least one of the transports is required to create a client application that connects to a remote server.
TnxNamedPipeTransport	This component connects two machines via Named Pipes. At least one of the transports is required to create a client application that connects to a remote server.

TnxEventLog	The event log is a centralized way for handling log entries. All the other components can be connected to a single event log.
TnxSimpleCommandHandler	Command handler that does not depend on any database components. Use if you need to create a message system or similar that doesn't require database functionality.
TnxSimpleSession	Session component that does not depend on any database components. Use if you need to create a message system or similar that doesn't require database functionality.
TnxServerInfoPlug-in	Server-side plug-in example component that implements a few nice to have utility functions (get server-side time, get unique server ID).
TnxRemoteServerInfoPlug-in	The client-side plug-in example component for accessing the TnxServerInfoPlug-in functions from a remote client.
TnxServerInfoPlug-inCommandHandler	Use this on the server side to route messages from remote clients to the TnxServerInfoPlug-in.
TnxServerManager	This component manages a server engine (and all its attached server modules) and their settings.

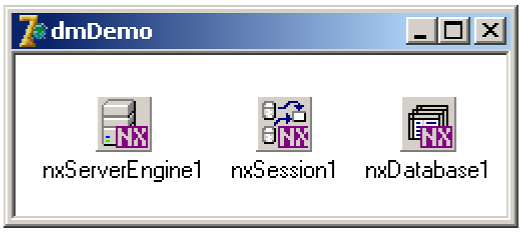
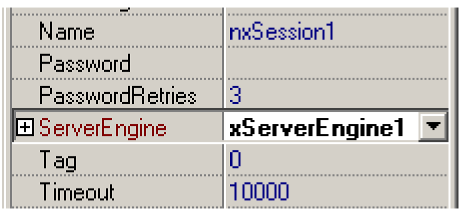
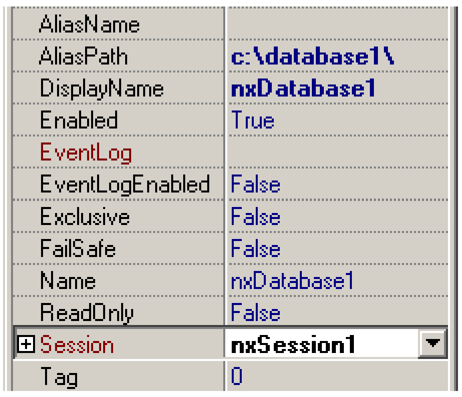
Table 2: NexusDB Components

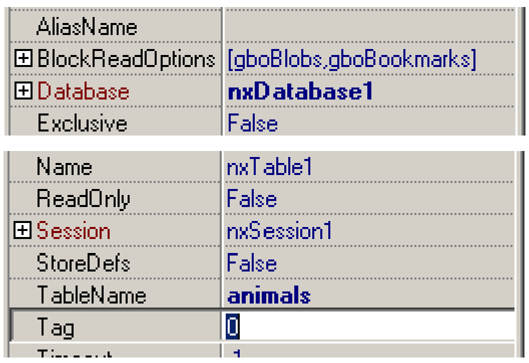
A.4 SingleExe Applications – The Embedded Server

NexusDB is a true Client/Server DBMS. Even so, one of its many features is that the engine can be compiled directly into a project so that your application can be distributed as a single exe without the need to separately install, configure, and maintain the NexusDB Server DBMS backend.
Tests have shown that over a wide range of database operations and table structures a NexusDB application built this way is roughly twice as fast as a single exe equivalent PX application. This is actually quite amazing as you would not expect a Client/Server backend to outperform PX in single-exe mode.
In this section we show you how to set up a general Embedded Server in a data module that can be used in any Single-Exe application and then provide a complete example of a single-exe application.

A.4.1 Setting up a general NexusDB Embedded Server Data Module

This section explains how to set up a generic simple data module that you can use to create database enabled applications. In the Delphi IDE you will need to create a new project. Now follow these steps to create the data module shown in the figure below (for Delphi 7):

Step number and description	Graphical result
Create a new Data Module in the IDE. Drop a server engine (TnxServerEngine) on it. Do not bother setting any properties. Now add in turn a TnxSession and a TnxDatabase component to the data module as shown in the figure.	
Connect the nxSession1 component to your NexusDB ServerEngine1 component (pull down list of the Server Engine property). The properties of your session should now look as shown on the right.	 Name: nxSession1 Password: PasswordRetries: 3 ⊞ ServerEngine: xServerEngine1 Tag: 0 Timeout: 10000
Connect the nxDatabase1 component to the Session component (from the pull down list). Finally, you will need to point the AliasPath (not the AliasName) property to where your data tables reside, in this case, I've got mine sitting in C:\Database1\. The Property Inspector for your database component should now look pretty much the same as the one shown.	 AliasName: AliasPath: c:\database1\ DisplayName: nxDatabase1 Enabled: True EventLog: EventLogEnabled: False Exclusive: False FailSafe: False Name: nxDatabase1 ReadOnly: False ⊞ Session: nxSession1 Tag: 0

You are now ready to use this data module to connect to tables. For instance, add a form to your project and drop on a TnxTable. To connect to the database available in the data module, just connect the Database property via the drop down list and the correct Session will be set for you. Finally select the table for this instance in the TableName property. Your TnxTable component should now look as follows (after selecting a table).	

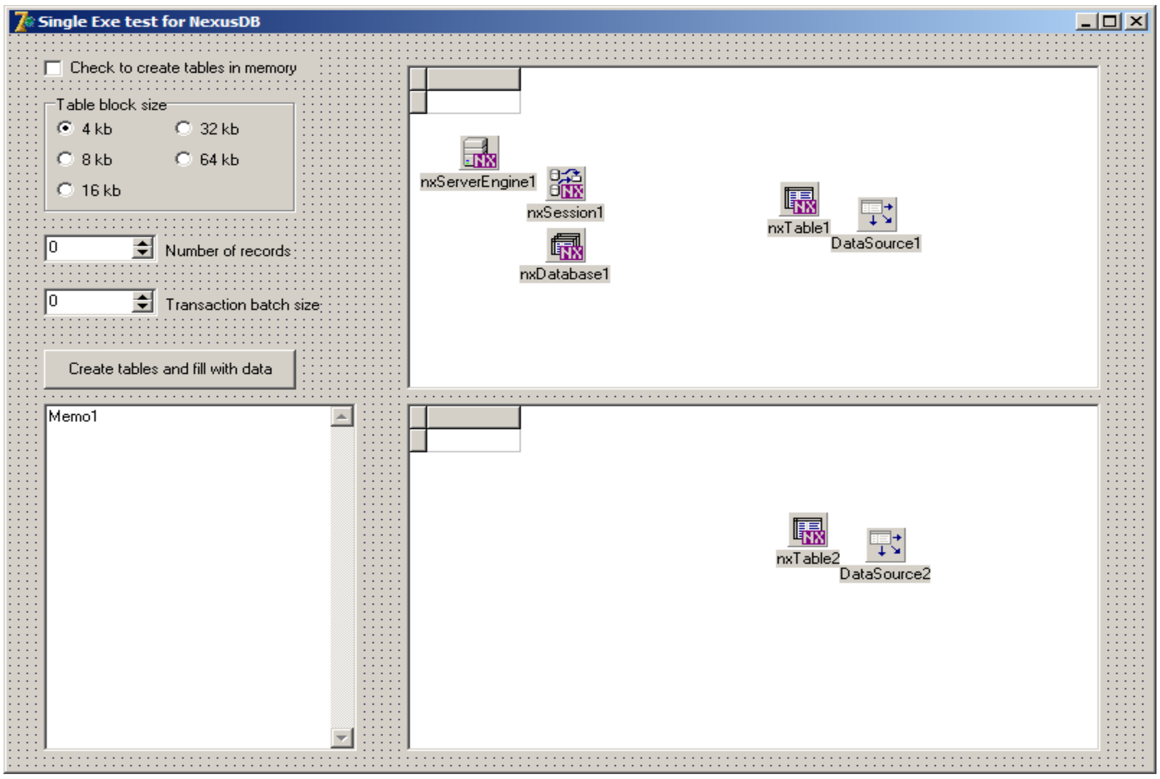
AliasName	
⊞BlockReadOptions	[gboBlobs,gboBookmarks]
⊞Database	nxDatabase1
Exclusive	False
Name	nxTable1
ReadOnly	False
⊞Session	nxSession1
StoreDefs	False
TableName	animals
Tag	0

Table 3: Setting up a general NexusDB Embedded Server Data Module

You are now ready to go! It's that straightforward! You can now connect to this data module in any single exe application by simply including it in the "uses" clause of any unit that wishes to access it.

A.4.2 An example Application

Here we will show how to build a simple database application from scratch. In the Delphi IDE you will need to create a new project. You will be building the project shown below (connect the database components in the same way as shown above for the data module.



Screenshot 2: Embedded Application Example

Alternatively you add the generic data module to your project and reference its components. The rest of the components on the form are the familiar ones from the Delphi VCL. You should have a Checkbox, a Radio Group, 2 Spin Edits, 2 Labels, a Button, a Memo, 2 data sources and 2 DBGrids. Connect the data sources to the two TnxTables by setting the Dataset properties and link the DBGrids to the data sources by setting their DataSource properties.
The only thing left to do is to make sure the tables are opened when starting the app. For this purpose simply set their ActiveRuntime properties to true.

That's it. Compile the app and run. The other controls will be uses in later in this section for tweaking the database.
Caveat: When using embedded mode always keep in mind that the IDE opens tables if you have ActiveDesignTime enabled. Trying to run your embedded application from the IDE/Debugger will fail to open the table at runtime, because it is using a different ServerEngine. Nexus only allows ONE server engine to access a file at a time. In general we recommend to always use an external server while developing.

A.4.3 Making the single-exe application path independent

Let's add a run-time enhancement to our data module so that the resulting exe is not restricted to the path coded into the AliasPath property of the database component. That is, let's make the application automatically change this to point to the same path as the exe itself (here we are assuming that the data tables will reside in the same directory as the exe).
Please note that for this to work you need to set the ActiveRuntime properties of the database and tables to false.
To have the tables created in the same directory as where the exe resides, plus to activate the NexusDB component chain, add the following code to the OnCreate event handler of the form (or data module if you used one) as follows:

```
procedure TMainformDialog.FormCreate(Sender: TObject);
begin
  nxDataBase1.AliasPath:=ExtractFilePath(paramStr(0));
  nxTable1.Active:=True;
  nxTable2.Active:=True;
end;
```

That's it. When the exe starts, the engine will look for the tables in the same directory as the executable. If you want to keep your data tables in a sub-directory away from the exe, then you would modify the above to, for example,

```
nxDataBase1.AliasPath:=extractFilePath(paramStr(0))+'\data'
```

In the same way you can set all properties of the database components at start-up. Just make sure that the components have the ActiveRuntime property set to false, if you want to use code changes.

A.4.4 Playing with the NexusDB components at runtime

To make this more fun, let's create the tables, including their field and index definitions at runtime. We'll also put in some code to let you play with the final product and experiment with various settings such as transaction batch sizes, table block sizes and in-memory tables.
Enter the following code:

```
procedure TMainformDialog.createTables;
var
  aBlockSize:TnxBlockSize;
begin
  case radioGroup1.itemIndex of
    0 : aBlockSize:=nxbs4k;
    1 : aBlockSize:=nxbs8k;
    2 : aBlockSize:=nxbs16k;
    3 : aBlockSize:=nxbs32k;
    else aBlockSize:=nxbs64k;
  end;
  with nxTable1 do
```

```
begin
  if Active then Close;
  if CheckBox1.Checked
   then TableName:='<Table1>'
    else TableName:=’Table1’;
  with FieldDefs do
  begin
    Clear;
    with AddFieldDef do
    begin
      Name:='Rec';
      DataType:=ftAutoinc;
    end;
    with AddFieldDef do
    begin
      Name:='Description';
      DataType:=ftString;
      Size:=30;
    end;
  end;
  with IndexDefs do
  begin
    Clear;
    with AddIndexDef do
    begin
      Name:='NX$Primary';
      Fields:='Rec';
      Options:=[ixPrimary,ixUnique];
    end;
  end;
  CreateTableEx(aBlockSize);
  Open;
end;
with nxTable2 do
begin
  if Active then Close;
  if CheckBox1.Checked
    then TableName:='<Table2>'
    else TableName:=’Table2’;
  with FieldDefs do
  begin
    Clear;
    with AddFieldDef do
    begin
      Name:='Number';
      DataType:=ftFloat;
    end;
```

```
      with AddFieldDef do
      begin
        Name:='Rec';
        DataType:=ftAutoinc;
      end;
      with AddFieldDef do
      begin
        Name:='When';
        DataType:=ftDateTime;
      end;
    end;
    with IndexDefs do
    begin
      Clear;
      with AddIndexDef do
      begin
        Name:='NX$Primary';
        Fields:='Number';
        Options:=[ixPrimary,ixUnique];
      end;
    end;
    CreateTableEx(aBlockSize);
    Open;
  end;
end;

procedure TMainformDialog.Button1Click(Sender: TObject);
var
  i,n:Integer;
  t:TDateTime;

Function RandomWord:String;
var
  i:integer;
begin
  result:='';
  for i:=1 to 3+Random(27) do result:=result+chr(30+Random(30));
end;

begin
  Memo1.Lines.Append('BlockSize '+RadioGroup1.Items[RadioGroup1.ItemIndex]);
  Memo1.Lines.Append('# Records '+IntToStr(SpinEdit1.Value));
  Memo1.Lines.Append('Trans size '+IntToStr(SpinEdit2.Value));
  n:=SpinEdit2.Value;
  t:=Now;
  CreateTables;
  Memo1.Lines.Append(FormatDateTime('"Create: "ss.zzz" seconds"',Now-t));
  DataSource1.DataSet:=nil;
  DataSource2.DataSet:=nil;
  t:=Now;
  nxDataBase1.StartTransaction(False);
  for i:=1 to SpinEdit1.Value do
  begin
    With nxTable1 do
    begin
      Insert;
      FieldByName('Description').asString:=RandomWord;
      Post;
    end;
    With nxTable2 do
    begin
      Insert;
      FieldByName('Number').asFloat:=Random;
      FieldByName('When').asFloat:=Now-31*Random;
      Post;
    end;
    if (i mod n)=0 then
    begin
      nxDataBase1.Commit;
      nxDataBase1.StartTransaction(False);
    end;
  end;
  nxDataBase1.Commit;
  Memo1.Lines.Append(FormatDateTime('"Append: "ss.zzz" seconds"',Now-t));
  Memo1.Lines.Append('----------------');
  DataSource1.DataSet:=nxTable1;
  DataSource2.DataSet:=nxTable2;
end;
```

Also, make sure the uses clause of the form is as follows:

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Db, DBTables, StdCtrls, Spin, Buttons, Grids, ExtCtrls,
  ComCtrls, nxllComponent, nxsdServerEngine, nxsrServerEngine,
  nxdb,nx1xAllEngines,nxllTypes, DBGrids;
```

You can now compile and run this program. Have a play with the various table sizes and transaction batch sizes. It is also interesting to see what performance improvements result as a consequence of using in-memory tables.

A.5 C/S Applications – Client application separate from the Server

In this section we set up a data module that you can use in any application that you are going to deploy as a true C/S system. That is, you will need to install and configure the NexusDB server engine (NexusDB Server.exe) on an appropriately designated file server. Your client applications must then connect to this server with an appropriate transport. We end the section with a small example C/S application.

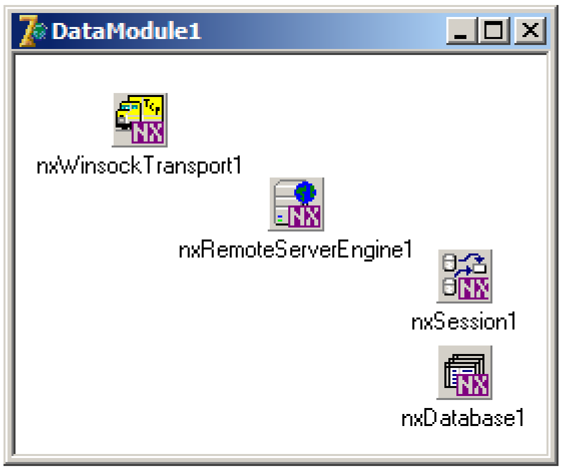
A.5.1 Before you start Delphi

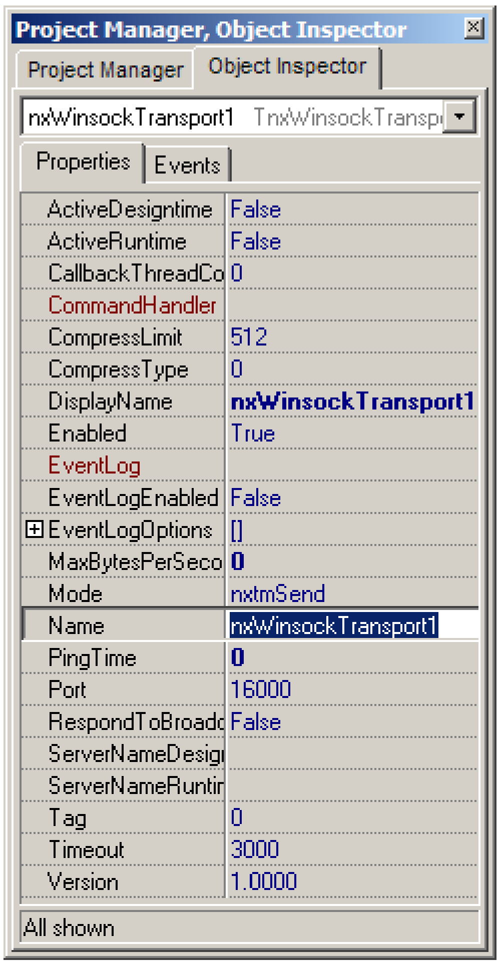
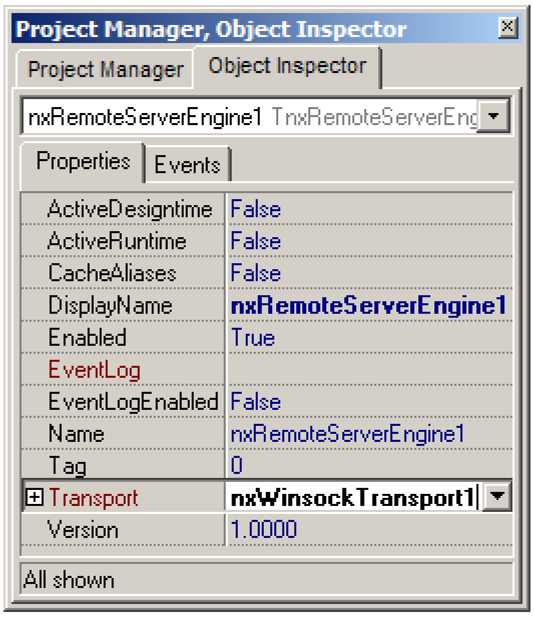
Start the NexusDB Server on your workstation and ensure that the Winsock transport is started and an appropriate alias has been created. Find more information about the NexusDB Server in section 5.

A.5.2 Setting up a generic NexusDB Data Module for C/S

In the Delphi IDE you will need to create a new project. Now follow these steps to create the data module shown in the figure below.

A.5.2.1 Creating the NexusDB Remote Server Data Module

Description	Graphical result
Create a new Data Module in the IDE.	 DataModule1 nxWinsockTransport1 nxRemoteServerEngine1 nxSession1 nxDatabase1

Place a TnxWinsockTransport component onto the form. Set its ServerNameRuntime property to YourName@ipaddresofserver (e.g. NexusDB@192.168.1.1)	
Grab a TnxRemoteServerEngine component and add it to the data module. Connect it to the Transport component by simply double-clicking the Transport property (or select from the drop-down list)	

Project Manager, Object Inspector

Project Manager | Object Inspector

nxWinsockTransport1 TnxWinsockTransp

Properties | Events

ActiveDesigntime	False
ActiveRuntime	False
CallbackThreadCo	0
CommandHandler	
CompressLimit	512
CompressType	0
DisplayName	**nxWinsockTransport1**
Enabled	True
EventLog	
EventLogEnabled	False
⊞EventLogOptions	[]
MaxBytesPerSeco	**0**
Mode	nxtmSend
Name	nxWinsockTransport1
PingTime	**0**
Port	16000
RespondToBroadc	False
ServerNameDesigı	
ServerNameRuntir	
Tag	0
Timeout	3000
Version	1.0000

All shown

Project Manager, Object Inspector

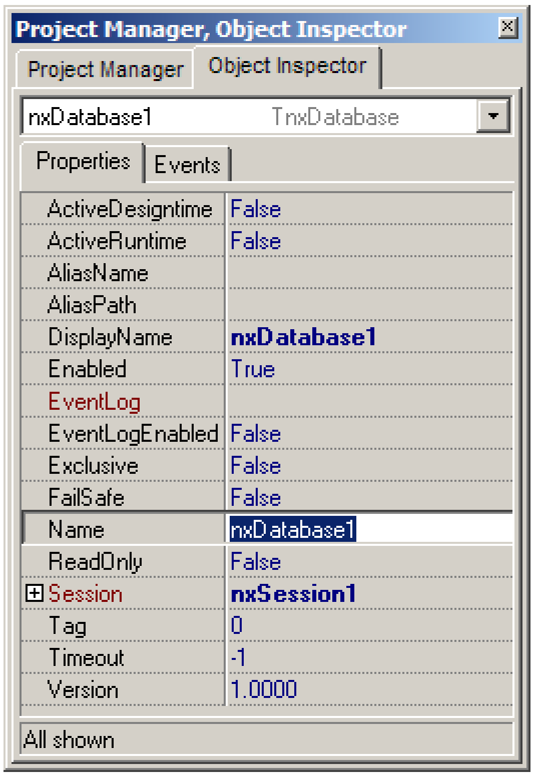
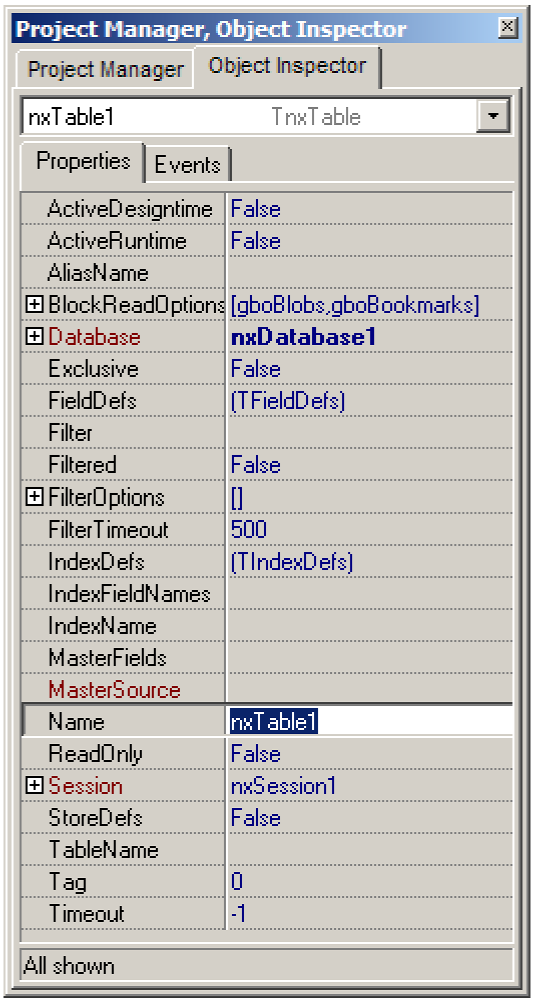
Project Manager | Object Inspector

nxRemoteServerEngine1 TnxRemoteServerEng

Properties | Events

ActiveDesigntime	False
ActiveRuntime	False
CacheAliases	False
DisplayName	**nxRemoteServerEngine1**
Enabled	True
EventLog	
EventLogEnabled	False
Name	nxRemoteServerEngine1
Tag	0
⊞Transport	**nxWinsockTransport1**
Version	1.0000

All shown

Now add in turn TnxSession and TnxDatabase components to the data module. Connect the session component to the remote server engine component. Connect the database component to the session component. You will need to select a name from the pull down list of the AliasName property which will connect you to an alias that you have already established in NexusDB Server.	
You are now ready to use this data module to connect to tables. For instance, create a new form and drop on a TnxTable. Double-click the Database property to select the database component from the data module you have created. When you do this, and the component has been selected, the session property will be automatically completed. When you click down on the TableName property, a list of all the tables pointed to by the selected alias on the NexusDB Server will be available to select from (or enter a new table name if you are going to create one programmatically).	

ActiveDesigntime	False
ActiveRuntime	False
AliasName	
AliasPath	
DisplayName	nxDatabase1
Enabled	True
EventLog	
EventLogEnabled	False
Exclusive	False
FailSafe	False
Name	nxDatabase1
ReadOnly	False
⊞ Session	nxSession1
Tag	0
Timeout	-1
Version	1.0000

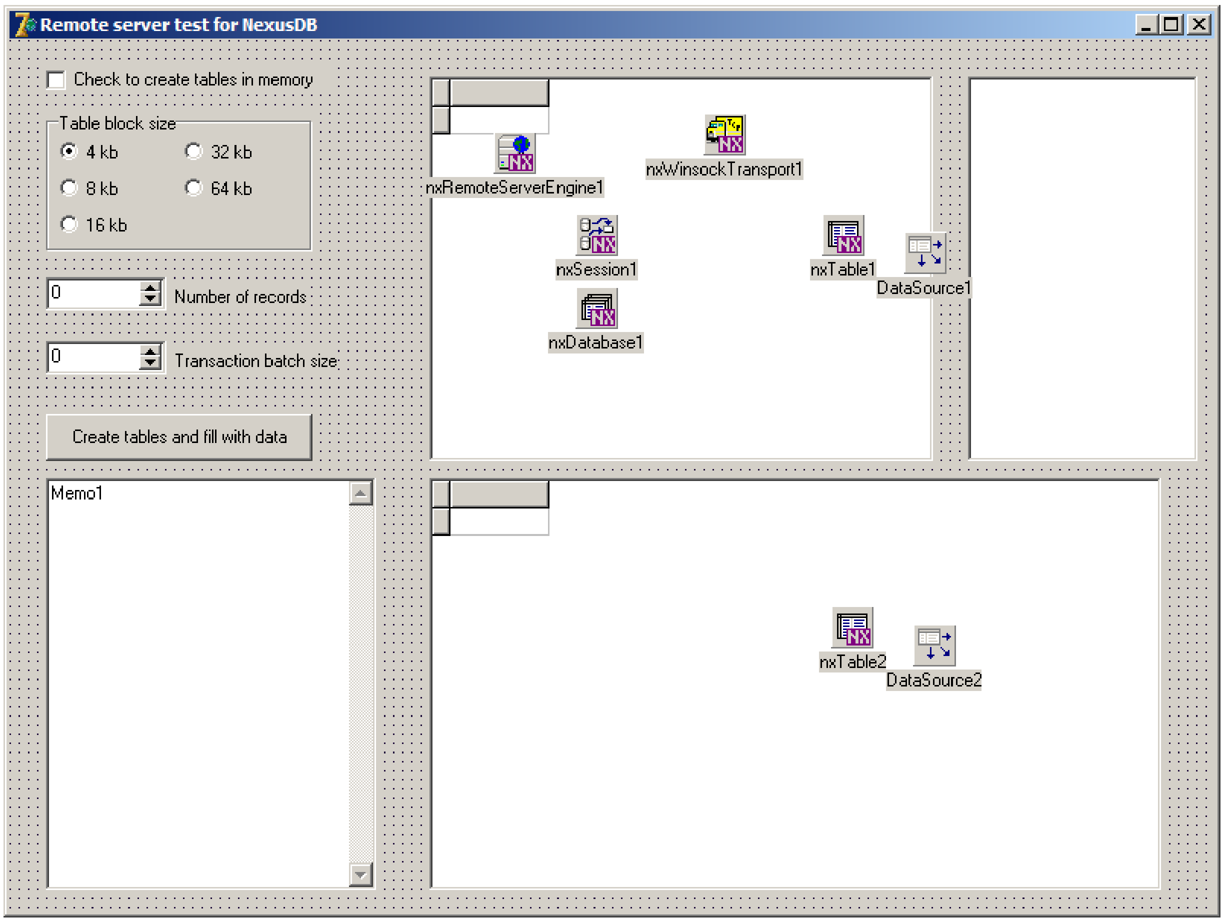
ActiveDesigntime	False
ActiveRuntime	False
AliasName	
⊞ BlockReadOptions	[gboBlobs,gboBookmarks]
⊞ Database	nxDatabase1
Exclusive	False
FieldDefs	(TFieldDefs)
Filter	
Filtered	False
⊞ FilterOptions	[]
FilterTimeout	500
IndexDefs	(TIndexDefs)
IndexFieldNames	
IndexName	
MasterFields	
MasterSource	
Name	nxTable1
ReadOnly	False
⊞ Session	nxSession1
StoreDefs	False
TableName	
Tag	0
Timeout	-1

Table 4: Creating the NexusDB Remote Server Data Module

Well, that's it. You now have a generic data module that you can use in you application to access a NexusDB Server via TCP/IPv4.

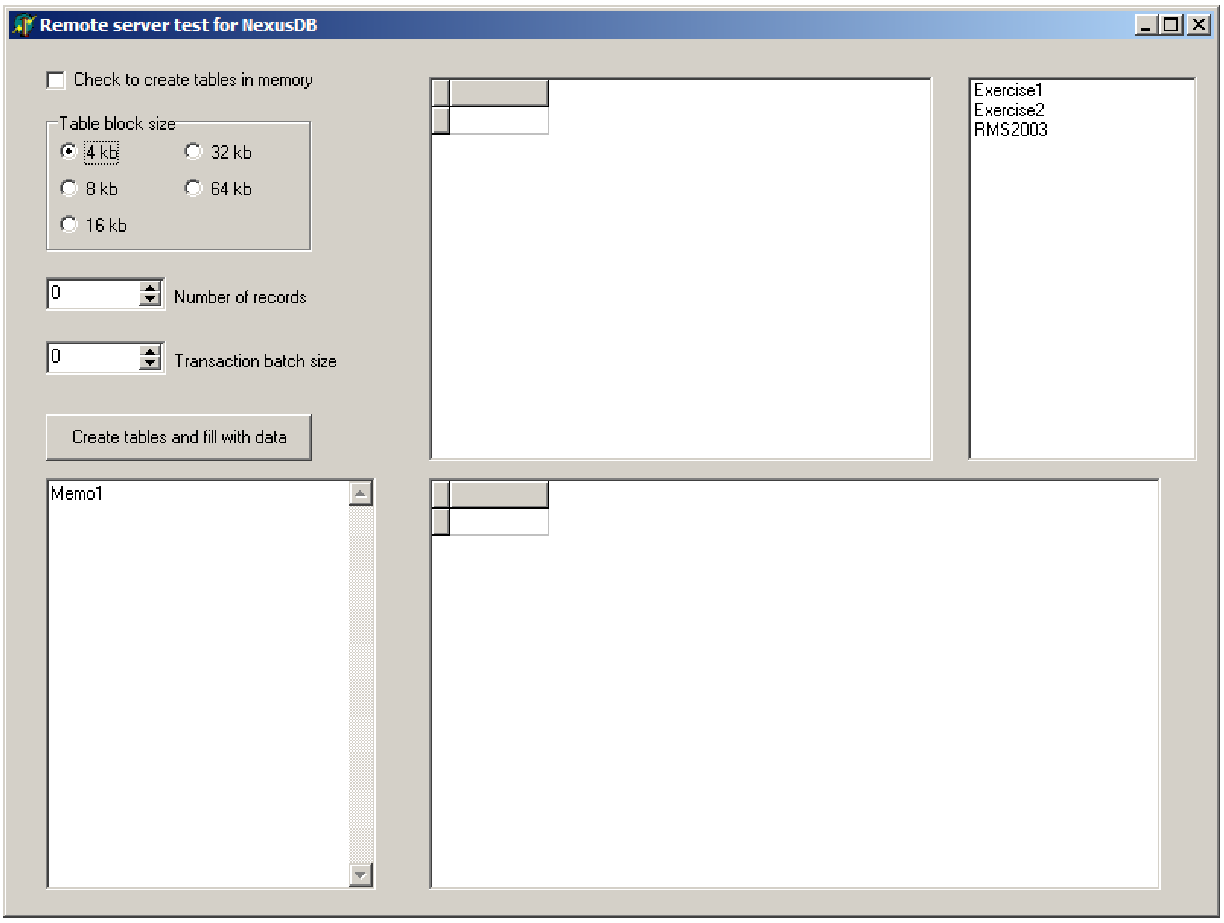
A.5.3 An example application

Make a copy of the previous (SingleExe) example and save it under a different name. Use this as a quick start for this new example. Replace the embedded server engine with a TnxRemoteServer engine component and add a TnxWinsockTransport component. Please link them as shown above and don't forget to link the session to the remote server engine!
Alternatively add a generic C/S server data module to you project and reference its components instead.
Now add a list box to the main form as shown below. You will see that the full server name and TCP/IP address is in the DesignTimeServerName and RunTimeServerName properties. To get these addresses automatically, you need only boot up nxServer, set the Winsock transport active, and then double-click the DesignTime property (after setting ActiveDesigntime to true). Copy and paste the address then into the RunTime property beneath it.



Screenshot 3: A Client/Server Application Example (design)

You can now compile and run the application. At runtime the following screen is displayed. In this example you will see that on startup the Aliases available on the server we are connected to are now displayed in the list box at the top right of the form. Double clicking on any one of these Aliases will direct the 2 created/overwritten tables to be placed with that highlighted Alias.

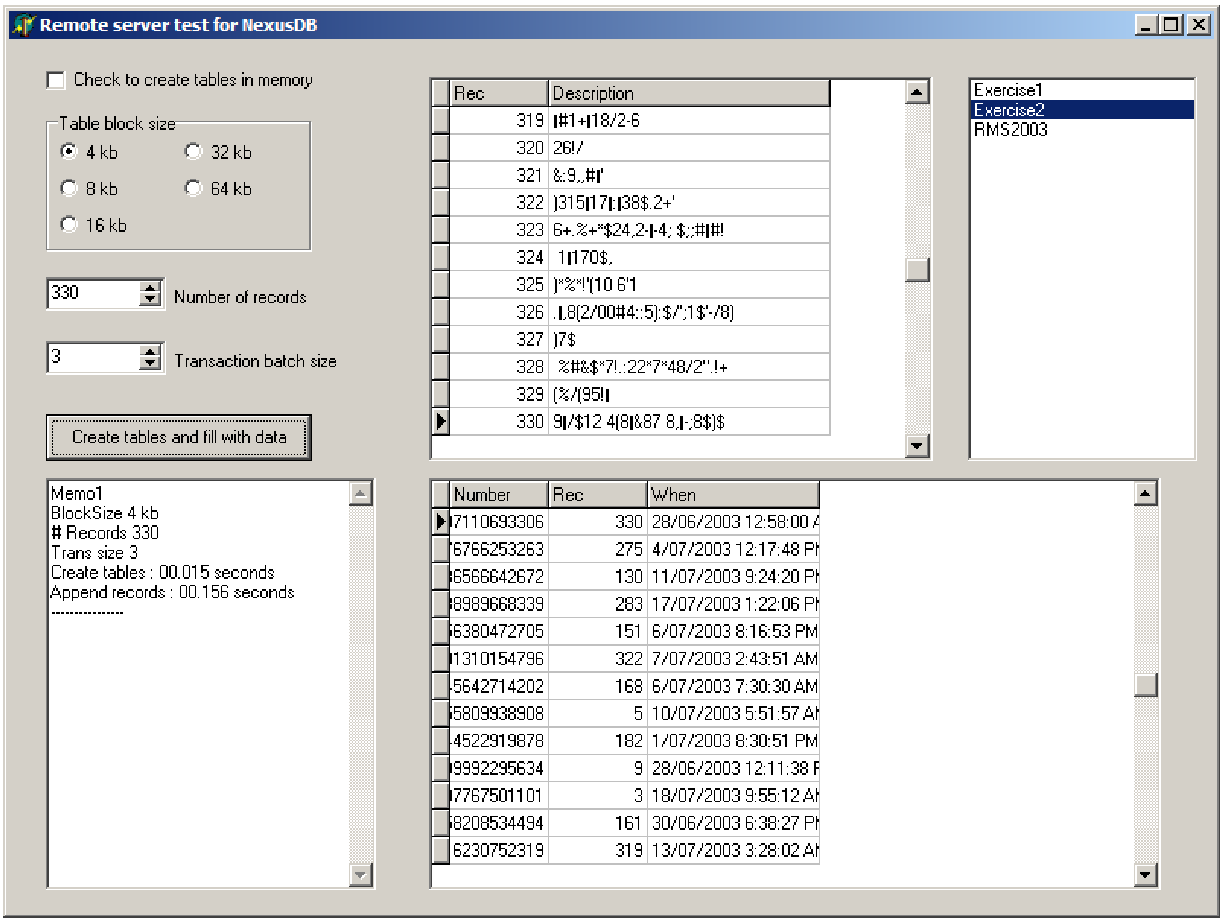


Screenshot 4: A Client/Server Application Example (runtime)

The code necessary to do this is as follows:

```
procedure TMainformDialog.InitCommsEngine;
begin
  ListBox1.Clear;
  try
    nxSession1.Open;
    LoadAliases;
  except
    on E:Exception do
    begin
      MessageDlg(E.Message,mtError,[mbOk],0);
    end;
  end;
end;

procedure TMainformDialog.LoadAliases;
var
  Aliases:TStringList;
begin
  Aliases:=TStringList.Create;
  try
    nxSession1.GetAliasNames(Aliases);
    ListBox1.items.assign(aliases);
  finally
    Aliases.Free;
  end;
end;

procedure TMainformDialog.FormCreate(Sender: TObject);
begin
  InitCommsEngine;
end;

procedure TMainformDialog.ListBox1DblClick(Sender: TObject);
begin
  nxDatabase1.Connected:=False;
  nxDatabase1.AliasName:=ListBox1.Items[ListBox1.ItemIndex];
  nxDatabase1.Connected:=True;
end;
```

After double clicking on Exercise2, we can then build tables and so forth back at the server Alias:


Remote server test for NexusDB
Create tables and fill with data

[ ] Check to create tables in memory

Table block size: [✓ 4 kb] 32 kb 8 kb 64 kb 16 kb

330 Number of records

3 Transaction batch size

Exercise1
Exercise2
RMS2003

```
Memo1
BlockSize 4 kb
# Records 330
Trans size 3
Create tables : 00.015 seconds
Append records : 00.156 seconds
----------------
```

Rec	Description
319	|#1+|18/2-6
320	26!/
321	&:9,,#|'
322	)315|17|:|38$.2+'
323	6+.%+*$24,2-|-4; $;;#|#!
324	1|170$,
325	)*%*!'(10 6'1
326	.|,8(2/00#4::5):$/';1$'-/8)
327	)7$
328	%#&$*7!.:22*7*48/2''.!+
329	(%/(95!|
330	9|/$12 4(8\|&87 8,\|-;8$)$

Number	Rec	When
7110693306	330	28/06/2003 12:58:00 A
6766253263	275	4/07/2003 12:17:48 PI
6566642672	130	11/07/2003 9:24:20 PI
8989668339	283	17/07/2003 1:22:06 PI
6380472705	151	6/07/2003 8:16:53 PM
1310154796	322	7/07/2003 2:43:51 AM
5642714202	168	6/07/2003 7:30:30 AM
5809938908	5	10/07/2003 5:51:57 AI
4522919878	182	1/07/2003 8:30:51 PM
9992295634	9	28/06/2003 12:11:38 F
7767501101	3	18/07/2003 9:55:12 AI
8208534494	161	30/06/2003 6:38:27 PI
6230752319	319	13/07/2003 3:28:02 AI

Screenshot 5: A Client/Server Application Example (with data)

A.6 Mixing Embedded and C/S Applications

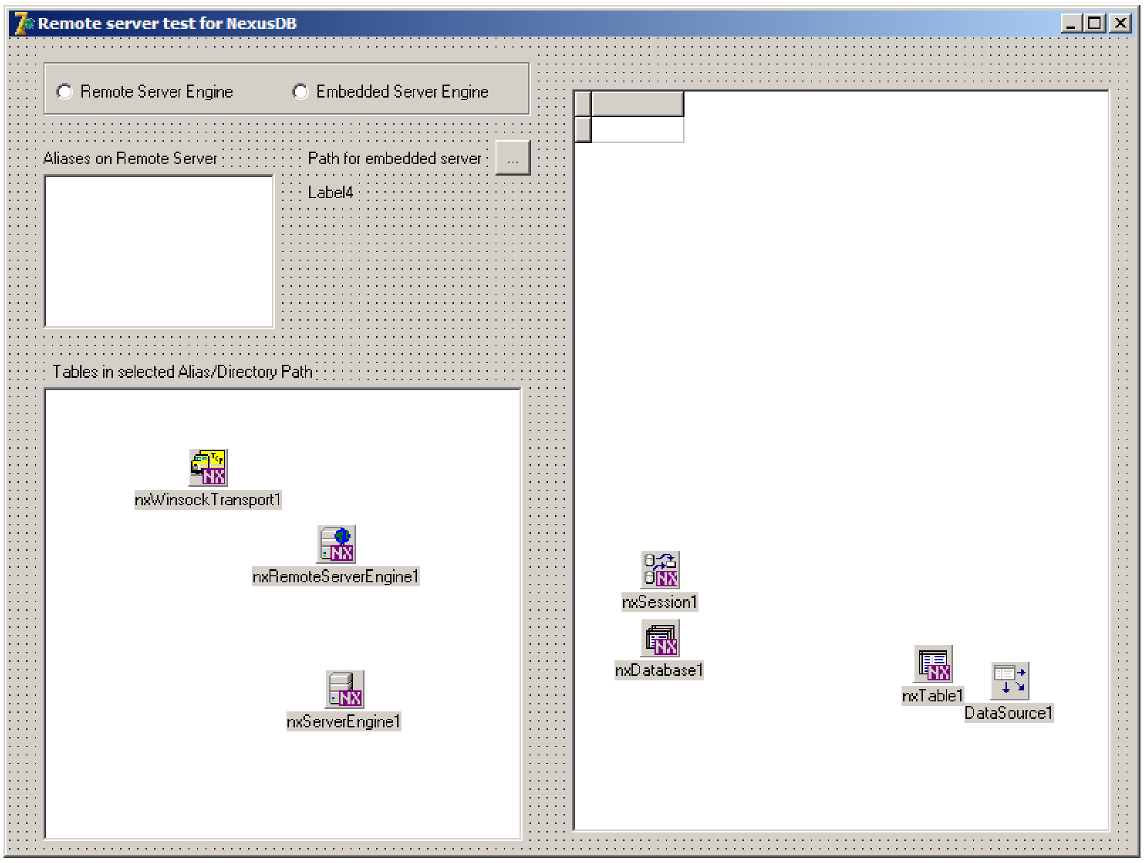
In this section we create an application that holds both embedded server and remote server engines. A TradioGroup is used to switch between tables associated with the 2 different engines.

A.6.1 Before you start Delphi

Start the NexusDB Server on your workstation and ensure that the Winsock transport is started and an appropriate alias has been created. Find more information about the NexusDB Server in section 5.

A.6.2 An example application

By now you are an expert at whipping up NexusDB applications. So create the application form as shown below:



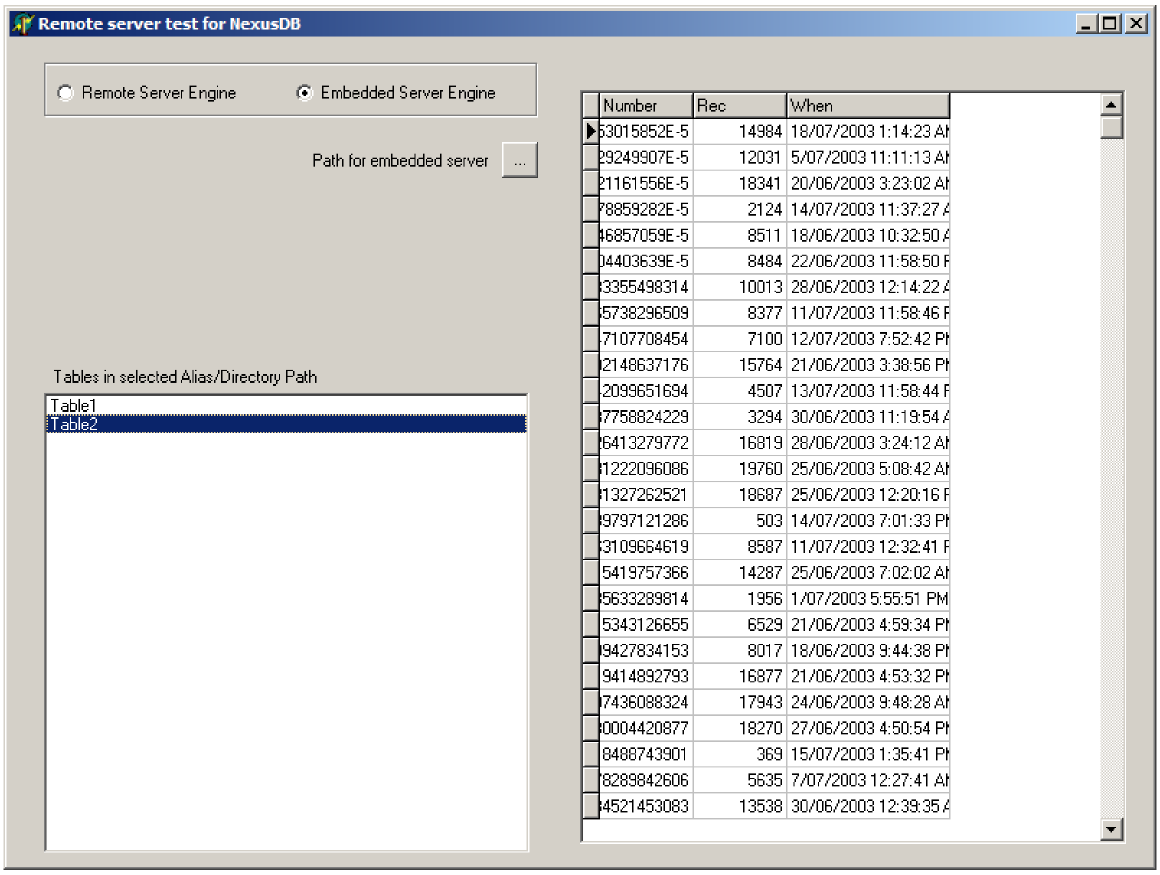
Screenshot 6: A Mixed Mode Example

Please enter the following code. Not that the top ListBox is ListBox1 and the lower (bigger) one is ListBox2. The other components follow easily from their usage in the code.

```
procedure TMainformDialog.InitCommsEngine;
begin
  ListBox1.Clear;
  try
    nxSession1.Open;
    LoadAliases;
  except
    on E:Exception do
    begin
```

```
      MessageDlg(E.Message,mtError,[mbOk],0);
    end;
  end;
end;

procedure TMainformDialog.LoadAliases;

var
  Aliases:TStringList;
begin
  Aliases:=TStringList.Create;
  try
    nxSession1.GetAliasNames(Aliases);
    ListBox1.items.assign(aliases);
  finally
    Aliases.Free;
  end;
end;

procedure TMainformDialog.LoadTables;
var
  Tables:TStringList;
begin
  Tables:=TStringList.Create;
  try
    nxDatabase1.GetTableNames(Tables);
    ListBox2.items.assign(Tables);
  finally
    Tables.Free;
  end;
end;

procedure TMainformDialog.ListBox1DblClick(Sender: TObject);
begin
  nxDatabase1.Connected:=False;
  nxDatabase1.AliasName:=ListBox1.Items[ListBox1.ItemIndex];
  nxDatabase1.Connected:=True;
  LoadTables;
end;

procedure TMainformDialog.RadioGroup1Click(Sender: TObject);
begin
  nxSession1.Connected:=false;
  if RadioGroup1.ItemIndex=0 then
  begin
    Button1.Hide;
    Label3.Hide;
    Label4.Hide;
    nxSession1.ServerEngine:=nxRemoteServerEngine1;
    InitCommsEngine;
    ListBox1.Show;
    Label2.Show;
  end else
  begin
    ListBox1.Hide;
    Label2.Hide;
    nxSession1.ServerEngine:=nxServerEngine1;
    Button1.Show;
    Label3.Show;
    Label4.Hide;
    Label4.Caption:='';
  end;
  nxSession1.Connected:=True;
end;

procedure TMainformDialog.Button1Click(Sender: TObject);
var
  aPath:String;
begin
  aPath:=ExtractFilePath(paramStr(0));
  if SelectDirectory(aPath,[sdAllowCreate,sdPerformCreate,sdPrompt],SELDIRHELP)
then
  begin
    with nxDataBase1 do
    begin
      Connected:=false;
      AliasPath:=aPath;
      Connected:=true;
    end;
    LoadTables;
  end;
end;

procedure TMainformDialog.ListBox2DblClick(Sender: TObject);
begin
  With nxTable1 do
  begin
    Close;
    TableName:=ListBox2.Items[ListBox2.ItemIndex];
    Open;
  end;
end;
```

Once the application starts up, select the mode you want by pressing the correct radio button. Depending on your selection you will end up with either


Remote server test for NexusDB

( ) Remote Server Engine [✓ Embedded Server Engine]

Path for embedded server ...

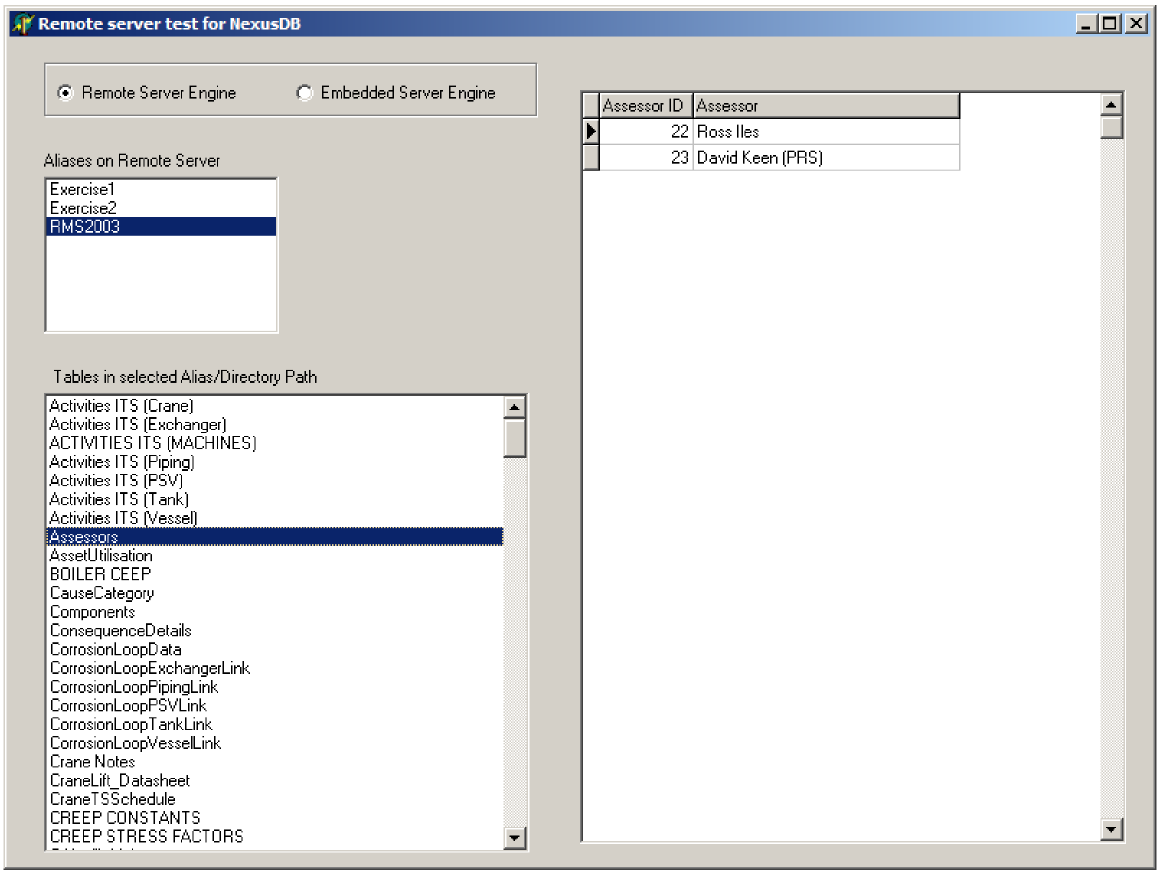
Tables in selected Alias/Directory Path

Table1
Table2

Number	Rec	When
53015852E-5	14984	18/07/2003 1:14:23 A
29249907E-5	12031	5/07/2003 11:11:13 A
21161556E-5	18341	20/06/2003 3:23:02 A
78859282E-5	2124	14/07/2003 11:37:27
46857059E-5	8511	18/06/2003 10:32:50
04403639E-5	8484	22/06/2003 11:58:50
3355498314	10013	28/06/2003 12:14:22
5738296509	8377	11/07/2003 11:58:46
7107708454	7100	12/07/2003 7:52:42 P
2148637176	15764	21/06/2003 3:38:56 P
2099651694	4507	13/07/2003 11:58:44
7758824229	3294	30/06/2003 11:19:54
6413279772	16819	28/06/2003 3:24:12 A
1222096086	19760	25/06/2003 5:08:42 A
1327262521	18687	25/06/2003 12:20:16
9797121286	503	14/07/2003 7:01:33 P
3109664619	8587	11/07/2003 12:32:41
5419757366	14287	25/06/2003 7:02:02 A
5633289814	1956	1/07/2003 5:55:51 PM
5343126655	6529	21/06/2003 4:59:34 P
9427834153	8017	18/06/2003 9:44:38 P
9414892793	16877	21/06/2003 4:53:32 P
7436088324	17943	24/06/2003 9:48:28 A
0004420877	18270	27/06/2003 4:50:54 P
8488743901	369	15/07/2003 1:35:41 P
8289842606	5635	7/07/2003 12:27:41 A
4521453083	13538	30/06/2003 12:39:35

Screenshot 7: A Mixed Mode Example (running - embedded)

or


Remote server test for NexusDB

[✓ Remote Server Engine] ( ) Embedded Server Engine

Aliases on Remote Server

Exercise1
Exercise2
RMS2003

Tables in selected Alias/Directory Path

Activities ITS (Crane)
Activities ITS (Exchanger)
ACTIVITIES ITS (MACHINES)
Activities ITS (Piping)
Activities ITS (PSV)
Activities ITS (Tank)
Activities ITS (Vessel)
Assessors
AssetUtilisation
BOILER CEEP
CauseCategory
Components
ConsequenceDetails
CorrosionLoopData
CorrosionLoopExchangerLink
CorrosionLoopPipingLink
CorrosionLoopPSVLink
CorrosionLoopTankLink
CorrosionLoopVesselLink
Crane Notes
CraneLift_Datasheet
CraneTSSchedule
CREEP CONSTANTS
CREEP STRESS FACTORS

Assessor ID	Assessor
22	Ross Iles
23	David Keen (PRS)

Screenshot 8: A Mixed Mode Example (running – c/s)

The technique shown above gives you full control over where to store data. You can for example cache data on client side to avoid slow lookups over the network or you can divide the data into private and public parts. A good use of this is to copy lookup tables from a remote server into a local server engine (and usually as in-memory tables for extra speed) on the assumption that the data in the lookup tables change rarely if at all.

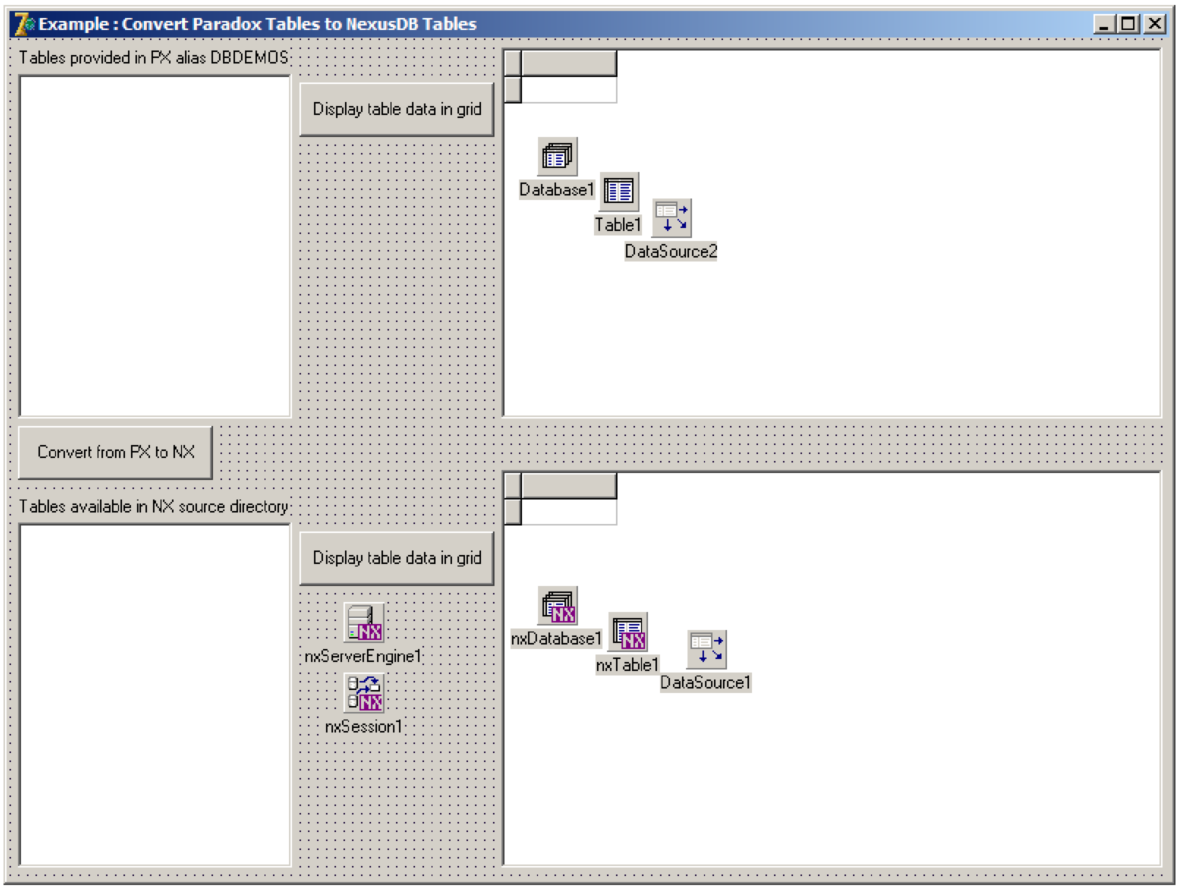
A.7 Some Simple Projects to Get Started

In this section we present some example projects, in Delphi 7, which illustrate some of the more commonly used features of a DBMS. Each of these projects are intended to illustrate a straightforward way of implementing a common task. There are many ways to do each of these (for instance, with table creation we use the standard TDef's approach rather than the TnxDataDictionary object) and the way chosen is usually for familiarity with the BDE way. Naturally there are also a number of projects that illustrate NexusDB specific features such as Memory Tables and the Block Mode operations.
The projects illustrated are

1. Simple application to convert Paradox tables to NexusDB tables
2. Illustrative example of creation and display of Memory Tables
3. Moving batches of records between the server and client efficiently with Block Mode

A.7.1 Converting Paradox tables to NexusDB tables

This small project is designed to give you a quick method for converting Paradox tables to NexusDB tables. This will be useful if you need to do some special processing during conversion (eg, delete or process records on during the conversion process) which you wish to keep separate from the Converter Utility that comes with the full NexusDB install system.
Create a new project with a single form as shown below. Note that the DB components at the top are all from the BDE tab of the VCL. Connect up the components : the NexusDB components as per Section 6 above and the BDE components as per the Delphi help (if you need a reference).


Example : Convert Paradox Tables to NexusDB Tables
Tables provided in PX alias DBDEMOS
Display table data in grid
Database1
Table1
DataSource2
Convert from PX to NX
Tables available in NX source directory
Display table data in grid
nxServerEngine1
nxSession1
nxDatabase1
nxTable1
DataSource1

Screenshot 9: Convert Paradox to Nexus Example (design)

Connect up the following code to the appropriate button click events:

Ancillary Code:

```
procedure TMainformDialog.pxLoadTables;
var
  Tables:TStringList;
begin
  Tables:=TStringList.Create;
  try
    Database1.GetTableNames(Tables);
    ListBox2.Items.Assign(Tables);
  finally
    Tables.Free;
  end;
end;

procedure TMainformDialog.nxLoadTables;
var
  Tables:TStringList;
begin
  Tables:=TStringList.Create;
  try
    nxDatabase1.GetTableNames(Tables);
    ListBox1.Items.Assign(Tables);
  finally
    Tables.Free;
  end;
end;

procedure TMainformDialog.FormCreate(Sender: TObject);
begin
  DataBase1.Connected:=True;
  nxDataBase1.AliasPath:=ExtractFilePath(ParamStr(0));
  nxDataBase1.Connected:=True;
  pxLoadTables;
  nxLoadTables;
end;
```

Top Button (Display PX):

```
procedure TMainformDialog.Button4Click(Sender: TObject);
begin
  If ListBox2.ItemIndex<0 then ShowMessage('No Table Selected') else
  begin
    Table1.Close;
    Table1.TableName:=ListBox2.Items[ListBox2.ItemIndex]+'.DB';
    if Table1.Exists then Table1.Open else ShowMessage('Sorry - that is not a
Paradox table!');
  end;
```

Middle Button (Convert):

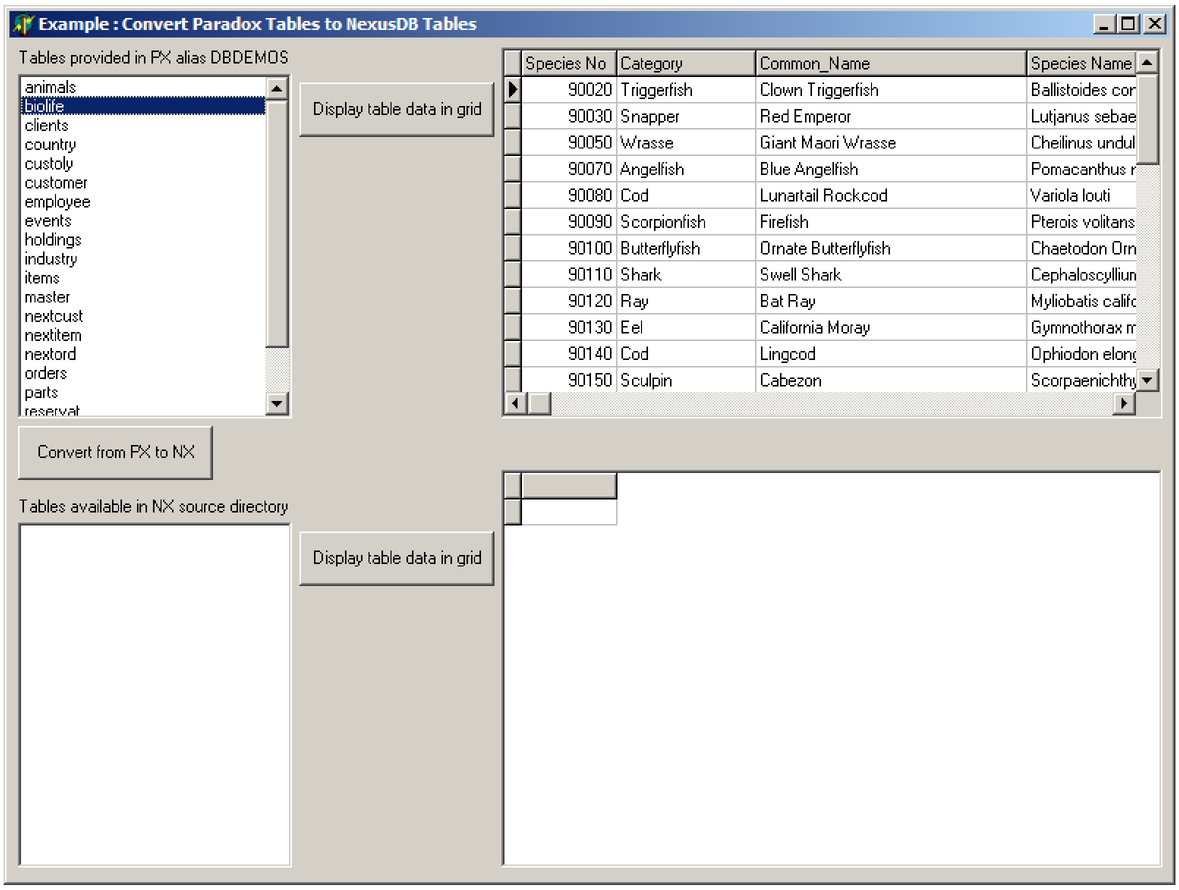
```
procedure TMainformDialog.Button3Click(Sender: TObject);
var
```

```
  i:integer;
  ok:Boolean;
begin
  If Not Table1.Active then ShowMessage('No table is open') else
  begin
    with nxTable1 do
    begin
      Close;
      FieldDefs.Clear;
      FieldDefs.Assign(Table1.FieldDefs);
      IndexDefs.Clear;
      IndexDefs.Assign(Table1.IndexDefs);

TableName:=System.Copy(Table1.TableName,1,System.Pos('.DB',Table1.TableName)-1);
      if Exists then OK:=MessageDlg('Table already exists.
Replace?',mtConfirmation,[mbYes,mbNo],0)=mrYes else OK:=True;
      if ok then
      begin
        CreateTable;
        Open;
        with Table1 do
        begin
          first;
          while not eof do
          begin
            nxTable1.Insert;
            for i:=0 to Pred(FieldCount) do
            nxTable1.Fields[i].Assign(Fields[i]);
            nxTable1.Post;
            next;
          end;
        end;
      end;
    end;
    nxLoadTables;
  end;
```

Bottom Button (Display NX):

```
procedure TMainformDialog.Button2Click(Sender: TObject);
begin
  If ListBox1.ItemIndex<0 then ShowMessage('No Table Selected') else
  begin
    nxTable1.Close;
    nxTable1.TableName:=ListBox1.Items[ListBox1.ItemIndex];
    nxTable1.Open;
  end;
```

When you compile and run this button and then double-click, for example, the biolife table, you should see the following on your screen:



Example : Convert Paradox Tables to NexusDB Tables

Tables provided in PX alias DBDEMOS

animals
biolife
clients
country
custoly
customer
employee
events
holdings
industry
items
master
nextcust
nextitem
nextord
orders
parts
reservat

Display table data in grid

Species No	Category	Common_Name	Species Name
90020	Triggerfish	Clown Triggerfish	Ballistoides cor
90030	Snapper	Red Emperor	Lutjanus sebae
90050	Wrasse	Giant Maori Wrasse	Cheilinus undul
90070	Angelfish	Blue Angelfish	Pomacanthus r
90080	Cod	Lunartail Rockcod	Variola louti
90090	Scorpionfish	Firefish	Pterois volitans
90100	Butterflyfish	Ornate Butterflyfish	Chaetodon Orn
90110	Shark	Swell Shark	Cephaloscyllium
90120	Ray	Bat Ray	Myliobatis califo
90130	Eel	California Moray	Gymnothorax m
90140	Cod	Lingcod	Ophiodon elong
90150	Sculpin	Cabezon	Scorpaenichthy

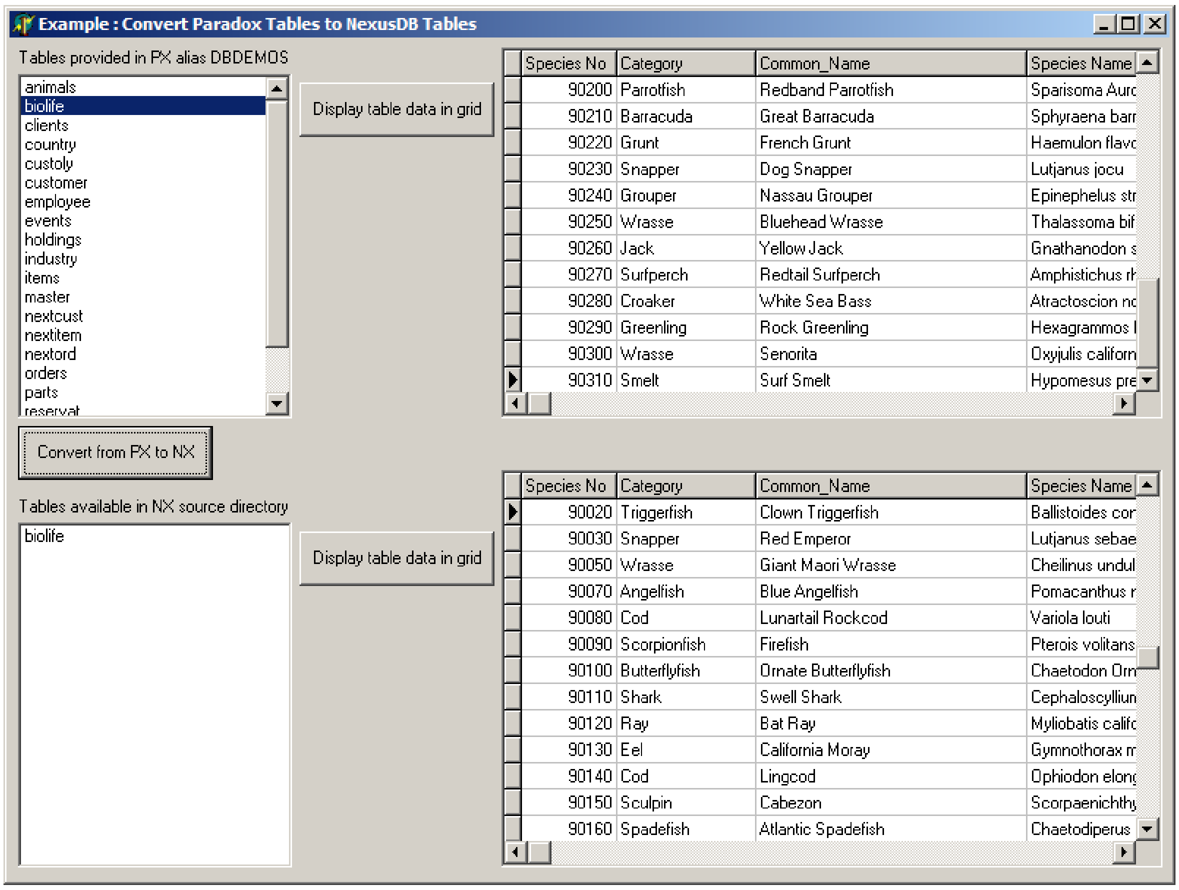
Convert from PX to NX

Tables available in NX source directory

Display table data in grid

Screenshot 10: Convert Paradox to Nexus Example (runtime)

If you now click the Convert button the following screen will be displayed.


Example : Convert Paradox Tables to NexusDB Tables
Display table data in grid
Convert from PX to NX
Display table data in grid

Tables provided in PX alias DBDEMOS

animals
biolife
clients
country
custoly
customer
employee
events
holdings
industry
items
master
nextcust
nextitem
nextord
orders
parts
reservat

Species No	Category	Common_Name	Species Name
90200	Parrotfish	Redband Parrotfish	Sparisoma Aurc
90210	Barracuda	Great Barracuda	Sphyraena barr
90220	Grunt	French Grunt	Haemulon flavc
90230	Snapper	Dog Snapper	Lutjanus jocu
90240	Grouper	Nassau Grouper	Epinephelus str
90250	Wrasse	Bluehead Wrasse	Thalassoma bif
90260	Jack	Yellow Jack	Gnathanodon s
90270	Surfperch	Redtail Surfperch	Amphistichus rh
90280	Croaker	White Sea Bass	Atractoscion nc
90290	Greenling	Rock Greenling	Hexagrammos l
90300	Wrasse	Senorita	Oxyjulis californ
90310	Smelt	Surf Smelt	Hypomesus pre

Tables available in NX source directory

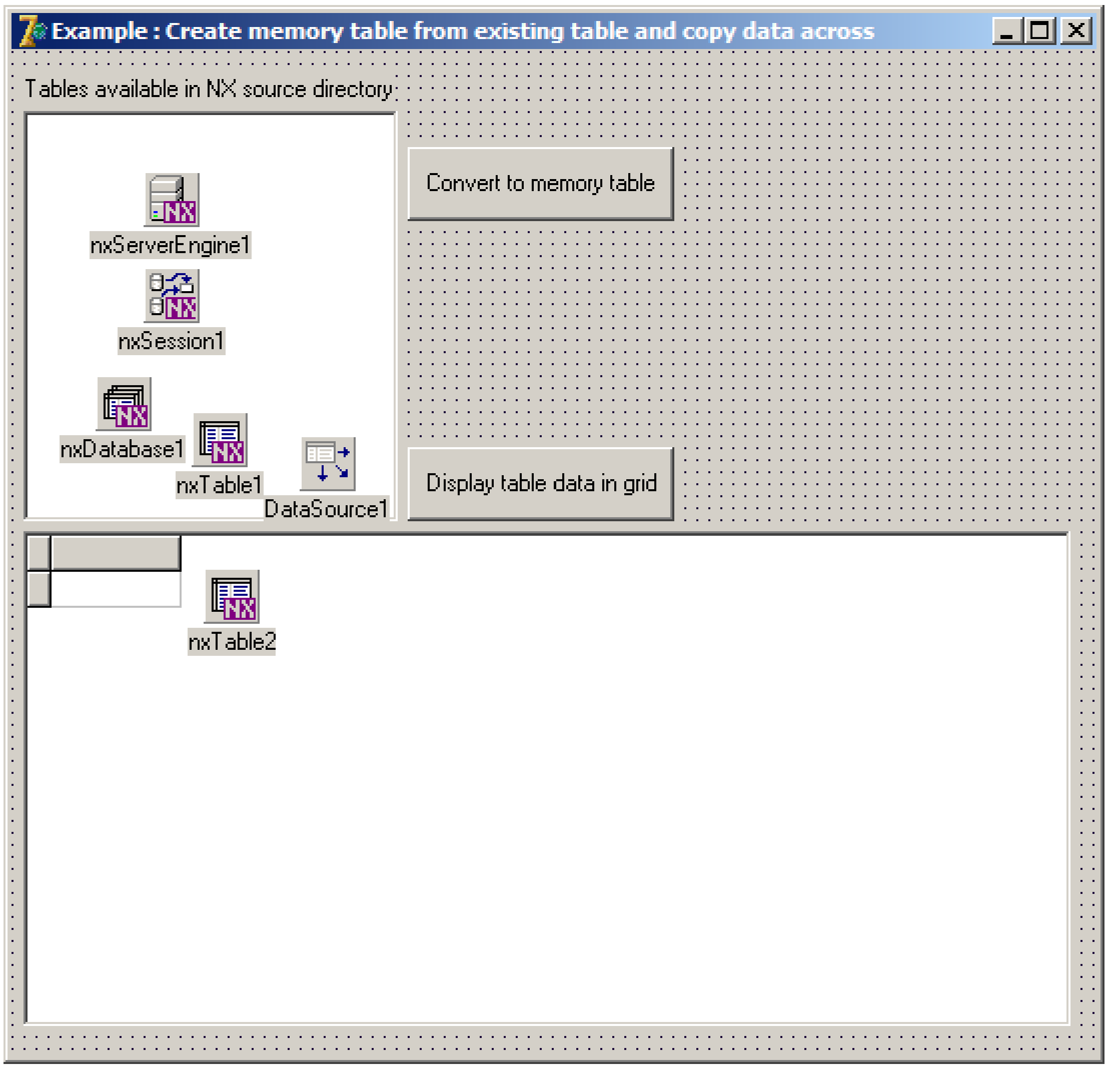
biolife

Species No	Category	Common_Name	Species Name
90020	Triggerfish	Clown Triggerfish	Ballistoides cor
90030	Snapper	Red Emperor	Lutjanus sebae
90050	Wrasse	Giant Maori Wrasse	Cheilinus undul
90070	Angelfish	Blue Angelfish	Pomacanthus r
90080	Cod	Lunartail Rockcod	Variola louti
90090	Scorpionfish	Firefish	Pterois volitans
90100	Butterflyfish	Ornate Butterflyfish	Chaetodon Orn
90110	Shark	Swell Shark	Cephaloscylliun
90120	Ray	Bat Ray	Myliobatis califc
90130	Eel	California Moray	Gymnothorax m
90140	Cod	Lingcod	Ophiodon elong
90150	Sculpin	Cabezon	Scorpaenichthy
90160	Spadefish	Atlantic Spadefish	Chaetodiperus

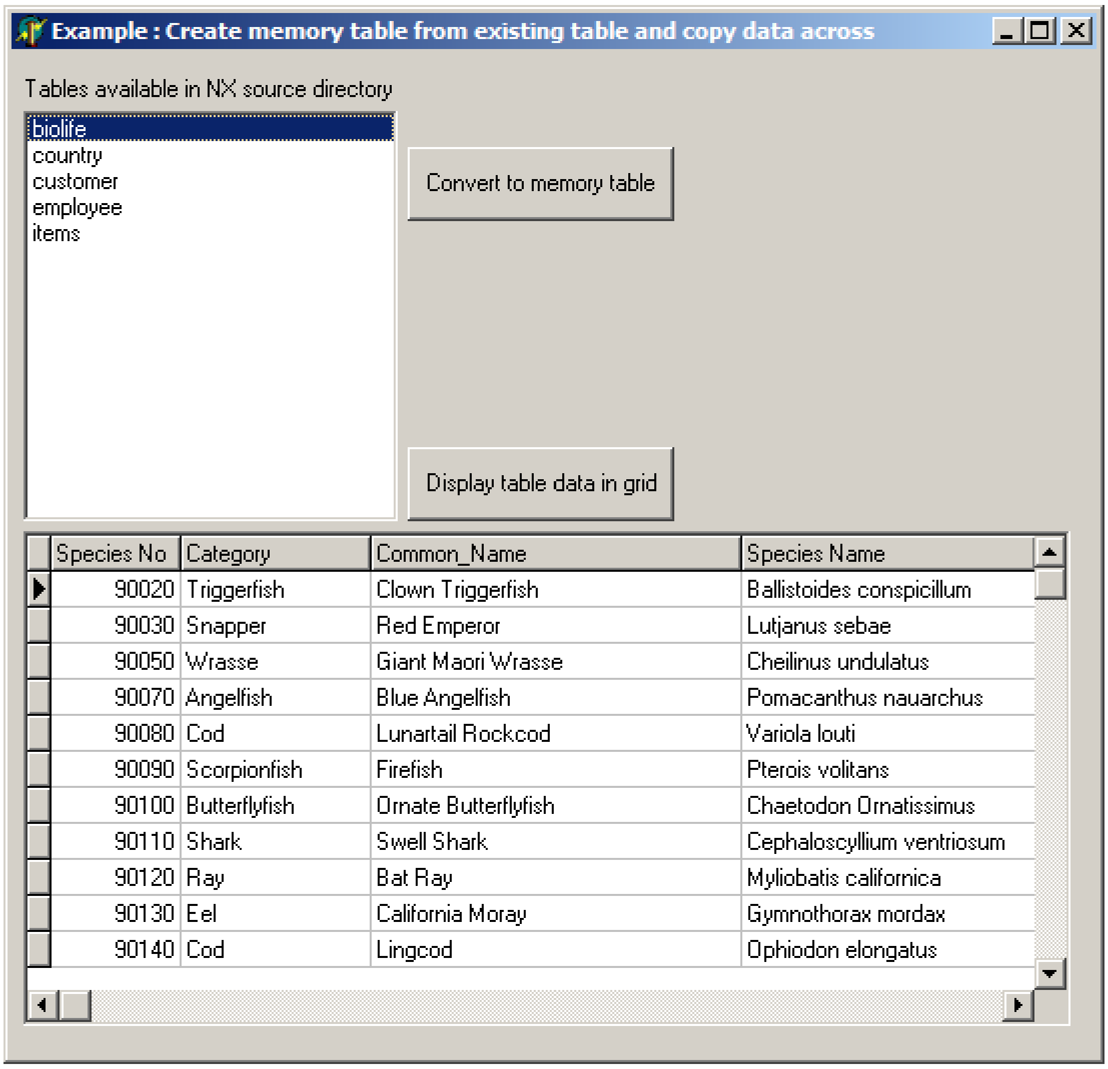
Screenshot 11: Convert Paradox to Nexus Example (runtime 2)

This example has shown you, in a very quick way, how to read all the tables from a particular directory (or alias) into a list box as well as how to create a NexusDB copy of an existing Paradox table, including transferring the data across.

A.7.2 Introducing Memory Tables

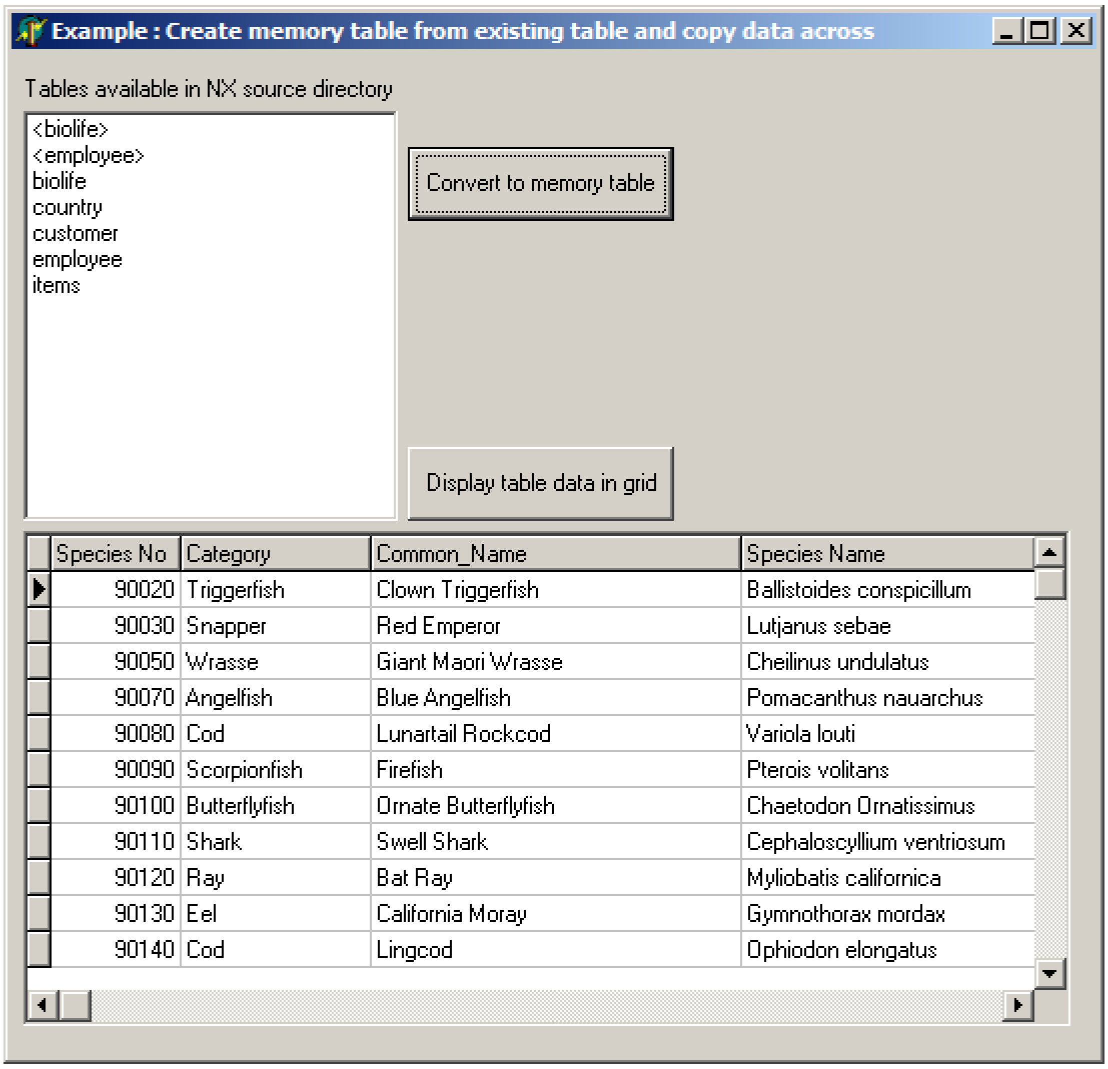

Example : Create memory table from existing table and copy data across
Tables available in NX source directory
nxServerEngine1
nxSession1
nxDatabase1
nxTable1
DataSource1
Convert to memory table
Display table data in grid
nxTable2

Screenshot 12: In-Memory table example (design)


Example : Create memory table from existing table and copy data across
Tables available in NX source directory
biolife
country
customer
employee
items
Convert to memory table
Display table data in grid

Species No	Category	Common_Name	Species Name
90020	Triggerfish	Clown Triggerfish	Ballistoides conspicillum
90030	Snapper	Red Emperor	Lutjanus sebae
90050	Wrasse	Giant Maori Wrasse	Cheilinus undulatus
90070	Angelfish	Blue Angelfish	Pomacanthus nauarchus
90080	Cod	Lunartail Rockcod	Variola louti
90090	Scorpionfish	Firefish	Pterois volitans
90100	Butterflyfish	Ornate Butterflyfish	Chaetodon Ornatissimus
90110	Shark	Swell Shark	Cephaloscyllium ventriosum
90120	Ray	Bat Ray	Myliobatis californica
90130	Eel	California Moray	Gymnothorax mordax
90140	Cod	Lingcod	Ophiodon elongatus

Screenshot 13: In-Memory table example (runtime)


Example : Create memory table from existing table and copy data across
Tables available in NX source directory
Convert to memory table
Display table data in grid

<biolife>
<employee>
biolife
country
customer
employee
items

Species No	Category	Common_Name	Species Name
90020	Triggerfish	Clown Triggerfish	Ballistoides conspicillum
90030	Snapper	Red Emperor	Lutjanus sebae
90050	Wrasse	Giant Maori Wrasse	Cheilinus undulatus
90070	Angelfish	Blue Angelfish	Pomacanthus nauarchus
90080	Cod	Lunartail Rockcod	Variola louti
90090	Scorpionfish	Firefish	Pterois volitans
90100	Butterflyfish	Ornate Butterflyfish	Chaetodon Ornatissimus
90110	Shark	Swell Shark	Cephaloscyllium ventriosum
90120	Ray	Bat Ray	Myliobatis californica
90130	Eel	California Moray	Gymnothorax mordax
90140	Cod	Lingcod	Ophiodon elongatus

Screenshot 14: In-memory table (runtime 2)

A.7.3 Illustrating the Block Mode batch operations

A.8 Upgrading existing FF2 projects to NexusDB

This section describes for developers a fairly quick way of upgrading existing FF2 applications to NexusDB. The steps involved, at a high level, are:

1. Copy and convert FF2 tables to NexusDB tables in a new directory (alias) using the convertor utility.
2. Replace and update FF components in your application to NexusDB components.
3. If you used TffDataDictionary, then you may have to convert some of your source code.

Let us now go through the conversion steps in detail.

A.8.1 Step 1 – Database conversion

This is the easiest step. Simply run the nxImporter.exe utility. Don't forget to first create an output alias to pump the new tables and data into.

A.8.2 Step 2 – Component substitution

The following table quickly summarizes the component substitutions that you have to make. Note that the TffClient(TffCommsEngien) and TffSession components are replaced by a single TnxSession component.

Flashfiler	NexusDB
TffClient, TffCommsEngine	Remove; merged into TnxSession
TffSession	TnxSession
TffDatabase	TnxDatabase
TffTable	TnxTable
TffQuery	TnxQuery
TffServerEngine	TnxServerEngine
TffRemoteServerEngine	TnxRemoteServerEngine
TffSQLEngine	TnxSQLEngine
TffServerCommandHandler	TnxCserverCommandHandler
TffLegacyTransport	TnxWinsockTransport or TnxNamedPipeTransport
TffEventLog	TnxEventLog
TffSecurityMonitor	TnxSecurityMonitor
TffThreadPool	Remove; not available

Table 5: FlashFiler -> NexusDB component substitution

A.8.3 Step 3 – Source code changes

Most of the source code changes that you will need to make are related to the data dictionary object. Another change you need to be aware of is that the SessionName and DatabaseName properties no longer exist. Similarly, there is no longer a Client component at all.
For most of the other changes, we provide a simple automatic conversion utility for renaming all the FF2 related names and properties to the NexusDB equivalent. The utility is called Convert_Src2Nx.dpr and is located in the Convert subdirectory of you NexusDB installation. Please note that you almost certainly still have to some manual changes to make the application compile. At the moment the best advice is to run the code converter tool and then fix up the other issues by the old "compile and go to next syntax error" technique!

Finally, you will have a clean compile and go! You should now have converted your FF2 application to NexusDB. As with all such tasks, the first project is the hardest – you should find subsequent applications easier to convert.

A.9 Differences between FF and NexusDB

In this section is detailed some of the differences, enhancements and extensions, based on a broad comparison between FlashFiler and NexusDB. Even though NexusDB builds on many of the same principles as FlashFiler, and has a similar programming API, NexusDB is a complete rewrite. There is no FlashFiler source in the NexusDB code base.

A.9.1 The Memory Manager

This is completely new. It is better for NexusDB than Delphi's memory manager. The Delphi memory manager suffers from memory fragmentation and uses about 30% more memory. The new memory manager can be used in non-NexusDB projects as well. The memory manager is optimized for many small allocations (<64kb), everything larger goes directly to the OS (VirtualAlloc).
To use the memory manager you simply have to add "nxReplacementMemoryManager," as the very first line of the "uses …;" clause in your project source (.dpr file).

A.9.2 The Buffer Manager

The buffer manager is responsible for all reads and writes to/from the hard disk. It does this via a new OO (and extensible) file access layer. All transaction management is performed here. As a consequence a new form of transaction has been developed and implemented: snapshot transactions. A snapshot transaction is a read only transaction that provides a consistent view of the entire database at the instant in time the transaction was started, without placing any locks on the system. This allows other clients to freely modify the database without affecting the integrity of the system from the viewpoint of the client who started the read only transaction. Snapshot transactions are deletion safe.
In FF2 all reads and writes were done inside a single lock. In NexusDB all write operations have been moved outside this lock. Thus a transaction can be committed and even while it is still writing to disk, other clients can access the data from the committed transaction and even start another transaction. There can even be a situation where multiple transactions are being committed at the same time provided they don't modify the same tables. At all times the data is safe and the integrity of the database assured to a level way above that of FF2.
The buffer manager requires no more than half the memory used by FF2 on newly inserted blocks. The reuse of pages is more efficient. It obeys the max RAM setting even if you have in-memory tables and large transactions, provided read-write temporary storage is available.

A.9.3 The Journal Engine

Completely rewritten and now has 3 settings compared with just one in FF2:

- Mode 1 – before and after images are stored. If a reboot finds these images then a dialog is displayed and the operator asked whether to commit or rollback the changes found.
- Mode 2 – stores before images only. If a reboot finds these images an automatic rollback is performed. This can be thought of as a conservative, or cautious, mode of operation.
- Mode 3 – stores after images only. If a reboot finds these types of images an automatic commit is performed.

Mode 2 and 3 require no user interaction. This makes them perfect if the server is running as a service.

A.9.4 Data Dictionary for a Table

This has been redesigned and uses a fundamentally different architecture. For example, block sizes are now a set of types and not an integer value. The best way to get to grips with it is to read the help file.
It is now completely OO and extensible. You can create your own descriptor objects. Every item in the data dictionary has a string list attached so that you can store extra information for use later by your program. This allows you to select the different sub-engines for use by the table.

A.9.5 The Stream Engine

FF2 could only save a single stream in a table – the data dictionary. In NexusDB an unlimited number of named streams can be stored in a table. Think of this as table specific Blobs. There are four functions to operate on this stream engine – list all streams; read a stream; write a stream; and delete a stream.
Using named streams, additional custom information can be stored in a table besides the records and indices. For example, when the backup component creates a copy of a table, the original data dictionary is stored as a named stream. No indices are added to the copy, just the raw records are copied. When restoring, the indices are recreated based on the information stored in the 2nd stream.

A.9.6 The Record Engine

This manages the records in the pages. A significant improvement over FF2 is the way NexusDB reuses deleted records: NexusDB is faster to reuse a mass of them – it's highly optimized. FF2 uses a linear linked list. NexusDB uses a linked list of blocks and, internal to each block, a linked list. Thus NexusDB will fill block-wise whereas FF2 fills essentially at random.
Currently the record size is fixed, but we also plan to implement a variable length record engine. Because of the extensible architecture of NexusDB it will be possible to use both record engines at the same time. That is, you can mix tables with different record engines in the same database.

A.9.7 The Key Engine

The complete process of key creation and comparison is in a different (new) engine – FF2 does not have this. There are 2 default key engines:

4. RefNr (sequential access), and,
5. Default composite engine (much like FF2)

The composite engine has one composite key field engine per field. This is comparable to the index helper objects in FF2. ASC/DESC can be set on a per field basis. Similarly with locale and string sort/word sort. Within limits this can be accessed with the VCL. It is straightforward to write your own key field engine or even your own complete key engine to, e.g., implement a SoundEx Sort.

A.9.8 The Index Engine

FF2 used a b-Tree index. The new default index implementation is a modified b*-Tree (hybrid) – the modification is for speed. It is more compact. The modular architecture allows you to write additional index engines and assign them on a per index basis. e.g., hash tables or even a sequential sort.

A.9.9 The BLOB Engine

It is now possible to use different BLOB engines for each table. The new (default) BLOB engine has 3 layers: an intra-block heap manager, the management layer, and the interface layer. It is highly efficient, can deal with huge objects, avoids fragmentation and has a very high speed. Overall it's a big improvement over FF2.

A.9.10 Command Handler

Works much the same as in FF2 but much more efficiently. Less messages and they are faster in what they do. There is a "look-ahead" capability built in to reduce message counts and thus speed things up significantly over slow data connections.

A.9.11 Transports

Basic structure has been kept similar to FF by request from FF developers. Two new classes – Winsock TCP over IP4 and Named Pipes. These 2 both support broadcasts and compression. There is an extensible compression architecture with 3 different default compression algorithms already implemented. On NT-based systems these transports use, if possible, only as many threads as the number of CPU's in the system. (Blocking sockets, asynchronous I/O and I/O Completion Ports are used to achieve this.)
Winsock can be used on Win 95+ clients and servers. If the server is not running on an NT-based system it will require one thread per connection as I/O Completion Ports are not available on these systems. If you want to use TCP as transport but have no network card installed in your machine you can simply add a dialup adapter to your configuration. This will install the Winsock dlls.
Named Pipes can be used on Win95+ clients and NT-based servers.
Both support call-backs (server -> client messaging) but with Named Pipes this only works if the client is NT-based.

A.9.12 Client side Components

NexusDB has no TffClient equivalent, the functionality is now merged in with the TnxSession component. There are no more [automatic] components. The database, session and table/query components link directly by component instead of by SessionName etc, as in FF2.
From the client side you also have access to snapshot transactions and nested transactions. Just remember that for nested transactions you call STARTTRANSACTION multiple times and thus have to call either COMMIT or ROLLBACK the same number of times.

A.9.13 New Batch Modes

You can set a BlockReadSize to be the minimum number of bytes for each batch operation. This includes optional support for Blobs and Bookmarks to reduce network traffic – just set an option. Once the BlockReadSize is set to greater than zero byes, you are automatically positioned on the first record. That is, it does a Table.First for you. Thereafter you can only do Table.Next and Table.First to navigate through the records. Use Table.EOF as per normal. To turn this off, just set BlockReadSize back to zero. Note that BlockReadSize in the BDE is the number of records. In NexusDB it means the number of bytes because we have Blobs as well with variable sizes.
To use Batch Writes/Appends look at the following structure:

```
BeginBatchAppend;   // to start
Repeat
APPEND:
// set field values including blobs
BATCHPOST;  // instead of post
Until {batch complete condition};
ENDBATCHAPPEND;    // to post batch off
```

The last statement turns batch appending off.

A.9.14 In-memory and Tables

To use an in-memory table just use <> around the table name. E.g.,

```
Table.TableName:='<MemTable1>';
```

In-memory tables live for as long as the server (or embedded server in the client app if created on the client side) is up.

A.9.15 Other Stuff

To avoid the problems you could run into in FlashFiler if you left an Active property set to True at designtime, NexusDB has ActiveDesignTime and ActiveRunTime properties. They are available in all stateful components including:

- Transports
- Server Engines
- Command Handlers
- Plug-ins
- Extenders
- Sessions
- Databases
- Tables/Queries

Furthermore, Transports also have ServerNameDesignTime and ServerNameRunTime as visible properties, for the same reason. With this enhancement over FlashFiler, you need never worry about resetting your development settings before compiling your application.

Plug-in support is similar to FF2. Extenders have been redesigned with a slightly different architecture. Existing FF2 extenders will have to be modified for NexusDB.

A.10 Transactions and Deadlocks in NexusDB

This section describes the way in which database transactions are implemented in NexusDB. Any such discussion must embrace a large terminology. To ensure consistency in terminology, we begin with a short section, independent of NexusDB, of database transactions and deadlocks. The importance of locks (record, table and database) are also considered in the context of not only ensuring the most efficient use of transactions, but also to help minimize the occurrence of deadlock situations. Its important to mention the different types of locks available and how they are granted by the database engine:

A.10.1 Content locks

Content locks can be either record-level (these are always write locks, acquired by calling Edit) or table-level (these can be either read or write locks, acquired by calling LockTable).
Content locks can be contending for the same resource. If so, then an existing record-level content lock can prevent table-level content locks from being granted. Similarly a table-level content lock will prevent a record-level content lock from being granted.

A.10.2 Transaction locks

Transaction locks are independent from content locks. A shared transaction lock is acquired when you read from a table in the context of a transaction. An exclusive transaction lock is acquired when you write to a table in the context of a transaction.

A.10.3 How does transaction locking work?

Each server side table object has 2 different synchronization objects that manage access to the table:

A.10.3.1 TnxLockContainer

which handles transaction locking (shared/exclusive locks). This lock container is only used when the table is accessed in the context of a transaction. Read access needs a shared lock. Write access requires an exclusive lock. At any point in time there can be either no locks, a single or multiple shared locks or a single exclusive lock. No other combination is possible.

A.10.3.2 TnxReadWritePortal

which handles thread synchronization. Each thread that wants to read from the table outside of a transaction must acquire a read lock on this portal. (This read lock is always released before a server call returns to the client. It is only acquired for the split second it takes to perform the actual read of a single record). A thread that wants to commit a transaction which has acquired an exclusive lock on the table must acquire a write lock on this portal before it can commit its changes. This is needed to prevent reading threads from accessing the buffer manager while memory pages are copied from the transaction buffers into the real buffers.
In summary there can be different types of access to a table:

A.10.3.3 Read access

In the context of a transaction this acquires a shared lock until the transaction is completed. Outside a transaction read access acquires a read lock on the table for the split second it takes to actually perform the read.

A.10.3.4 Write access

This can only be done in the context of a transaction (a transaction is automatically started and immediately committed if no active transaction is present). Write access will acquire an exclusive lock until the transaction is completed and committed. It also requires an active transaction (which has already acquired an exclusive lock, preventing read access from inside other transactions) and acquires a write lock (preventing reads from outside a transaction) for the time it takes to commit the changes to disk.

A.10.4 Transaction isolation levels

First a quick description of the different possible isolation levels:

A.10.4.1 READ UNCOMMITTED or DIRTY READ

You can read an uncommitted transaction that might get rolled back later. This isolation level is also called a dirty read. This is the lowest isolation level.

A.10.4.2 READ COMMITTED

This ensures that data that another application has changed and not yet committed can not be read, but it does not ensure that the data will not be changed before the end of the current transaction.

A.10.4.3 NON REPEATABLE READ

This occurs when a transaction reads the same record more than once and, between the two (or more) reads, a separate transaction modifies that record. Because the record was modified between reads within the same transaction, each read produces different values, which introduces inconsistency.

A.10.4.4 REPEATABLE READ

When it's used, then dirty reads and no repeatable reads cannot occur.

A.10.4.5 PHANTOM

Phantom behavior occurs when a transaction attempts to read a record that does not exist and a second transaction inserts the record before the first transaction finishes. If the record is inserted, the record appears as a phantom to the first transaction, inconsistently appearing and disappearing.

A.10.4.6 SERIALIZABLE

Most restrictive isolation level. When this is used, then phantom values cannot occur. It prevents other users from updating or inserting records into the data set until the transaction is complete.

A.10.5 Transactions in NexusDB

There is only one physical type of transaction in NexusDB. The distinction between implicit and explicit transactions is a virtual one. Any operation that results in a write access to a table checks if a transaction is already present. If not present, then a transaction is started (implicit), the operation performed (insert, modify, delete) and the transaction committed/rolled back depending on the success of the operation.
An explicit transaction is a client side controlled transaction.
Using the above terminology, we can now classify NexusDB as follows:

Dataset state or context	NexusDB
outside of an explicit transaction	read committed
in the context of an explicit. transaction	serializable
locking granularity	table level

Table 6:NexusDB Transactions

A.10.6 Failsafe Transactions

An explicit transaction has its own “server cache” in memory on the server. This cache is only visible to that transaction. All updated records are written to the server cache. Pages of the cache

which have been changed are flagged “dirty” and when the transaction is committed will be written to the tables on disk. If the power fails when some, but not all, of the dirty pages have been written to disk the database will be left in an inconsistent state.
To prevent this possibility set the Failsafe property of your TnxDatabase component to true. As the transaction proceeds clean “before images” of the cache pages which are about to be updated are written to a Transaction Journal File (TJF). When a transaction is committed all the dirty pages are written as “after images” to the TJF and the completed file is closed and flushed to disk. The dirty pages are then written to the database tables on disk and when complete the TJF is deleted. If the power is lost the TJF can be used later to automatically complete the transaction.
For a Rollback the dirty cache pages are discarded and the TJF is deleted.
There are 3 settings available with Failsafe transactions:

- Mode 1 – before and after images are stored. If a reboot finds these images then a dialog is displayed and the operator asked whether to commit or rollback the changes found.
- Mode 2 – stores before images only. If a reboot finds these images an automatic rollback is performed. This can be thought of as a conservative, or cautious, mode of operation.
- Mode 3 – stores after images only. If a reboot finds these types of images an automatic commit is performed.

Mode 2 and 3 require no user interaction. This makes them perfect if the NexusDB Server is running as a service.

A.10.7 Deadlocks

There are 2 different kinds of deadlocks than can occur:
a) User A acquires a record-level content lock. (Releasing that lock means that user A must either cancel its modifications or post them, posting results in an implicit transaction that requires an exclusive transaction lock.) User B starts an explicit transaction (which acquires an exclusive transaction lock and wants to change the record that user A has locked (which requires a record-level content lock).
To resolve this deadlock user A must release its content-lock. It must ether cancel its modification or user B must rollback the transaction, so that user A can post its modification.
This type of deadlock could and has occurred with NexusDB 1.
b) User A starts a transaction and reads from a table (acquires a shared lock). User B starts a transaction and reads from the same table (acquires a shared lock). User A wants to write to the table (tries to acquire an exclusive lock on the table and must wait for user B to release its shared lock). If user B now completes its transaction without trying to write to the table everything is ok. But if user B wants to modify the table it too must acquire an exclusive lock. Now both transactions have a shared lock, want an exclusive lock and are waiting for each other. We have a deadlock.
To resolve this deadlock one of the transactions must be rolled back and retried later.
This type of deadlock is new in NexusDB.

A.10.8 Schemes for Avoiding Deadlocks

If you only have a single table, or only develop single-exe applications that do not use distributed processing or threads, then you can be absolutely certain that you will never need to plan for deadlocks. The moment you have more than one point in your application where you perform database operations in the context of a transaction, the chances are that at least some of them can be executed at the same time (by different users/threads) if the transaction locking is at the table level and can differentiate between shared and exclusive locks. An awareness of concurrency issues and consequent deadlock situations is thus paramount to an understanding of how your applications will perform in the real world when using a NexusDB backend.

A.11 Common suggestions

Here we will point out some things that will generally improve your code and its runtime speed.

A.11.1 Use the NexusDB Memory Manager

We have developed a complete new Memory Manager to optimize memory performance with NexusDB. Please make sure you are using it to get the best possible performance. You can find more information on the memory manager in Section 8.1.
To get the benefits of the new memory manager, it **must** be listed as the very first unit in the project file (.dpr)!

```
uses
  nxReplacementMemoryManager,
  Forms,
  Mainform in 'Mainform.pas' {MainformDialog};
```

A.11.2 Explicit Transactions

Use explicit transaction where possible, because this minimizes the number of I/O accesses NexusDB has to make. For a more detailed description of transactions (and how they are implemented in NexusDB), please look at the according section in this document.

A.11.3 Caution and notes

Ok, seems simple so far, doesn't it? Well, it is. However, if you are not careful, a few things may come back to bite you. Here's a list of cautions I have compiled over the past months:

- Always leave the ActiveRuntime properties as False, and activate the connection to the server in code at runtime. Why? Because otherwise the database component might try to connect to the AliasName/AliasPath set in the IDE of your development machine and this may not be the path the end user runs from.
- Always open the data module as the first form in the IDE before doing anything else. Otherwise your table and query components will have nothing to connect to and you will be wondering why the IDE has apparently gone AWOL.
- Make sure in your project options that the data module is the first form auto-created. It (obviously) is not going to be the main form, but it must be created before any other form (unless you want to do a lot of extra programming).

A.12 Problem reports

When reporting a problem, please use our newsgroups to do so. They are located at news://news.nexusdb.com and offer a variety of groups.
The main support newsgroup is news://news.nexusdb.com/nexusdb.public.support.
Please be sure to include the following information with any problem report.

- Platform and version information, e.g., using NexusDB 1.01 in Delphi 5.01 on Win2k SP3
- Communications transport, e.g., Connected via TCP, but the problem occurs even with Named pipes.
- When the problem cropped up, e.g., "This only occurs onsite – I can not reproduce the problem on my own machine", or, "The problem came as soon as I swapped the components"
- Anything unusual that you think might help in solving the issue, e.g., "This was all working ok last week (of course) and the only thing that has changed since then is that I've added another NIC to the workstation."
- If the problem involves code you have written, be sure to copy the relevant parts into your message. This avoids wasting time on wrong assumptions.

Finally, some suggestions that help to get a fast response.

- Do not include attachments with your postings in the discussion newsgroups. If you have sample code, or one of the experts asks for an example, post it to the nexusdb.public.binaries newsgroup, and then add a posting to the regular newsgroup, saying, eg., "Eivind, I've uploaded the test project to binaries and called it 'NX test project that wipes the HD clean'".
- Use of capitals (especially for all the words in one sentence) is considered yelling. This is rude. You should be able to get your point across, no matter how frustrating, without resorting to yelling. If you have an emergency situation (relatively rare), most users will respond to the words "urgent help needed" or similar in the title. Beware the boy who cried wolf too often, though.
- Try to give an indication of the issue in the first couple of sentences. Then provide the steps necessary to reproduce the problem (if applicable).
- Try not to email anyone directly when responding. Your thread might be describing a problem that someone else is also having and they are interested in a solution as well.

Another good thing about the NG is the number of OT (Off Topic) discussions that pop up from time to time. If you are new to using newsgroups, some of the more common acronyms used are:

<g> - grin	<bg> - big grin	<vbg> - very big grin
btw - by the way	fwiw - for what its worth	imho - in my humble opinion
asfaik – as far as I know	iirc – if I remember correctly	rtfm – read the fword-ing manual

Table 7: Common Newsgroup abbreviations

Finally, if you are seeking answers to questions you suspect might have been asked before, try using http://www.fulltextsearch.com to search our groups. Be sure to add "^nexusdb" to filter on our newsgroups.