

# **NativeDB 1.95 for Sybase SQL Anywhere**

## **A Programmer's Guide**



*Copyright © 1995-2005 Liodden Data - All rights reserved  
No part of this document covered by the copyright  
may be reproduced or otherwise copied  
without prior written consent of the authors*

# Contents

<b>1. Introduction .....</b>	<b>5</b>
1.1 Overview.....	5
1.2 Copyright and licensing terms .....	6
<b>2. Installation .....</b>	<b>7</b>
2.1 Files and directories .....	7
2.2 Online help .....	9
2.3 Units and OOP structures .....	10
2.3.1 Classes .....	10
2.3.2 Unit NdbBase.....	11
2.3.3 Unit NdbAsa.....	12
2.3.4 Unit NdbAsaLg.....	12
2.3.5 Unit NdbBasDS.....	12
2.3.6 Unit NdbAsaDS.....	13
2.3.7 Unit NdbAlert .....	13
2.3.8 Unit NdbApp .....	13
<b>3. Component Reference.....</b>	<b>14</b>
3.1 TAsaSession .....	14
3.1.1 Properties .....	14
3.1.2 Methods .....	19
3.1.3 Events .....	22
3.2 TAsaSQL .....	23
3.2.1 Properties .....	23
3.2.2 Methods .....	25
3.2.3 Events .....	30
3.3 TAsaLogin .....	30
3.3.1 Properties .....	30
3.3.2 Methods .....	31
3.4 TAsaDataset.....	31
3.4.1 Properties .....	31
3.4.2 Methods .....	32
3.4.3 Events .....	33
3.5 TAsaStoredProc .....	33
3.5.1 Properties .....	34
3.5.2 Methods .....	34
3.5.3 Events .....	34
3.6 TAsaAlert.....	35
3.6.1 Properties .....	35
3.6.2 Methods .....	36
3.6.3 Events .....	36
3.7 TNativeParam .....	37
3.7.1 Properties .....	37
3.7.2 Methods .....	38
3.8 TNativeParams.....	38
3.8.1 Properties .....	38
3.8.2 Methods .....	39
3.9 ENativeException .....	39
3.9.1 Properties .....	39
3.10 EAsaException .....	39
3.11 ENativeDatasetError .....	39
<b>4. Component Usage .....</b>	<b>40</b>
4.1 Working with TAsaSession.....	40
4.1.1 Personal engine connection .....	40
4.1.2 Network engine connection .....	41
4.1.3 Runtime engine connection .....	41

4.1.4 Client connection .....	41
4.1.5 Workgroup connection.....	42
4.1.6 Transaction handling .....	43
4.1.7 Backing up the database .....	44
4.1.8 Server messages.....	44
4.1.9 Multi-threading .....	45
4.2 Working with TAsaSQL .....	46
4.2.1 Queries .....	46
4.2.2 Datatypes.....	46
4.2.3 Binary data.....	47
4.2.4 Binary Large Objects (BLOBS).....	48
4.2.5 Handling NULL values .....	49
4.2.6 Appending rows .....	49
4.2.7 Appending rows containing a auto-incremental column .....	50
4.2.8 Modifying rows .....	50
4.2.9 Deleting rows .....	50
4.2.10 Executing SQL statements .....	50
4.2.11 Using parameters .....	51
4.2.12 Using prepared statements .....	52
4.2.13 Calling stored procedures.....	53
4.2.14 Using compound statements .....	55
4.2.15 Breaking a running stored procedure .....	55
4.3 Working with TAsaDataset .....	56
4.3.1 Queries .....	57
4.3.2 Parameters .....	57
4.3.3 Searching.....	57
4.3.4 Appending rows .....	57
4.3.5 Modifying rows .....	58
4.3.6 Deleting rows .....	58
4.3.7 Updating BLOBs .....	58
4.3.8 Concurrency issues .....	59
4.4 Working with TAsaStoredProc .....	62
4.4.1 Stored Procedures with INOUT and OUT parameter types .....	62
4.4.2 Stored Procedures returning Result sets.....	62
4.4.3 Resuming a Stored Procedure .....	62
<b>5. Appendices .....</b>	<b>63</b>
5.1 Converting from BDE to NDB.....	63
5.1.1 Step 1 - Replacing TDatabase or a BDE Alias .....	63
5.1.2 Step 2 - Replacing TQuery .....	64
5.1.3 Step 3 - Replacing TTable.....	64
5.1.4 Step 4 - Replacing TStoredProc.....	64
5.1.5 Step 5 - Changing Sourcecode .....	65
5.2 Using NativeDB with C++Builder .....	65
5.2.1 Connecting to a running database engine.....	65
5.2.2 A simple query .....	66
5.2.3 A query with host variables.....	66
5.2.4 Appending rows .....	66
5.2.5 Modifying rows .....	67
5.2.6 Deleting rows .....	67
5.2.7 Executing SQL statements .....	67
5.2.8 Calling Stored Procedures.....	67
5.3 Using NativeDB ActiveX with Visual Basic.....	68
5.3.1 Running the VB demo.....	68
5.3.2 Getting started .....	68
5.3.3 Client connection .....	68
5.3.4 Starting and connecting to the ASA personal or runtime engines.....	68
5.3.5 A simple query .....	69
5.3.6 A query with host variables.....	69
5.3.7 Executing SQL statements .....	70

5.3.8 Calling Stored Procedures.....	70
5.3.9 Appending rows .....	70
5.3.10 Modifying rows .....	71
5.3.11 Deleting rows .....	71
5.3.12 Using Events.....	71

# 1.Introduction

## 1.1 Overview

NativeDB is a set of Borland VCL and Microsoft ActiveX database interface components to be installed into favorite development environment. NativeDB supports your RDBMS at a low and direct level, skipping both the Borland Database Engine (BDE) and the Microsoft ADO and ODBC layers. NativeDB uses the same interface as Embedded C programs (ESQL). This provides the fastest interface available to your Sybase SQL Anywhere database engine.

### Key-features

- No need to install or configure the Borland Database Engine (BDE).
- No need to configure ODBC or maintain ODBC aliases.
- No need to worry about ADO, MDAC or any MDAC version conflict issues.
- No need to maintain any system-wide Windows registry keys.
- No required system PATH settings.
- No additional Windows SYSTEM DLLs needed when deploying your end-user application.
- Much smaller setup executable size when deploying your application without BDE or MDAC.
- High-speed database access.
- Supports many advanced database features not supported by BDE, ODBC or ADO based interfaces.
- Supports database callbacks, translated into VCL or ActiveX events.
- Intersession/interclient communication.
- Accurate query record-counters.
- Accurate scrollbar positioning when viewing data in VCL grids.
- Immediate access to auto-incremental or other default values when inserting rows.
- Full BLOB support.
- Supports live server-side cursors.
- Works in combination with many popular 3rd party data-access components.

### Supported Development tools

- Borland Delphi 3.02
- Borland Delphi 4.03
- Borland Delphi 5.01
- Borland Delphi 6.02
- Borland Delphi 7.00
- Borland Delphi 2005 (Win32)
- Borland C++Builder 4.02
- Borland C++Builder 5.01
- Borland C++Builder 6.00
- Borland Kylix 1.00
- Borland Kylix 2.00
- Borland Kylix 3.00
- Microsoft Visual Basic (ActiveX)
- Microsoft Visual C++ (ActiveX)
- Any ActiveX enabled development environment

### Supported Sybase editions

- Watcom SQL 3.2
- Watcom SQL 4
- SQL Anywhere 5
- Adaptive Server Anywhere 6

- Adaptive Server Anywhere 7
- Adaptive Server Anywhere 8
- Adaptive Server Anywhere 9

NativeDB was developed based on our own customers need. We wanted to create a native database interface that was easy to use and learn. With NativeDB you don't have to write lengthy blocks of code, instead you take advantage of our usage of passed parameters, variants and boolean return values.

Borland note: Even though we fully support VCL data-aware components through TAsaDataset and TAsaStoredProc, derived from TDataSet, we recommend you to investigate the power of our foundation component TAsaSQL.

## 1.2 Copyright and licensing terms

Please read this License Agreement before installing Liodden Data's NativeDB software.

This License Agreement is a legal agreement between you and Liodden Data for the NativeDB product, which includes the computer software, online documentation and associated media.

This software, including documentation, source code, and additional materials is owned by Liodden Data - Norway.

The registered license grants you a limited, nonexclusive, royalty-free right to use the software with the following conditions. Your application software product:

1. is distributed as a compiled binary.
2. in no way competes, commercially or otherwise, with this software.
3. does not distribute the source code or part thereof in any form.

By installing, copying, or otherwise using the software, you agree to be bound by the terms of this License Agreement. This software is licensed as a single product. It's component parts, may not be separated for use by more than one developer. Each developer must register a separate license.

This software is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The software is licensed, not sold. The software consists of computer software, product documentation, sample applications, technical information and development tools.

Treat the software like any other copyrighted material except that you are authorized to either:

1. make one copy of the software solely for backup purposes, or
2. install the software on a single computer provided you keep the original solely for backup purposes. You may not copy the printed materials, which accompany the software, if any.

Liodden Data grants to you a limited, nonexclusive, royalty-free right to reproduce and distribute those files required for run-time execution if any of compiled applications in conjunction with and as a part of your application software product that is created using Borland Delphi, Kylix, C++Builder or any ActiveX enabled tool, provided that:

1. you include a valid copyright notice in your software product.
2. you do not charge separately for the runtime files.
3. you do not modify the runtime files.
4. you agree to indemnify Liodden Data from any claims or lawsuits, including attorney's fees, that arise or result from the use or distribution of your application software product.

This software product is provided "as is" without warranty of any kind. To the maximum extent permitted by applicable law, Liodden Data disclaims all warranties, either expressed or implied, including but not limited to, fitness for a particular purpose. Liodden Data's entire liability and your exclusive remedy shall not exceed the price paid for the software.

Without prejudice to any other rights, Liodden Data may terminate this agreement if you fail to comply with the terms and conditions laid out here. In such event, you must destroy all copies of the software and all of its component parts.

If you have any questions regarding this License Agreement, please contact Liodden Data.

## 2. Installation

### 2.1 Files and directories

After you completed the setup executable, you will have a NativeDB directory with the following structure, depending on which NativeDB edition you installed:

```
NativeDB\  
  ActiveX  
  Bonus  
  CBuilder4  
  CBuilder5  
  CBuilder6  
  Delphi3  
  Delphi4  
  Delphi5  
  Delphi6  
  Delphi7  
  Delphi9  
  Kylix1  
  Kylix2  
  Kylix3  
  Demos  
  Doc
```

The component binaries are found under your compiler-named directory. If you purchased the professional edition, you will have an additional folder containing the Object Pascal source-codes. The ActiveX automation server is found in the ActiveX directory.

To install the Borland package, it's highly important that you follow the installation steps below. It's important that the base class VCL package is installed BEFORE the database specific package, or your Borland environment may refuse to load the packages properly.

#### Delphi 3

- Start Delphi
- Select Component | Install Packages, and press Add.
- Locate the directory "Delphi3\" from the NativeDB home directory.
- Install the package "NdbPack3.dpl" for the base classes.
- Install the package "NdbSa3.dpl" for the SA specific classes.
- Select Tools | Environment Options | Library tab | Library Path, and add the compiler named directory to your unit search path.

#### Delphi 4

- Start Delphi
- Select Component | Install Packages, and press Add.
- Locate the directory "Delphi4\" from the NativeDB home directory.
- Install the package "NdbPack4.bpl" for the base classes.
- Install the package "NdbSa4.bpl" for the SA specific classes.
- Select Tools | Environment Options | Library tab | Library Path, and add the compiler named directory to your unit search path.

### **Delphi 5**

- Start Delphi
- Select Component | Install Packages, and press Add.
- Locate the directory "Delphi5\" from the NativeDB home directory.
- Install the package "NdbPack5.bpl" for the base classes.
- Install the package "NdbSa5.bpl" for the SA specific classes.
- Select Tools | Environment Options | Library tab | Library Path, and add the compiler named directory to your unit search path.

### **Delphi 6**

- Start Delphi
- Select Component | Install Packages, and press Add.
- Locate the directory "Delphi6\" from the NativeDB home directory.
- Install the package "NdbPack6.bpl" for the base classes.
- Install the package "NdbSa6.bpl" for the SA specific classes.
- Select Tools | Environment Options | Library tab | Library Path, and add the compiler named directory to your unit search path.

### **Delphi 7**

- Start Delphi
- Select Component | Install Packages, and press Add.
- Locate the directory "Delphi7\" from the NativeDB home directory.
- Install the package "NdbPack7.bpl" for the base classes.
- Install the package "NdbSa7.bpl" for the SA specific classes.
- Select Tools | Environment Options | Library tab | Library Path, and add the compiler named directory to your unit search path.

### **Delphi 2005**

- Start Delphi
- Select Component | Install Packages, and press Add.
- Locate the directory "Delphi9\" from the NativeDB home directory.
- Install the package "NdbPack9.bpl" for the base classes.
- Install the package "NdbSa9.bpl" for the SA specific classes.
- In the left pane treeview highlight Environment Options | Delphi Options | Win32 | Library Path, and add your "NativeDB\Delphi9" directory to the Library path.

### **C++Builder 4**

- Start C++Builder
- Select Component | Install Packages, and press Add.
- Locate the directory "CBuilder4\" from the NativeDB home directory.
- Install the package "NdbPack4.bpl" for the base classes.
- Install the package "NdbSa4.bpl" for the SA specific classes.
- Select Tools | Environment Options | Library tab | Library Path, and add the compiler named directory to your unit search path.

### **C++Builder 5**

- Start C++Builder
- Select Component | Install Packages, and press Add.
- Locate the directory "CBuilder5\" from the NativeDB home directory.
- Install the package "NdbPack5.bpl" for the base classes.
- Install the package "NdbSa5.bpl" for the SA specific classes.
- Select Tools | Environment Options | Library tab | Library Path, and add the compiler named directory to your unit search path.

### **C++Builder 6**

- Start C++Builder
- Select Component | Install Packages, and press Add.



- Locate the directory "CBuilder6\" from the NativeDB home directory.
- Install the package "NdbPack6.bpl" for the base classes.
- Install the package "NdbSa6.bpl" for the SA specific classes.
- Select Tools | Environment Options | Library tab | Library Path, and add the compiler named directory to your unit search path.

#### **Kylix 1**

- Start Kylix
- Select Component | Install Packages, and press Add.
- Locate the directory "Kylix1" from the NativeDB home directory.
- Install the package "bplIndbbask1.so" for the base classes.
- Install the package "bplIndbsak1.so" for the SA specific classes.
- Select Tools | Environment Options | Library tab | Library Path, and add the compiler named directory to your unit search path.

#### **Kylix 2**

- Start Kylix
- Select Component | Install Packages, and press Add.
- Locate the directory "Kylix2" from the NativeDB home directory.
- Install the package "bplIndbbask2.so" for the base classes.
- Install the package "bplIndbsak2.so" for the SA specific classes.
- Select Tools | Environment Options | Library tab | Library Path, and add the compiler named directory to your unit search path.

#### **Kylix 3**

- Start Kylix
- Select Component | Install Packages, and press Add.
- Locate the directory "Kylix3" from the NativeDB home directory.
- Install the package "bplIndbbask3.so" for the base classes.
- Install the package "bplIndbsak3.so" for the SA specific classes.
- Select Tools | Environment Options | Library tab | Library Path, and add the compiler named directory to your unit search path.

#### **ActiveX**

- Locate the directory "ActiveX" from the NativeDB home directory.
- Open a Command window in this directory and type "regsvr32 ndbasa32.dll".
- Refer to chapter "5.3 Using the ActiveX library with Visual Basic" for further information in how to use NativeDB ActiveX.

NativeDB is designed around a OOP hierarchy. This is done to make it easy to move your application to another database, or support more than one database in the same application. With this structure, we can also easily add new database interfaces in the future, overriding virtual methods only.

## **2.2 Online help**

This section describes how to install the NativeDB online help system for Borland Delphi and C++Builder.

#### **Delphi 3 or Delphi 4**

- Quit Delphi.
- Locate Delphi's Help directory (e.g. C:\Program Files\Borland\Delphi 3/4\Help).
- In the same directory, edit the text file Delphi3.cfg or Delphi4.cfg and add the following line to the "Third-party Help" section:  
:Link <NativeDB home>\Doc\NdbAsa.hlp  
Where <NativeDB home> is the directory where you installed NativeDB.
- Delete the hidden files Delphi3.gid or Delphi4.gid, and NdbAsa.gid (if present). These files will be recreated the next time you access the help system.

#### C++Builder 4

- Quit C++Builder.
- Locate C++Builder 's Help directory (e.g. C :\Program Files\Borland\CBuilder4\Help).
- In the same directory, edit the text file bcb4.cfg and add the following line to the "Third-party Help" section:  
:Link <NativeDB home>\Doc\NdbAsa.hlp  
Where <NativeDB home> is the directory where you installed NativeDB.
- Delete the hidden files bcb4.gid and NdbAsa.gid (if present). These files will be recreated the next time you access the help system.

#### Delphi 5, 6, 7 or C++Builder 5,6

- Start Delphi or C++Builder.
- From the main menu choose Help | Customize...
- Select the Index tab.
- Choose Edit | Add Files..., then locate and open NdbAsa.hlp.
- Select the Link tab.
- Choose Edit | Add Files..., then locate and open NdbAsa.hlp.
- Choose File | Save Project, then File | Exit.

## 2.3 Units and OOP structures

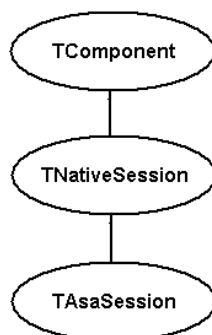
This chapter lists all the units in the NativeDB for SQL Anywhere package. This includes available classes, types, procedures and functions.

NativeDB is shipped as six Borland VCL based classes and two ActiveX classes to access your SQL Anywhere database.

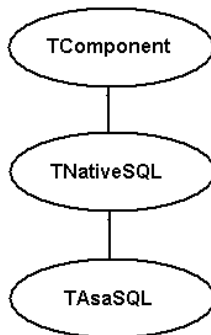
Class	Description
TAsaSession	Used to connect to the database.
TAsaSQL	Used to execute queries, DML statements and stored procedures.
TAsaLogin	Used to connect to the database using a connection expert-dialog.
TAsaDataset	Used to work with data-aware components (TDBEdit, TDBGrid, etc.)
TAsaStoredProc	Used to work with stored procedures and data-aware components.
TAsaAlert	Used to implement intersession communication.
XAsaSession	ActiveX dual interface to TAsaSession
XAsaSQL	ActiveX dual interface to TAsaSQL

### 2.3.1 Classes

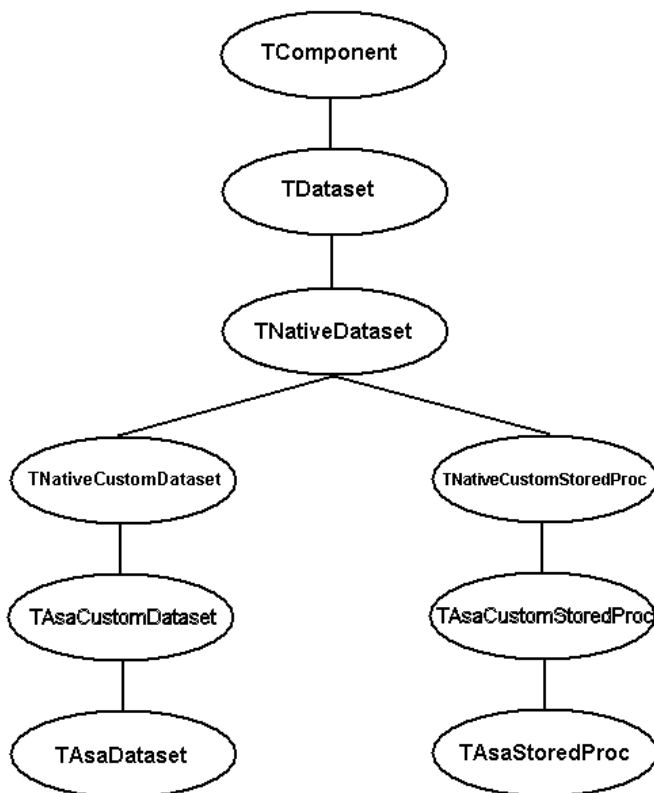
Inherited from the base class TNativeSession, TAsaSession is the connection and session class used to connect to your SQL Anywhere database. Based on the VCL ancestor class TComponent, TAsaSession inherits from the immediate ancestor class TNativeSession.



NativeDB includes a low-level class used to access all the available SQL Anywhere database features. This includes queries, stored procedures, DML, DDL statements. Also based on the common TComponent VCL class, TAsaSQL inherits from the immediate ancestor class TNativeSQL.



NativeDB fully support Borland data-aware components. Two classes are available to meet this requirement. The dataset and cursor oriented component TAsaDataset and the class to access SA stored procedures TAsaStoredProc. Both are originated from the common VCL ancestor class TDataSet.



### 2.3.2 Unit NdbBase

The NdbBase unit contains declarations of the common ancestor classes, inherited by the SA specific descendants.

ActiveX note: N/A

## **Components**

TNativeSession  
TNativeSQL

## **Types**

ENativeException  
TNativeFieldTypes  
TNativeGraphicType  
TNativeFieldDef = class  
TNativeFieldDefs = class

## **NdbEmptyParam: Variant**

A convenient global variable used to indicate a list of empty host variables. This variable is especially useful in Delphi 6 (and later), BCB6 or Kylix because the variant System.Null has changed its behavior with these compilers. NdbEmptyParam is also convenient as a standard way of indicating an empty list of host variables, to achieve code compatibility between all the different Borland compilers.

## **2.3.3 Unit NdbAsa**

The NdbAsa unit contains declarations of the SA specific classes.

ActiveX note: N/A

## **Components**

TAsaSession  
TAsaSQL

## **Types**

EAsaException  
TAsaServerType  
TAsaTransIsolation  
TServerMessageType

## **2.3.4 Unit NdbAsaLg**

The NdbAsaLg unit contains the declaration and implementation of the TAsaLogin component.

ActiveX note: N/A

## **Components**

TAsaLogin

## **Types**

TAsaLoginOption  
TAsaLoginOptions

## **2.3.5 Unit NdbBasDS**

The NdbBasDS unit contains declarations of the common ancestor classes, inherited by the SA specific data-aware descendants.

ActiveX note: N/A

## **Components**

TNativeDataset  
TNativeCustomDataset  
TNativeCustomStoredProc

TNativeParams  
TNativeParam

#### Types

ENativeDatasetError  
TNativeParamType  
TNativeParamDataType

### 2.3.6 Unit NdbAsaDS

The NdbAsaDS unit contains declarations of the NativeDB data-aware classes specific to SA.

ActiveX note: N/A

#### Components

TAsaCustomDataset  
TAsaCustomStoredProc  
TAsaDataset  
TAsaStoredProc

#### Types

ENativeDatasetError

### 2.3.7 Unit NdbAlert

The NdbAlert unit contains the declarations of the NativeDB intersession communication class.

#### Components

TAsaAlert

#### Types

EAsaAlertException  
TSubscriptionType

### 2.3.8 Unit NdbApp

The NdbApp unit declares some common utility functions used to manage the NativeDB classes on a global level. Be aware that this unit **uses** the VCL Forms (or CLX QForms) unit and will potentially increase the size of your executable. An increased size of the executable will only be an issue if you don't use Forms in other parts of your application. In addition, Forms also creates the global VCL Application instance. So don't put NdbApp in your uses class if you don't need it. NdbApp is designed this way, to maintain the modularity of NativeDB. The following functions are currently available:

ActiveX note: N/A

#### **procedure NdbGetDatabaseNames(List: TStrings)**

Fills the List with all available TAsaSession objects. The list will be filled with all active TAsaSession objects found in forms and datamodules, as well as run-time created instances that was created with a valid owner. The List will only contain TAsaSession instances that have the LoginDatabase set.

#### **function NdbFindDatabase(const DatabaseName: string): TNativeSession**

Returns an instance of a TAsaSession object that matches the LoginDatabaseName property with the DatabaseName parameter. The function returns the immediate ancestor class of TAsaSession, so you must typecast the return value to a TAsaSession variable.

```

uses NdbApp;
.
.
function TForm1.GetUserForDatabase(const ADatabase: string): string;
var
  AsaSession: TComponent; // Could be declared as TNativeSession
begin
  Result := '';
  AsaSession := NdbFindDatabase(ADatabase);
  if Assigned(AsaSession) then // Did we find it?
    Result := (AsaSession as TAsaSession).LoginUser;
end;

```

**procedure NdbGetTableNames(const DatabaseName: string; SystemTables, Creators: Boolean; List: TStringList; CursorClass: TNativeDatasetClass);**

Retrieves a list of tables associated with a given database. DatabaseName specifies the name of the database (TAsaSession.LoginDatabase) from which to retrieve all table names. Setting SystemTables to True, will populate the List with both data tables and system tables. Set Creators to True to prefix each table name with the creator/owner ID. List is a string list object, created and maintained by the application, into which to return the table names. CursorClass is a class-type parameter, used by the function, to create a temporary cursor class to query the system tables for the table names. You should pass the class-type TAsaDataset to this parameter.

```
NdbGetTableNames('asademo', False, True, Listbox1.Items, TAsaDataset);
```

**procedure NdbGetStoredProcNames(const DatabaseName: string; Creators: Boolean; List: TStringList; CursorClass: TNativeDatasetClass);**

Retrieves a list of stored procedures associated with a given database. DatabaseName specifies the name of the database (TAsaSession.LoginDatabase) from which to retrieve all stored procedure names. Set Creators to True to prefix each stored procedure name with the creator/owner ID. List is a string list object, created and maintained by the application, into which to return the stored procedure names. CursorClass is a class-type parameter, used by the function, to create a temporary cursor class to query the system tables for the stored procedures. You should pass the class-type TAsaDataset to this parameter.

```
NdbGetStoredProcNames('asademo', True, Listbox1.Items, TAsaDataset);
```

**procedure NdbGetFieldNames(const DatabaseName, TableName: string; List: TStringList; CursorClass: TNativeDatasetClass);**

Populates a string list with the names of fields in a table. Call NdbGetFieldNames to retrieve a list of fields in a table. The names of the fields are put into the already-existing string list object specified in the List parameter. Specify the table for which to retrieve the names of fields in the TableName parameter.

```
NdbGetFieldNames('asademo', 'employee', ListBox1.Items, TAsaDataset);
```

## 3. Component Reference

This chapter is a reference to all the available components in the NativeDB package.

### 3.1 TAsaSession



This component is the link to your SQL Anywhere database. With this component, you control the connection specific details.

ActiveX note: The class interface is named XAsaSession.

#### 3.1.1 Properties

**AutoCommit: boolean**

Specifies, if SQL statements should be automatically committed or manually committed. If you use manual commit (AutoCommit:=False) you must explicitly use the Commit or Rollback methods to commit or rollback the current transaction. Note that the AutoCommit is temporarily disabled inside a transaction (see StartTransaction) and re-enforced after a call to Commit or Rollback.

**ClientParams: string**

Specifies client side connection parameters. This property is used to specify network and client-side connection parameters. Separate each parameter with semicolon (;), when more than one parameter is used.

See the section 'Network communications parameters' in the SA online help, focusing on the client side. Example client connection parameters include:

Product version	ClientParams examples
Watcom SQL 4 network requestor	start=dbclient.exe -x tcpip
SQL Anywhere 5 network client	start=dbclient.exe -x tcpip
ASA6 network client	commlinks=tcpip{host=server;to=5};astop=false
ASA7 network client	commlinks=tcpip{host=server;to=5}
ASA8 network client	commlinks=tcpip{host=server;to=5}
ASA9 network client	commlinks=tcpip{host=server;to=5}

**CommitOnDisconnect: boolean**

Indicates whether or not any pending transactions should be committed upon database disconnection.

**Connected: boolean**

Used to connect to, or disconnect from the database. To connect, assign a true boolean value to this property. Disconnect with false.

**ConnectOnLoading: boolean;**

A design-time connection made during development is typically not very useful when your application is distributed to end-users. An end-user would normally need to login to the database with his own username and password. By setting this property to False, the design-time values of the properties Connected, LoginUser and LoginPassword will be cleared at run-time. This way you can ensure that your application is distributed unchanged, without affecting your design-time connection during development.

ActiveX note: N/A

**DataSetCount: integer**

Indicates the number of active datasets (TDataSet descendants) associated with the TAsaSession component.

ActiveX note: N/A

**DataSets[Index: integer]: TComponent**

Provides an indexed array of all active datasets (TDataSet descendants) for a TAsaSession component. The following example shows how to use this property to close all datasets associated with the TAsaSession component without disconnecting from the database server.

```

procedure CloseDataSets;
var
    I: Integer;
    DataSet: TAsaDataset;
begin
    with AsaSession1 do
        begin
            for I := DataSetCount - 1 downto 0 do
                if DataSets[I] is TAsaDataset then
                    TAsaDataset(DataSets[I]).Close;
        end;
end;

```

end;

ActiveX note: N/A

**Handle: pointer**

Runtime public property to allow one TAsaSession instance to share an existing connection of another TAsaSession instance. Assigning a value to this property makes it possible to let multiple TAsaSession instances share the same connection channel to the database. However note that it's illegal to share a connection handle between two TAsaSession instances that reside in separate threads. This property is especially useful when you want to share a single database connection between TAsaSession instances that reside in Dynamic Link Libraries.

ActiveX note: N/A

**InTransaction: boolean**

Public property to indicate whether a user defined database transaction is in progress or not. Calling StartTransaction sets InTransaction to True. Calling Commit or Rollback sets InTransaction to False.

**KeepConnection: boolean**

Specifies whether an application remains connected to a database even if no datasets are open. When KeepConnection is True (the default) the connection is maintained. When KeepConnection is False the connection is dropped when there are no open datasets.

ActiveX note: N/A

**LastErrorCode: integer**

Returns the low-level error code returned from SQL Anywhere after an unsuccessful database access call. Consider values greater than 0 (zero) as WARNINGS, while values less than 0 (zero) are ERRORS. 0 means SUCCESS.

**LastError: string**

Returns the low-level error message returned from SQL Anywhere after an unsuccessful database access call.

**LastStatement: string**

Returns the last requested database query or statement. Any query or SQL statement as processed by a dataset will report the executing statement to its linked TAsaSession instance. This property is typically used to log or monitor failed statements for later debugging purposes. The following shows an example using the property with a TAsaSession.OnError event handler.

```
procedure TForm1.AsaSession1Error(Sender: TObject);
begin
    ShowMessage(Format('%s. Caused by statement: %s'#10,
        [AsaSession1.LastError, AsaSession1.LastStatement]));
end;
```

**LibraryFile: string**

Specifies the name of the SQL Anywhere interface DLL (or Linux .so) used to communicate with the server engine. This must be the same library interface as used by embedded C programs. For SA version 7 on Windows, this is 'dblib7.dll', which is the default. This is a useful property to easily upgrade to future SA versions.

Product version	Client library
Watcom SQL 4 (WSQL)	dbi40t.dll
SQL Anywhere 5 (SA5)	dbi50t.dll
Adaptive Server Anywhere 6 (ASA6)	dblib6.dll
Adaptive Server Anywhere 7 (ASA7)	dblib7.dll
Adaptive Server Anywhere 8 (ASA8)	dblib8.dll



Product version	Client library
Adaptive Server Anywhere 9 (ASA9)	dblib9.dll
Adaptive Server Anywhere 6 (ASA6 – Linux)	libdblib6.so
Adaptive Server Anywhere 7 (ASA7 – Linux)	libdblib7.so
Adaptive Server Anywhere 8 (ASA8 – Linux)	libdblib8.so
Adaptive Server Anywhere 9 (ASA9 – Linux)	libdblib9.so

**LibraryPath: string**

Full path to the SQL Anywhere interface DLLs and executables directory. If blank, TAsaSession uses the system PATH to locate these files. The path typically points to your SA Win32 directory as installed by the SA setup. If you decided to deploy the minimum required SA files along with your application, the LibraryPath should point to this location instead. This feature allows you to bundle your application with SA for an easy and compact distribution of your application. In general, the LibraryPath should specify the directory where the LibraryFile is located.

Linux note: This property is not applicable in the Linux environment. Assign your library path through the “LD\_LIBRARY\_PATH” environment variable instead.  
If you don’t want to set your “LD\_LIBRARY\_PATH”, edit your /etc/ld.so.conf instead and add the location of the directory containing libdblib?.so to the list and then run ldconfig (as root). But in addition to this, you would also need to set the environment variable ASANY to point to your application’s directory where your deployed ASA files are stored.

**LoginEngineName: string**

Enter the name of the SQL Anywhere server engine to connect to. If empty, SA extracts the file prefix of the LoginDatabase property, and uses this as the engine name. If you have a running server with multiple databases, use LoginEngineName and LoginDatabase to distinguish the different databases loaded by the same server engine.

**LoginDatabase: string**

**LoginUser: string**

**LoginPassword: string**

These three properties specify your logon to the database. You can use a full path, including UNC path, to the database file for both a client and a server connection. By default, SA will extract the file prefix and use it to name the database.

If you are making a client connection, it’s optional to enter just the name of the database, without the full path and filename to the .db file.

**OEMConvert: boolean**

OEMConvert controls whether to convert string values between the ANSI character set and OEM characters, when reading from or writing to the database. Note that this feature utilizes the CharToOem and OemToChar Win32 API functions. These functions rely on your current Windows setting, and that these setting are the same as the OEM collation used in your database. To verify the OEM Windows settings, look up the registry key \HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Control\Nls\CodePage, and compare the value of OEMCP against your current collation.

Linux note: N/A

**ServerParams: string**

Used to specify the SQL Anywhere server startup line. If ServerType is stClient this property is ignored. If more than one parameter is specified, separate them with semicolon. As an example, starting the ASA 7 personal database engine, assign a value 'start=dbeng7.exe' to this property. If you want to include a specification of SA’s cache size, network protocol and quietmode, you could change it to 'start=dbeng7.exe -c 4096K -x tcpip{to=5} -Q'. Refer to the Adaptive Server Anywhere Reference manual for details. Some typical server parameters include:

SQL Anywhere version	ServerParams examples
Watcom SQL 4 personal engine	start=dbeng40.exe

Watcom SQL 4 network engine	start=dbsrv40.exe
SQL Anywhere 5 network engine	start=dbsrv50.exe -c 4096K -x tcpip{to=5}
SQL Anywhere 5 personal engine	start=dbeng50.exe -c 4096K
SQL Anywhere 5 runtime engine	start=rtask50.exe
ASA6 network engine	start=dbsrv6.exe -c 4096K -x tcpip{to=5} -gd all -Z
ASA6 personal engine	start=dbeng6.exe -c 4096K -Q;astop=False
ASA6 runtime engine	start=rteng6.exe
ASA7 network engine	start=dbsrv7.exe -x tcpip -gd all
ASA7 personal engine	start=dbeng7.exe
ASA7 runtime engine	start=rteng7.exe
ASA8 network engine	start=dbsrv8.exe -x tcpip -gd all
ASA8 personal engine	start=dbeng8.exe
ASA8 runtime engine	start=rteng8.exe
ASA9 network engine	start=dbsrv9.exe -x tcpip -gd all
ASA9 personal engine	start=dbeng9.exe
ASA9 runtime engine	start=rteng9.exe
ASA6 personal engine (Linux)	start=dbeng6
ASA7 network engine (Linux)	start=dbsrv7 -x tcpip
ASA8 network engine (Linux)	start=dbsrv8 -x tcpip

#### ServerType: TServerType

Specifies the type of connection. Valid options are (stClient, stServer, stWorkgroup). The default server type is stClient. The following rules apply to the different options:

ServerType	Rule
stClient	Connects to a running SA server using the ClientParams property.
stServer	First executes the engine specified by the ServerParams property. Then connects to it, using ClientParams.
stWorkgroup	First tries to connect as stClient. If this fails, then tries to connect as stServer.

#### TransIsolation: TTransIsolation

Sets the locking isolation level used within the current connection. This allows you to control the degree to which the operations in one transaction are visible to the operations in other concurrent transactions. Valid options are (atUnchanged, atReadUncommitted, atReadCommitted, atRepeatableRead, atSerializable). The relationship between the TTransIsolation type and the low-level SA types is listed in the following table:

TTransIsolation level	SA isolation level
atUnchanged	Keeps the current isolation level unchanged
atReadUncommitted	Level 0
atReadCommitted	Level 1
atRepeatableRead	Level 2
atSerializable	Level 3

Refer to your SA Reference manual for details.

#### TrimBlanks: boolean

Specifies whether right-padded space characters in character fields (strings) should be removed. When TrimBlanks = False any right-padded space characters remain in the string field. This property is useful if your database was initially (dbinit) created with the "-b" switch, specifying that blanks should be ignored in comparisons. In this state, ESQL will pad every character field with space characters.

#### WantExceptions: boolean

Control if exceptions are raised or not. TAsaSession raises an EAsaException for errors if this property is true. The default is False. The property does not affect critical errors.

**WantRowCounts: boolean**

Controls if rowcounts should be accurate for cursors opened for this session. The default is True. When True the TAsaSQL property RowCount will hold an accurate value, after a cursor is opened. If False, the RowCount will hold an estimated value. There is a small performance hit if this property is True. (see TAsaSQL.RowCount).

Note: When using TAsaDataset it's recommended to set WantRowCounts to True to be able to properly relocate the correct row after a TAsaDataset.Refresh call. Refresh is also internally used after a successful TAsaDataset.Post and TAsaDataset.Delete call. Also note that if WantRowCounts is False, data-aware scrollbars will only be shown in a 3-state position.

**3.1.2 Methods****constructor Create(AOwner: TComponent)**

Creates a TAsaSession object instance.

ActiveX note: N/A

**destructor Destroy**

Destroys a TAsaSession object instance.

ActiveX note: N/A

**function GetLibraryVersion: string**

Returns the current SQL Anywhere version string (EBF).

**function Backup(AFilePath: string; ABackupType: TBackupType): boolean;**

This method utilizes the online backup feature of SQL Anywhere. It takes a backup of the database, the transaction log and the mirror log file if any. AFilePath specifies the target directory of the backup. ABackupType specifies the type of backup to execute. Valid values include btFull, btFullRenameLog, btRenameLog, btFullRenameLocalLog, btRenameLocalLog, btFullRestartLog, btRestartLog. A return value of True, indicates a successful operation. The following table lists each backup type, their meaning and their corresponding ASA dbbackup flags.

Backup type	Description	dbbackup
btFull	Full backup of .db, .log and .mlg.	-d -t -w
btFullRenameLog	Full backup and renames transaction log.	-d -t -w -r
btRenameLog	Renames the transaction log only.	-r
btFullRenameLocalLog	Full backup and renames local transaction log to match server.	-d -t -w -r -n
btRenameLocalLog	Backup and renames local transaction log to match server.	-t -r -n
btFullRestartLog	Full backup. Deletes and restarts the transaction log.	-d -t -w -x
btRestartLog	Deletes and restarts the transaction log only.	-xo

**procedure Close**

Closes all open datasets associated with the database component and disconnects from the database server. A database connection can also be closed by setting the Connected property to False.

**procedure Commit**

Commits the current transaction.

**function GetConnectionInfo(ConnInfo: TAsaConnectionInfo; var Value: string): boolean**

Used to retrieve connection information for the current connected database. The information to retrieve is specified by ConnInfo, and the result is returned in Value. Valid connection information options, as given by ConnInfo, are ciEngineVersion, ciEngineName or

ciDatabaseName. This method is typically used in cases where you connect to the default server engine, without specifying LoginEngineName or LoginDatabase. The method returns True if the information was retrieved successfully.

```
if AsaSession1.GetConnectionInfo(ciEngineName, szValue) then
    Caption := szValue;
```

**function GetOption(Option: PChar; var Value: string): boolean**

Used to retrieve SQL Anywhere database options. See your SQL Anywhere Reference manual. Returns True if the option was successfully retrieved.

```
if AsaSession1.GetOption('date_order', szValue) then
    Caption := szValue;
```

**function GetProperty(PropType: TAsaPropertyType; var Value: string): boolean**

Used to retrieve properties of an ASA8 (or later) database or server. Currently, only one property ptServerAddress is supported. This property is used to obtain the address of the server to which you are currently connected. Other database properties can also be obtained by executing a regular SELECT statement (e.g. select db\_property('PageSize')).

```
if AsaSession1.GetProperty(ptServerAddress, szValue) then
    Caption := szValue;
```

**function IsWorking: boolean**

Returns True if your application has a database request in progress using the same TasaSession instance. This function can be called asynchronously. Refer to your SA documentation on the "db\_is\_working" function for more details.

**function LocateServers: boolean**

This method implements the same functionality as the ASA7 "dblocate.exe" utility. The LocateServers method locates all available ASA (7.00 and later) servers on the local network that are listening on TCP/IP. LocateServers is only supported by ASA7.01 or later. When executed, the OnLocateServers event is called for each server engine found.

**function NeedQuotes(Identifier: PChar): boolean**

Returns a Boolean value that indicates whether the string requires double quotes around it when it is used as a SQL identifier. Refer to your SA documentation on the "sql\_needs\_quotes" function for more details.

**procedure Open**

Connects to a database server. Setting Connected to True also connects to the database server.

**function Ping(Params: PChar): boolean**

This method implements the same functionality as the ASA DBPing utility, introduced with ASA version 6.02. Typically used to assist in diagnosing connection problems. The function returns true if the specified database server engine is found. If Ping is used with SQL Anywhere 5.5 or earlier releases, it uses the SA db\_find\_engine function. The following example shows result information after the function call.

```
if not AsaSession1.Ping('eng=asademo;links=tcip{host=server}') then
    ShowMessage(AsaSession1.LastError)
else
    ShowMessage('Ping server successful.');
```

```
if AsaSession1.Ping('eng=sademo') then
    ShowMessage('Engine found.');
```

**procedure Rollback**

Rolls back the current transaction.

**function SetOption(Option, Value: PChar): boolean**

Used to set a SQL Anywhere database option. Returns True if the option was updated successfully. See your SQL Anywhere Reference manual.

```
AsaSession1.SetOption('date_order', 'MDY');
```

### procedure StartTransaction

Used to mark a beginning of a new user defined transaction against the database server. Check the status of the InTransaction property and adjust the setting of the TransIsolation property as desired, before calling StartTransaction. If InTransaction is True, indicating that a user transaction is already in progress, any subsequent call to StartTransaction without first calling Commit or Rollback to end the current transaction, raises a ENativeException.

### function StartDatabase(UtilityConnParams: PChar): boolean

Used to start the database LoginDatabase on an existing server LoginEngineName, if it's not already running. Typically used to start LoginDatabase, on a remote engine. With ASA6 or later, if the start or stop requests are protected by a password, you can optionally use the UtilityConnParams parameter to connect through the utility database. To use the utility database, the file 'util\_db.ini' must exist in the same directory as the server engine executable. See the section 'Utility database server security' in the ASA User's Guide. The following example shows how to start the LoginDatabase on a remote engine with ASA6 or later, using the utility database:

```
AsaSession1.LoginEngineName := 'asademo';
AsaSession1.LoginDatabase := 'c:\Sybase\Asa6\asademo.db';
AsaSession1.ClientParams := 'links=tcipip{dobroadcast=no;host=srv}'
if not AsaSession1.StartDatabase('eng=srv;dbn=utility_db;uid=dba;pwd=sql') then
    ShowMessage(AsaSession1.LastError)
else
    ShowMessage('Started database successfully.');
```

The following example shows how to start a database without using the utility database, or with SA 5 or earlier releases:

```
if not AsaSession1.StartDatabase('') then
    ShowMessage(AsaSession1.LastError)
else
    ShowMessage('Started database successfully.');
```

### function Stop: boolean

Stops or breaks the currently active database server request. Useful to let another thread stop a running stored procedure or another long running SA server request, executed by a different thread. This function can be called asynchronously. Returns true if the operation was successful. Can also be called from the OnServerWait callback event, in case of a single threaded environment.

### function StopDatabase(UtilityConnParams: PChar): boolean

Used to stop the database identified by LoginDatabase on the server identified by LoginEngineName. See TAsaSession.StartDatabase. With ASA6 or later, you can even stop a database that has connected users. To force a database stop in this situation, add the 'unc=yes' (unconditional stop) parameter to ClientParams. Note, if you encounter problems starting and stopping your database, check the '-gd' switch available with ASA6 or later.

### function TimeChange: boolean

This is an optional function to be used to notify an ASA8 (or later) server that the time has changed on the client. For WIN32 clients only, it is recommended that the client application calls this function in response to a WM\_TIMECHANGE message. The WM\_TIMECHANGE message is sent to all top-level windows after changing the system time. Internally the function recalculates any time zone adjustment and sends it to the server. This will make sure that UTC timestamps are consistent over time changes, time zone changes, or daylight savings time changeovers. This function has no effect if not connected to an ASA8 (or later) server.

```
type
    TForm1 = class(TForm)
    private
        procedure WMTimeChange(var Message: TWMTTimeChange); message WM_TIMECHANGE;
```

```

end;

procedure TForm1.WMTimeChange(var Message: TWMTTimeChange);
begin
    AsaSession1.TimeChange;
end;

```

### 3.1.3 Events

#### **AfterConnect: TNotifyEvent**

Triggers right after a successful connection is made.

#### **BeforeConnect: TNotifyEvent**

Triggers just before a connection attempt is made.

#### **AfterDisconnect: TNotifyEvent**

Triggers right after a successful disconnection is made.

#### **BeforeDisconnect: TNotifyEvent**

Triggers just before a disconnection attempt is made.

#### **OnBackup: TBackupEvent**

Triggers for each page backed up from the database file(s). This event is useful for displaying a progress bar, showing the status of the running backup.

TBackupEvent = procedure(Sender: TObject; BackupFile: string; CurrPage, NumOfPages: Integer; var AbortBackup: Boolean).

BackupFile represent the current file been backed up (Mydb.db, Mydb.wri, Mydb.log). NumOfPages specifies the total number of buffer pages the current file consists of, while CurrPage specifies the page number currently backed up. Assigning True to AbortBackup will immediately stop the backup.

#### **OnConnectionDropped: TNotifyEvent**

This event notifies the client whenever the client interface DLL is about to drop a connection caused by liveness timeouts. The event is only supported in ASA 7.01 or later releases.

TAsaSession.Connected is NOT forced to False because of ESQL reentrancy rules. To force Connected to False, using this event, in your own application, you must avoid reentrancy. A possible solution is to use a Win32 API PostMessage call, and post a user defined message to your window handle. In the message handler you can toggle the Connected property to reconnect to the server engine.

#### **OnConnectionTerminated: TNotifyEvent**

This event is called when a database connection is lost or dropped. Upon calling the event Connected is forced to False, cursors are closed, and all internal buffers are freed. It's fully possible to reconnect in this event handler, but note that all transactions in progress are invalidated, and all cursors and statements need to be re-executed.

#### **OnDebugMessage: TDebugMessageEvent**

This event will get called once for each client-side debug message which normally only goes to a debug log file.

TDebugMessageEvent = procedure(Sender: TObject; Msg: string) of object;  
Msg specifies string containing the text of the debug message.

This event is only supported in ASA 8.00 or later releases.

#### **OnError: TNotifyEvent**

Triggers when an error occurs. LastError is updated just before the event. Useful for logging error messages in a common event. OnError is of type TNotifyEvent = procedure (Sender: TObject).

#### **OnLocateServers: TLocateServersEvent**

This event is called by TAsaSession.LocateServers for each server engine it locates on the

Local Area Network.

`TLocateServersEvent` = procedure(Sender: TObject; ServerName, ServerAddress: string; ServerPort: Integer; var AbortLocate: Boolean) of object;  
ServerName specifies the name of the server engine that was located.  
ServerAddress specifies the name of the computer hosting the server engine.  
ServerPort specifies the server engine listening port number.  
If you assign True to AbortLocate, `TAsaSession.LocateServers` will stop iterating through servers. This event is only supported with ASA 7.01 or later releases.

#### **OnServerFinish: TNotifyEvent**

This event is called after the responds to the database request has been received by the client. Note: Spending too much time in the event handler could lead to a performance hit with the current database request. For more information regarding this event, locate the index "db\_register\_a\_callback" in SA/ASA online help.

#### **OnServerMessage: TServerMessageEvent**

This event is used to enable the application to handle messages received from the server during the processing of a request. The event is only supported in ASA 6.01 or later releases.  
`TServerMessageEvent` = procedure(Sender: TObject; MessageType: TServerMessageType; MessageCode: integer; Msg: string)  
MessageType states how important the message is, and you may wish to handle different message types in different ways. Available message types are `smtUnknown`, `smtInfo`, `smtWarning`, `smtAction` and `smtStatus`. MessageCode is an additional message identifier. Msg specifies the message. For more information regarding this event, locate the 'db\_register\_a\_callback' index in SA/ASA online help, or refer to the "MESSAGE statement" index for a discussion of the 'MESSAGE ??? TO CLIENT' statement.

#### **OnServerStart: TNotifyEvent**

This event is called just before a database request is sent to the server. Note: Spending too much time in event handler could lead to a performance hit in the current database request. For more information regarding this event, locate the index 'db\_register\_a\_callback' in SA/ASA online help.

#### **OnServerWait: TServerWaitEvent**

This event is called repeatedly while the database server or client library is busy processing your database request.

`TServerWaitEvent` = procedure(Sender: TObject; var AbortRequest: Boolean)  
AbortRequest allows the event handler to stop or break the current database request. Note that this event is subject to reentrancy. So be careful to avoid reentrant code while the server is busy processing your request. A call to `TAsaSession.Stop` or `TAsaSession.IsWorking` is the only safe methods to call in this context. Also note that spending too much time in the event handler could lead to a performance hit with the current database request. For more information regarding this event, locate the index 'db\_register\_a\_callback' in SA/ASA online help.

## **3.2 TAsaSQL**



This component is used to execute any SQL statement. Set the Session property to define the `TAsaSession` in which the SQL statements will execute. `TAsaSQL` have many easy-to-use methods for retrieving data with "select" statements, DML statements or stored procedure calls.

ActiveX note: The class interface is named `XAsaSQL`.

### **3.2.1 Properties**

**BlobSize: integer**

Used to control the size of the inline fetch buffer used to pre-fetch BLOBs. The default value zero (0) means that the SA BLOB functions will be used to fetch BLOBs, while a value greater than zero indicates that the SA BLOB functions will be used only when BlobSize isn't large enough to fetch the entire BLOB. This property is mainly used to increase the speed of fetch operations, and assume you can predefine the maximum size of the BLOB. If none of these reasons suites you, leave this property to zero (0).

ActiveX note: N/A

**CursorType: TAsaCursorType**

Specifies the SA/ASA generic cursor type to be used when opening cursors. Valid values are actDynamicScroll, actFixedScroll, actInsensitive, actNoScroll and actUnique. actDefault currently corresponds to the actDynamicScroll cursor but may change with future versions.

CursorType	Description
actDynamicScroll	Implements the ESQL DYNAMIC SCROLL cursor type. This cursor type is sensitive to table changes made by other cursors/users, either local or remote.
actFixedScroll	This cursor type conforms to the ESQL SCROLL cursor. Similar to the actDynamicScroll cursor, the actFixedScroll cursor is also sensitive to table changes made by other users, but will produce a "hole" in the cursor if another cursor/user deletes a row in the same table. Accessing data in this "hole" will produce a "-197 No current row of cursor" error. This behavior can be useful in circumstances when you want to inform the user that another user has deleted the row. In this case you could easily translate the "-197" error into a more descriptive message like "The information in this record has been deleted by another user!".
actInsensitive	An insensitive cursor has its membership fixed when it is opened. This cursor type conforms to the ODBC requirements for a static cursor. This cursor type is especially useful if you don't want other users commits to be immediately visible to the current user. An additional benefit from this cursor type is that it will allow a unidirectional cursor, obtained from a proxy-table, to scroll. If you ever experience a "-668: Cursor is restricted to FETCH NEXT operations" error, when a proxy-table is involved, use the actInsensitive cursor instead. The actInsensitive cursor conforms to the ESQL INSENSITIVE cursor type.
actNoScroll	This cursor is forward-only and conforms to the ESQL NO SCROLL cursortype. A forward-only cursor is also known as a unidirectional cursor.
actUnique	The ESQL UNIQUE cursor type will automatically include any column that is part of the primary key and make these columns available to the cursor to guarantee uniquely identifiably rows.

**IsPreparedOpen: boolean**

A readonly property indicating if a query statement is currently prepared with PrepareOpen.

**IsPreparedExecute: boolean**

A readonly property indicating if a SQL statement is currently prepared with PrepareExecute.

**LastErrorCode: integer**

Returns the low-level error code returned from SQL Anywhere after an unsuccessful database access call. Consider values greater than 0 (zero) as WARNINGS, while values less than 0 (zero) is ERRORS. 0 means SUCCESS.

**LastError: string**

Returns the low-level error message returned from SQL Anywhere after an unsuccessful database access call.



**MaxFetchCount: integer**

Used to control the maximum size of the fetch-buffer for readonly cursors. If the cursor contains one or more BLOBs or if ReadOnly = False, then a fetch-buffer of 1 is always used. The default fetch-buffer size is 25. Valid values are between 1 and 1000. Experimenting with this setting can dramatically improve the performance, depending on the particular cursor type. For prepared cursors, remember to set this property before the call to PrepareOpen. With non-prepared cursors, it's sufficient to set the size before the call to OpenRead.

**Opened: boolean**

A readonly property indicating if a cursor is currently open.

**RowCount: integer**

Public property that returns the accurate number of rows in the current open cursor. Accurate only if TAsaSession.WantRowCounts is enabled. If WantRowCounts is disabled it holds the estimated number of rows. This property is useful to size a progressbar or scrollbar after a cursor is opened successfully. RowCount is set right after a cursor is opened.

**RowPos: integer**

Public property that returns the current row position, starting from zero (0) of the current open cursor. RowPos is not guaranteed to be accurate unless TAsaSession.WantRowCounts is True.

**RowsProcessed: integer**

A readonly property to retrieve the number of rows processed by the last insert, update or delete SQL statement. Typically used after the last call to Execute.

**Session: TNativeSession**

Required property to link with a TAsaSession object.

ActiveX note: Link this property to an XAsaSession object instance.

**StillInProc: boolean**

Public property to return the current state of a running stored procedure. A return value of True indicates that the stored procedure is still executing but has returned because of a result set (a cursor). In this state the stored procedure can be resumed.

**TableName: string**

Used to return the name of the current active table. TAsaSQL extracts the tablename from the last successful opened query.

**TransIsolation: TTransIsolation**

Sets the locking isolation level individually for the cursor. See TAsaSession.TransIsolation for further details.

**WantExceptions: boolean**

Control if exceptions are raised or not. TAsaSQL raises an EAsaException for errors if this property is true. The default is false. The property does not affect critical errors, which always raises an exception. This property gives you a convenient way of avoiding the need to handle exceptions in large exception handlers, after the query returns. Simply having to check for a boolean result of True or False, after a OpenRead or OpenWrite, is very useful for lookup types of queries. Still, with more critical queries, setting WantException to True, better protects important sections of your code.

## 3.2.2 Methods

**constructor Create(AOwner: TComponent)**

Creates a TAsaSQL object instance.

ActiveX note: N/A

**destructor Destroy**

Destroys a TAsaSQL object instance.

ActiveX note: N/A

**function PrepareOpen(AQry: string): boolean**

Used to prepare the specified query. Should be used in combination with OpenPrepared, OpenReadPrepared or OpenWritePrepared.

**function Open(AQry: string; const VarValues: array of variant; ReadOnly: boolean): boolean**

Similar to OpenRead when ReadOnly = True, or OpenWrite when ReadOnly = False. But in contrast to these methods it will return True even with an empty result-set. The cursor will be positioned at the “crack” row before the first. So upon a successful return, use Next to fetch the first row.

**function OpenPrepared(const VarValues: array of variant; ReadOnly: boolean): boolean**

Similar to OpenReadPrepared when ReadOnly = True, or OpenWritePrepared when ReadOnly = False. But in contrast to these methods it will return True even with an empty result-set. The cursor will be positioned at the “crack” row before the first. So upon a successful return, use Next to fetch the first row.

**function OpenReadPrepared(const VarValues: array of variant): boolean**

Same as OpenRead, but requires the query to be prepared with PrepareOpen.

**function OpenWritePrepared(const VarValues: array of variant): boolean**

Same as OpenWrite, but requires the query to be prepared with PrepareOpen.

**function OpenRead(AQry: string; const VarValues: array of variant): boolean**

Opens a cursor for the specified query statement. AQry can also specify a stored function, but it must return a valid result-set. The result set is always read-only. Returns True if the result-set has a valid cursor and holds at least one row. Returns False in any other case. Upon a successful return, an implicit fetch Next operation is done and the cursor is positioned at the first row of the result-set. Use OpenRead as much as possible for better performance over OpenWrite. OpenRead utilizes an optimised array-fetch mechanism. If the result set is empty the return value is False, but the Opened property would still be True. When you use host variables, you pass the variable values in the VarValues parameter, in a 1 to 1 correspondence with the host variables in the query statement.

**function OpenWrite(AQry: string; const VarValues: array of variant): boolean**

Same as OpenRead, but the result set allows the Append, Delete or Modify calls to update the current row of the table.

**function Describe(AQry: string): boolean**

Describes a SQL statement. Follow this call to one of the GetField???? functions, to get field and schema information. A successful (True) return value indicates that the described statement represents a result set (cursor), while False means that the statement is a DML or DDL statement.

**procedure Close**

Closes the cursor opened by Open, OpenPrepared, OpenRead, OpenWrite, OpenReadPrepared and OpenWritePrepared. It's optional to call this method explicitly. It's called automatically whenever a new query is opened.

**procedure UnprepareOpen**

Drops the prepared statement as prepared by PrepareOpen.

**function PrepareExecute(ASQL: string): boolean**

Used to prepare the specified SQL statement. Should be used in combination with ExecutePrepared.

**function ExecutePrepared(const VarValues: array of variant): boolean**

Same as Execute, but requires that the SQL statement is prepared with PrepareExecute before executed.

**function Execute(ASQL: string; const VarValues: array of variant): boolean**

Executes any SQL statement. Returns true on success, false on failure. If WantExceptions is enabled, an EAsaException exception is raised. If no host variables are required, specify a Null variant value in the VarValues parameter (e.g NdbEmptyParam). When host variables are used, pass the variable values, in VarValues, in a 1 to 1 relationship to the host variables in the SQL statement.

**procedure UnprepareExecute**

Drops the prepared SQL statement as prepared by PrepareExecute.

**function GetOutVar(const AVarName: variant): variant**

Returns the current value of the specified host variable name. Typically used after a call to Execute or ExecutePrepared, to retrieve INOUT, OUT or any types of "execute-into" host variables. It is optional to identify the host variable by position, starting from 0 (zero), instead of name.

**function ExecuteImmediate(ASQL: string): boolean**

Execute any SQL statement that has NO host variables for INPUT and/or OUTPUT. This function utilizes the ESQL EXECUTE IMMEDIATE command. This method executes the statement without preparing it first. This method results in an error if your statement contains any host variables. If host variables are require, use Execute or ExecutePrepared instead.

**function Explain: string**

Used to retrieve a text specification of the optimization strategy used for the current open query (opened by OpenRead or OpenWrite). See the EXPLAIN statement (ESQL) in your SQL Anywhere Reference Manual. Also known as "the explain plan".

**function Resume: boolean**

Resumes the execution of a stored procedure after the processing of a result set (cursor) within the SP. See the RESUME statement (ESQL) in your SQL Anywhere Reference Manual.

**function First: boolean**

Navigates to the first row in the current open cursor. It's not necessary to call First right after OpenRead, OpenWrite, OpenReadPrepared or OpenWritePrepared. These methods position the cursor on the first row by default. Only allowed for scrollable cursors. Returns False if the result-set is empty.

**function Last: boolean**

Navigates to the last row in the result set. Only allowed for scrollable cursors. Returns False if the result-set is empty.

**function Curr: boolean**

Re-reads the current row in the result set.

**function Next: boolean**

Navigates to the next row in the result set. Returns False if an EOF state is detected and the EOF property will be updated accordingly.

**function Prev: boolean**

Navigates to the prior row in the result set. Only allowed for scrollable cursors. Returns False if a BOF state is detected and the BOF property will be updated accordingly.

**function MoveTo(ARowPos: integer): boolean**

Moves to the row given by ARowPos, starting from row zero (0). Only allowed for scrollable cursors.

**function MoveToFirst: boolean**

Moves the cursor to the “crack” row before the first row. Only allowed for scrollable cursors.

**function MoveToLast: boolean**

Moves the cursor to the “crack” row after the last row. Only allowed for scrollable cursors.

**function Lock: Boolean**

Places a write-lock on the current row, preventing others from updating the row. The lock will be held on the row until an explicit Commit or Rollback, or, if in AutoCommit mode, until the cursor is closed or updated.

**procedure Clear**

Clear all fields in the current row buffer. Each field is cleared (zeroed), and sets the NULL indicator.

**procedure ClearNotNull**

Clear all fields in the current row buffer. Each field is cleared (zeroed), and sets the NOT NULL indicator.

**procedure ClearField(const FieldName: string)**

Clears the specified field and sets the NULL indicator.

**procedure ClearFieldNotNull(const FieldName: string)**

Clears the specified field and sets the NOT NULL indicator.

**function Modify: boolean**

Updates the current row in the result set.

**function Append: boolean**

Appends the current row to the result set.

**function Delete: boolean**

Deletes the current row in the result set.

**function FindField(const FieldName: string): boolean**

Returns true if FieldName is found in the current open query.

**function GetFieldCount: integer**

Returns the number of fields in the current open query.

**function GetFieldName(const FieldNo: integer): string**

Returns the fieldname for FieldNo. FieldNo is the zero based field number in the current open cursor.

**function GetFieldType(const FieldName: variant): TNativeFieldTypes**

Returns the type of the field specified by FieldName. Valid types are (ntUnknown, ntString, ntInteger, ntInt64, ntFloat, ntCurrency, ntDate, ntTime, ntDateTime, ntBoolean, ntBinary, ntMemo, ntBlob). The TNativeFieldTypes are declared in the NdbBase unit, so when referencing field types, remember to include NdbBase in your uses clause. The FieldName parameter is declared a variant, to allow you to specify the string name of the field, or the zero based field number.

**function GetFieldSize(const FieldName: variant): integer**

Returns the data size of the field specified by FieldName. FieldName can be either the fieldname literal or the field reference number.

**function GetFieldLength(const FieldName: variant): integer**

Returns the display length of the field specified by FieldName. FieldName can be either the fieldname literal or the field reference number.

**function GetFieldPrecision(const FieldName: variant): integer**

Returns the numeric precision of NUMERIC and DECIMAL fields. FieldName can be either the fieldname literal or the field reference number.

**function GetFieldDecimals(const FieldName: variant): integer**

Returns the number of decimals for ntFloat or ntCurrency fieldtypes. FieldName can be either the fieldname literal or the field reference number.

**function GetFieldNulls(const FieldName: variant): boolean**

Returns True if a NULL value is allowed in the specified field. FieldName can be either the fieldname literal or the field reference number.

**function IsFieldNull(const FieldName: variant): boolean**

Returns True if the specified field has the database field-state of NULL. FieldName can be either the fieldname literal or the field reference number.

**function IsFieldDirty(const FieldName: variant): boolean**

Returns True if the specified field has been changed. FieldName can be either the fieldname literal or the field reference number.

**function IsDirty: boolean**

Returns True if ANY field in the current row has been changed.

**function CanModify: boolean**

Returns true if a valid cursor is opened by OpenWrite.

**function GetOldValue(const FieldName: variant): variant**

Returns the original value of the field before any new values was assigned to it. FieldName can be either the fieldname literal or the field reference number.

**function GetFieldValue(const FieldName: variant): variant**

This is the default class method used to retrieve field values. You don't need to call this directly. Use square brackets with the object name instead. (e.g. MyStr:=MySQL['first\_name']). FieldName can be either the fieldname literal or the field reference number.

**procedure SetFieldValue(const FieldName: variant; const Value: variant)**

This is the default class method to assign values to the specified field. You don't need to call this directly. Use square brackets with the object name instead. Assign a Null variant to a field to set it to the NULL state (e.g NdbEmptyParam).

**function StrToVarByte(StrBuf: PChar; BufLen: integer): variant**

Converts a character buffer of BufLen to a variant array of byte. Used when passing binary data to functions that require the variant array of byte datatype.

ActiveX note: N/A

**function StreamToVarByte(AStream: TMemoryStream): variant**

Converts a memory stream to a variant array of byte. Used when passing binary data to functions that require the variant array of byte datatype.

ActiveX note: N/A

**function StreamToStr(AStream: TMemoryStream): string**

Converts a memory stream to a string.

ActiveX note: N/A

**procedure StreamToBlob(const FieldName: variant; AStream: TStream)**

Assigns AStream to the specified BLOB field.  
ActiveX note: N/A

**procedure BlobToStream(const FieldName: variant; AStream: TStream)**

Assigns a BLOB field to AStream.  
ActiveX note: N/A

**procedure GraphicToBlob(const FieldName: variant; APicture: TPicture;  
AGraphicType: TNativeGraphicType)**

Assigns a Picture object, of the specified type, to a BLOB field. Valid options for the AGraphicType parameter are (gtBitmap, gtMetafile, gtIcon).  
ActiveX note: N/A  
Linux note: N/A

**procedure BlobToGraphic(const FieldName: variant; APicture: TPicture;  
AGraphicType: TNativeGraphicType)**

Assigns a BLOB field to a Picture of the specified type.  
ActiveX note: N/A  
Linux note: N/A

### 3.2.3 Events

**AfterOpen: TNotifyEvent**

Triggers right after a cursor is successfully opened.

**BeforeOpen: TNotifyEvent**

Triggers just before an attempt to open a cursor is made.

**AfterClose: TNotifyEvent**

Triggers right after a cursor is closed.

**BeforeClose: TNotifyEvent**

Triggers just before a cursor is closed.

**AfterPrepare: TNotifyEvent**

Triggers right after a statement has been prepared.

**AfterUnprepare: TNotifyEvent**

Triggers right after a statement has been unprepared.

## 3.3 TAsaLogin



This component provides a convenient method to connect to the database by using a standard login dialog box. The login dialog has built-in support for many SQL Anywhere specific connection features. When calling the Execute method, the connection properties of the linked TAsaSession are used as default. Similar, when the Execute method returns with a successful connection to the database, the current dialog settings are copied back to the TAsaSession component.

ActiveX note: N/A

### 3.3.1 Properties

**Session: TAsaSession**

A required property to link with a TAsaSession object. This specifies the session to use when connecting to the database.

### Options: TAsaLoginOptions

Specifies different options to control the behavior and appearance of the login dialog. Valid values include a combination of loAutoConnect, loAutoLibraryPath, loAllowDetails and loAlwaysDetails.

Options	Description
loAutoConnect	Will cause the Execute method to connect to the database without showing the dialog, if the connection attempt is successful.
loAutoLibraryPath	Will cause TAsaSession.LibraryPath to be assigned the extracted path of the selected database file, if any.
loAllowDetails	Will enable a "More" button on the login dialog to allow the user to enter additional connection parameters.
loAlwaysDetails	The login dialog will always show all the connection parameters.

## 3.3.2 Methods

### constructor Create(AOwner: TComponent)

Creates a TAsaLogin object instance.

### destructor Destroy

Destroys a TAsaLogin object instance.

### procedure Execute;

Executes the login dialog box. The Connected property of the linked TAsaSession will be True if the login attempt was successful.

```
procedure TForm1.btnLoginClick(Sender: TObject);
begin
    AsaLogin1.Options := [loAutoConnect, loAllowDetails];
    AsaLogin1.Execute;
end;
```

## 3.4 TAsaDataset



This is the component you use with Borland data-aware components. To support these components, you link a TAsaDataset instance to a TAsaSession with its TAsaDataset.Session property. Then you link a TDataSource to the TAsaDataset with its TDataSource.Dataset property. Finally, you link the data-ware components (i.e TDBGrid, TDBEdit) to the datasource with the Datasource property.

Consider TAsaDataset for smaller databases and simpler forms, while TAsaSQL will be the best choice when working with large databases.

TAsaDataset inherits all the properties and methods from TDataset. In addition some specialized properties and methods were added. Our goal is to make TAsaDataset as close as possible to the VCL component TQuery. In most of the cases, you should expect compatibility between the two.

ActiveX note: N/A

### 3.4.1 Properties

The following properties were added to those already inherited from TDataset:

#### DataSource: TDataSource

Used to extract current field values from another dataset, to use to bind otherwise unassigned parameters in the SQL statement in this component's SQL property. Typically used to define a master-detail relationship, between two tables.

**CursorType: TAsaCursorType**

See TAsaSQL.CursorType.

**IndexedLocate: Boolean**

Used to control if you want Locate and LocateNext to attempt to use a compatible index when searching. The default value is True. In situations when your current database collation is incompatible with the default system locale, it could be necessary to disable the indexed search.

**ParamCheck: Boolean**

Parameters are automatically build, based on the current query, if this property is true.

**Params: TNativeParams**

Used to manage parameters assigned to the dataset.

**ParamCount: Word**

Used to determine how many parameters are defined in the Params property. If the ParamCheck property is True, ParamCount always corresponds to the number of actual parameters in the SQL statement for the query. Same as Params.Count.

**Prepared: Boolean**

Determines whether or not the SQL statement is prepared for execution.

**ReadOnly: Boolean**

Indicates if the dataset allows editing.

**RowsAffected: Integer**

To determine how many rows were updated or deleted by the last query operation. If RowsAffected is zero, the query did not update or delete any rows.

**Session: TAsaSession**

A required property to link to a TAsaSession object.

**StillInProc: Boolean**

Public property to return the current state of a running stored procedure. A return value of True indicates that the stored procedure is still executing but has returned because of a result set (a cursor). In this state the stored procedure can be resumed.

**SyncInserts: Boolean**

This property is used to avoid the "lost inserts" affect in TAsaDataset. A row inserted into a cursor has no position in the result set, and will cause the row to disappear from view when using data-aware controls. When this property is set to True, it uses the primary key (if any) of the table to reposition the cursor to the inserted row (if it still matches the WHERE clause criteria in the refreshed result set) and it will also make any auto-incremental value or other column defaults immediately visible. If the table doesn't include any primary key, ALL fields are used to synchronize the inserted row with the new result set. The default value is False.

**SQL: TStrings**

Used to define the query to be executed.

### 3.4.2 Methods

The following methods were added to those already inherited from TDataset:

**procedure ExecSQL**

To execute the INSERT, UPDATE, or DELETE statement currently assigned to the SQL property. ExecSQL is also used to execute data definition statements.



**function FindKey(const KeyValues: array of const): Boolean;**

Use FindKey to search for a specific record in a dataset. KeyValues contains a comma-delimited array of field values, called a key. Each value in the key can be a literal, a variable, a null, or nil. If the number of values passed in KeyValues is less than the number of columns in the index used for the search, the missing values are assumed to be null. The key is matched against the index built up by the current ORDER BY clause assigned to the query statement. If a search is successful, FindKey positions the cursor on the matching record and returns True. Otherwise the cursor is not moved, and FindKey returns False.

**procedure FindNearest(const KeyValues: array of const);**

Use FindNearest to move the cursor to a specific record in a dataset or to the first record in the dataset that is greater than the values specified in the KeyValues parameter. KeyValues contains a comma-delimited array of field values, called a key. If the number of values passed in KeyValues is less than the number of columns in the index used for the search, the missing values are assumed to be null. The key is matched against the index built up by the current ORDER BY clause assigned to the query statement.

**function Locate(const KeyFields: string; const KeyValues: Variant; Options: TLocateOptions): Boolean**

To search a dataset for a specified record and makes that record the current record. Returns True if it finds a matching record, and makes that record the current one.

**function LocateNext(const KeyFields: string; const KeyValues: Variant; Options: TLocateOptions): Boolean**

To search a dataset for a record after the current cursor position. Returns True if it finds a matching record, and makes that record the current one.

**function Lookup(const KeyFields: string; const KeyValues: Variant; const ResultFields: string): Variant**

To retrieve field values from a record that matches specified search values.

**function ParamByName(const Value: string): TNativeParam**

To set or use parameter information for a specific parameter based on its name. Value is the name of the parameter for which to retrieve information. The return value is an instance of the TNativeParam class described in section 3.4 and 3.5.

**procedure Prepare**

Prepares the SQL statement for later execution.

**procedure Resume**

Resumes the execution of a stored procedure after the processing of a result set (cursor) within the SP. See also TAsaSQL.Resume.

**procedure UnPrepare**

Frees the resources and un-prepares the current prepared SQL statement.

### 3.4.3 Events

No additional events are added to the events derived from TDataset.

## 3.5 TAsaStoredProc



This is the component you use to access SA stored procedures in a Borland VCL/CLX data-aware manner. You can either use the component independently or connect it to a TDataSource component. TAsaStoredProc supports both DML based SP's or SP's that return one or more result sets.

TAsaStoredProc inherits all the properties and methods from TDataset. In addition some specialized properties and methods were added. Our goal is to make TAsaStoredProc as

close as possible to the VCL component TStoredProc. In most of the cases, you should expect compatibility between the two.

ActiveX note: N/A

### 3.5.1 Properties

The following properties were added to those already inherited from TDataset:

**Params: TNativeParams**

Used to manage parameters assigned to the stored procedure.

**ParamCount: Word**

Used to determine how many parameters are defined in the Params property. Same as Params.Count.

**Prepared: Boolean**

Determines whether or not the SQL statement is prepared for execution.

**Session: TAsaSession**

A required property to link with a TAsaSession object.

**StillInProc: boolean**

Public property to return the current state of a running stored procedure. A return value of True indicates that the stored procedure is still executing but has returned because of a result set (a cursor). In this state the stored procedure can be resumed.

**StoredProcName: string**

Used to specify the name of the stored procedure.

### 3.5.2 Methods

The following methods were added to those already inherited from TDataset:

**procedure CopyParams(Value: TNativeParams)**

Copies a stored procedure's parameters into another parameter list.

**procedure ExecProc**

Executes the stored procedure. If the SP returns a result set, use Open (Active=True) instead.

**function ParamByName(const Value: string): TNativeParam**

To set or use parameter information for a specific parameter based on its name. Value is the name of the parameter for which to retrieve information. The return value is an instance of the TNativeParam class described in section 3.4 and 3.5.

**procedure Prepare**

Prepares the stored procedure for later execution.

**procedure Resume**

Resumes the execution of a stored procedure after the processing of a result set (cursor) within the SP. Also see TAsaSQL.Resume.

**procedure UnPrepare**

Frees the resources and un-prepares the current prepared stored procedure.

### 3.5.3 Events

No additional events are added to the events derived from TDataset.

## 3.6 TAsaAlert



The TAsaAlert component is used in an application that needs to send or receive messages to/from other connected clients. TAsaAlert requires a set of server-side external procedures available by the product "NativeDB Alerts". Refer to our website or support Email service for further information about "NativeDB Alerts".

TAsaAlert executes in a separate working thread, in the background, without interfering with the normal program flow or main thread of your application. When a message is received the OnReceive event is called, which is synchronized with the application's main thread.

After dropping a TAsaAlert component on a form or by creating it in your application, link it to a TAsaSession by setting the Session property. To receive messages, set the SubscriptionType property, and specify one or more message names in the MessageNames property, separated by semicolons. Write an OnReceive event handler to react on incoming messages. If you want to know that no messages has arrived in a specified amount of time, set the Timeout property accordingly, and write an OnTimeout event handler. Also write an OnStopped event handler if you want know if the process waiting for incoming messages has stopped caused by an unsuccessful call to Start or as the result of an explicit call to Stop.

When calling the Start method of TAsaAlert, it will login using a duplicated TAsaSession and create the background working thread. Then it will start to wait for incoming messages. Each time a new message arrive the OnReceive event handler will be called. To stop the TAsaAlert component, call the Stop method. When a TAsaAlert component is destroyed, the Stop method is automatically called. The Stop method will also logoff the duplicated session and destroy the working thread.

To send a message, call the Send method of TAsaAlert, identify the message by the MsgName parameter and supply the message contents using the Msg and MsgParam parameters. If the message has currently no subscribers, an EAsaAlertException exception is raised.

### 3.6.1 Properties

#### MessageNames: string

The MessageNames property defines the name(s) of the message(s) to subscribe for or signal interest in. If you want to subscribe for multiple message-types, separate the names by semicolons. When one of multiple messages is received, the OnReceive event will provide the actual message-name.

#### SubscriptionType: TSubscriptionType

The SubscriptionType property should be considered in combination with the MessageNames property. Valid values are mtByNames, mtByUserID and mtByUserIDAndNames. Together with MessageNames it builds a final subscription request to be used when the Start method is called.

SubscriptionType	Message Subscription
mtByNames	The complete subscription request will include the MessageNames property only.
mtByUserID	The complete subscription request will include the Session.LoginUser only.
mtByUserIDAndNames	The complete subscription request will include both the Session.LoginUser and the MessageNames property.

#### Session: TAsaSession

A required property to link with a TAsaSession object. When calling the Start method this TAsaSession will be duplicated to avoid interference with other database access in your application. The calls to Clear and Send will use the original linked Session as is.

**Timeout: Integer**

Defines how many seconds to wait before a new message is received before the OnTimeout event handler is called. If the value is 0, the OnTimeout event will not occur.

**3.6.2 Methods****procedure Clear(const MsgName: string)**

Removes all messages from the message queue identified by a case insensitive MsgName parameter.

**function IsWaiting: Boolean**

Indicates if TAsaEvent is currently waiting for messages.

**procedure Send(const MsgName: string; const Msg: string; MsgParam: Integer)**

Posts a new message in the message queue identified by a case insensitive MsgName parameter. Msg and the optional MsgParam parameter define the actual contents of the message.

**procedure Start**

Start to wait for the incoming messages.

**function SubscribingFor: string**

Returns the final subscription request string depending on MessageNames, SubscriptionType and Session.LoginUser.

**procedure Stop**

Stops the TAsaAlert component from waiting for incoming messages.

**3.6.3 Events****OnReceive: TReceiveEvent**

```
type TReceiveEvent = procedure(Sender: TObject; const MsgName: String;
    const Msg: string; MsgParam: Integer) of object;
```

Called when a new message is received. The following is an example of a simple event handler that will list all incoming messages in a memo:

```
procedure TForm1.AsaAlert1Receive(Sender: TObject; const MsgName,
    Msg: String; MsgParam: Integer);
begin
    Memo1.Lines.Add('A message "' + Msg + '" identified by "' + MsgName +
        '" has arrived!');
end;
```

**OnStopped: TNotifyEvent**

Called as a result of an unsuccessful call to Start or when an explicit call to Stop is made.

**OnTimeout: TTimeoutEvent**

```
type TTimeoutEvent = procedure(Sender: TObject; var Continue: Boolean) of object;
```

Called when the number of seconds specified by the Timeout property has passed since the last message was received. Set the Continue parameter to False to stop the TAsaAlert component.

### 3.7 TNativeParam

When you access parameters or host variables through TAsaDataset the class type returned is an instance of TNativeParam. TNativeParam is designed very close to the familiar TParam VCL class, as used by the VCL TQuery component.

ActiveX note: N/A

#### 3.7.1 Properties

**AsBlob: string**

Specifies the value of the parameter when it represents a binary large object (BLOB) field.

**AsBoolean: boolean**

Specifies the value of the parameter when it represents a boolean field.

**AsCurrency: double**

Specifies the value of the parameter when it represents a field of type Currency.

**AsDate: TDateTime**

Specifies the value of the parameter when it represents a date field.

**AsDateTime: TDateTime**

Specifies the value of the parameter when it represents a date-time field.

**AsFloat: double**

Specifies the value of the parameter when it represents a float field.

**AsInteger: longint**

Specifies the value of the parameter when it represents an integer field.

**AsSmallInt: longint**

Specifies the value of the parameter when it represents a small integer field.

**AsMemo: string**

Specifies the value of the parameter when it represents a memo field.

**AsString: string**

Specifies the value of the parameter when it represents a string field.

**AsTime: TDateTime**

Specifies the value of the parameter when it represents a time field.

**AsWord: longint**

Specifies the value of the parameter when it represents a word field.

**Bound: Boolean**

Indicates if the parameter has been assigned a value.

**DataType: TNativeParamDataType**

Indicates the type of field whose value the parameter represents. TNativeParamDataType is based on the TNativeFieldTypes type. Valid types are (ntUnknown, ntString, ntInteger, ntInt64, ntFloat, ntCurrency, ntDate, ntTime, ntDateTime, ntBoolean, ntBinary, ntMemo, ntBlob). Parameters of SmallInt and Word are internally treated as an ntInteger.

**IsNull: boolean**

Indicates whether the value assigned to the parameter is NULL (blank).

**ParamType: TNativeParamType**

Indicates the parameter type used to bind host variables. Valid values are (nptUnknown, nptInput, nptInputOutput, nptOutput, nptResult).

**Text: string**

Represents the value of the parameter as a string. This property is the same as using AsString.

**Value: variant**

Represents the value of the parameter as a Variant.

### 3.7.2 Methods

**procedure Assign(Source: TPersistent)**

Used to assign another TNativeParam instance to this TNativeParam instance.

**procedure AssignField(Field: TField)**

Assigns a TField descendant's value and properties to this TNativeParam instance.

**procedure AssignFieldValue(Field: TField; const Value: Variant)**

Assigns a TField descendant's value to this TNativeParam instance.

**procedure Clear**

Assigns the NULL state indicator to this TNativeParam instance.

**procedure GetData(Buffer: Pointer)**

Returns the raw binary data currently assigned to the parameter.

**function GetDataSize: Integer**

Returns the size of the data currently assigned to the parameter.

**procedure LoadFromFile(const FileName: string; BlobType: TNativeParamBlobType)**

Loads the BLOB parameter with data from a file. TNativeParamBlobType is a subset of TNativeParamDataType and can be either set to ntMemo or ntBlob.

**procedure LoadFromStream(Stream: TStream; BlobType: TNativeParamBlobType)**

Loads the BLOB parameter with data from a stream. TNativeParamBlobType is a subset of TNativeParamDataType and can be either set to ntMemo or ntBlob.

**procedure SetBlobData(Buffer: Pointer; Size: Integer)**

Assigns raw binary data to the BLOB parameter.

**procedure SetData(Buffer: Pointer)**

Assigns raw binary data to the parameter.

### 3.8 TNativeParams

TNativeParams manages a list of TParam instances. To add or remove parameters from the list, use TNativeParams methods. TNativeParams is designed very close to the familiar TParams VCL class, as used by the VCL TQuery component.

ActiveX note: N/A

#### 3.8.1 Properties

**Items[Index: word]: TNativeParam**

Default class property. e.g. You can access the required parameter using [Index] only. The return value is an instance of TNativeParam.

### 3.8.2 Methods

**procedure Assign(Source: TPersistent)**

Used to assign another TNativeParams instance to this TNativeParams instance.

**procedure AddParam(Value: TNativeParam)**

Adds a parameter to the list.

**procedure RemoveParam(Value: TNativeParam)**

Removes a parameter from the list.

**function Count: Integer**

Returns the current number of parameters.

**procedure Clear**

Clears the list of parameters.

**function ParamByName(const Value: string)**

Returns an instance of TNativeParam as given by Value.

### 3.9 ENativeException

Derived from the base Exception class, ENativeException exceptions are raised for errors in base class operations.

ENativeException is only raised when TAsaSession.WantExceptions or TAsaSQL.WantExceptions are set to True.

ActiveX note: N/A

#### 3.9.1 Properties

The following properties were added to those already inherited from Exception:

**ErrorCode: integer**

Contain the error code that caused the exception. Zero (0) mean unspecified error.

### 3.10 EAsaException

Derived from ENativeException, EAsaException exceptions are raised for errors in SA specific database operations.

EAsaException is only raised when TAsaSession.WantExceptions or TAsaSQL.WantExceptions are set to True.

The ErrorCode property inherited from ENativeException contains the low-level database error code that caused the exception

ActiveX note: N/A

### 3.11 ENativeDatasetError

Derived from the base EDatabaseError class, ENativeDatasetError exceptions are raised for errors in the TAsaDataset class.

EDatabaseError exception is raised for errors in base class operations, while a ENativeDatasetError is raised for errors in SA specific database operations.

ActiveX note: N/A

The following exception handler, illustrates how to catch specific low level SA database error codes:

```
try
  AsaDataset1.Active := True;
except
  on E:ENativeDatasetError do
  begin
    if (E.ErrorCode = -101) then ShowMessage('NOT_CONNECTED!') else
    if (E.ErrorCode = -308) then ShowMessage('CONNECTION_DROP!') else
      raise;
    end;
  end;
end;
```

## 4. Component Usage

### 4.1 Working with TAsaSession

To connect to a SA database you need at least to set the LoginDatabase, LoginUser and LoginPassword. Once these properties are established, either run time or design time, you can connect to the database with the "Connected" property or "Open" method.

Windows note: When TAsaSession tries to connect to SA, it uses the LibraryPath to find the SA server engine and the interface libraries. This includes in general SA files in the win32\ directory. Which client interface library to use, is specified by the LibraryFile property. The need for the different SA library files depends on what connection type you are using. If LibraryPath is not specified, NativeDB uses the system path to locate the SA interface files. If you want to bundle SA with your application in a sub-directory controlled by your app., you should use the LibraryPath. If LibraryPath is empty, TAsaSession relies on the system PATH to find the client library files. The SA files you need, is a subset of the files in the SA Win32\ directory. If you want to know only the minimum required files, see the "SQL Anywhere User's Guide" or the "Online Help". In general, you need at least one of the server engine executables (e.g. dbeng7.exe) and a few DLLs.

Linux note: The SA library files (.so) is always attempted loaded through the Linux environment variable LD\_LIBRARY\_PATH or through the configuration file /etc/ld.so.conf. See the description on LibraryPath for further details.

#### 4.1.1 Personal engine connection

SA supports a personal database connection. This connection is meant to connect an application in a standalone environment. The personal connection will allow one PC only, and will not allow any PC clients (except of a maximum of 10 simultaneous connections on the same PC). To connect with a personal engine, you specify stServer as the ServerType before you try to connect (this means personal server).

```
AsaSession1: TAsaSession;
.
.
try
  AsaSession1.ServerParams := 'start=dbeng7.exe';
  AsaSession1.ServerType := stServer;
  AsaSession1.LoginDatabase := 'c:\MyApp\MyDB.db';
  AsaSession1.LoginUser := 'dba';
  AsaSession1.LoginPassword := 'sql';
  AsaSession1.Connected := True;
except
  on E: EAsaException do ShowMessage(E.Message);
end;
```



Running the personal engine in the Linux environment would require a coding similar to this:

```
AsaSession1: TAsaSession;
.
.
try
  AsaSession1.ServerParams := 'start=dbeng7';
  AsaSession1.ServerType := stServer;
  AsaSession1.LoginDatabase := '/home/frank/myapp/mydb.db';
  AsaSession1.LoginUser := 'dba';
  AsaSession1.LoginPassword := 'sql';
  AsaSession1.Connected := True;
except
  on E: EAsaException do ShowMessage(E.Message);
end;
```

#### 4.1.2 Network engine connection

To run your application as a SA network server, you specify stServer in the ServerType property and 'start=dbsrv7.exe' in the ServerParams property. In addition, you should also specify the network protocol or protocols to use. The SA engine name defaults to the file prefix of the database filename. If the LoginDatabase is "c:\MyApp\MyDB.db", the server engine is named "MyDB". To override this behavior you can set the LoginEngineName. From the client side, it's optional to use the database name without the path to it in the LoginDatabase property, but TAsaSession will do the naming extraction, if you specify the full database file path at the client side too.

```
AsaSession1: TAsaSession;
.
.
try
  AsaSession1.ServerParams := 'start=dbsrv7.exe -x tcpip';
  AsaSession1.ServerType := stServer;
  AsaSession1.LoginEngineName := 'MyDB'; // Optional
  AsaSession1.LoginDatabase := 'c:\MyApp\MyDB.db';
  AsaSession1.LoginUser := 'dba';
  AsaSession1.LoginPassword := 'sql';
  AsaSession1.Connected := True;
except
  on E: EAsaException do ShowMessage(E.Message);
end;
```

#### 4.1.3 Runtime engine connection

For small apps distributed to many users, the SA runtime is convenient. You connect to the runtime version of SA, the same way as with the personal engine.

```
AsaSession1: TAsaSession;
.
.
try
  AsaSession1.ServerParams := 'start=rteng7.exe';
  AsaSession1.ServerType := stServer;
  AsaSession1.LoginDatabase := 'c:\MyApp\MyDB.db';
  AsaSession1.LoginUser := 'dba';
  AsaSession1.LoginPassword := 'sql';
  AsaSession1.Connected := True;
except
  on E: EAsaException do ShowMessage(E.Message);
end;
```

#### 4.1.4 Client connection

When you do a client connection, a running SA server must be found in your immediate network or through a WAN using a "host=" parameter in the ClientParams property. The server could be either a dedicated server or it could be your application running SA as a network server. To connect to your server, you need to specify stClient in the ServerType

property, and the necessary connection parameters in the ClientParams property. In addition, you need to specify the database name either as the fully qualified database filename or only the prefix part of the file. In addition, you can also specify the database engine name in LoginEngineName. Typically, useful if the server holds more than one database in the same server engine.

```
AsaSession1: TAsaSession;
.
.
try
  AsaSession1.ClientParams := 'links=tcipip';
  AsaSession1.ServerType := stClient;
  AsaSession1.LoginEngineName := 'MyEngine';
  AsaSession1.LoginDatabase := 'MyDB';
  AsaSession1.LoginUser := 'dba';
  AsaSession1.LoginPassword := 'sql';
  AsaSession1.Connected := True;
except
  on E: EAsaException do ShowMessage(E.Message);
end;
```

If you want to connect to a SQL Anywhere 5 server you need to run the SQL Anywhere client executable as well. The following is an example of a client connection to SQL Anywhere 5:

```
AsaSession1: TAsaSession;
.
.
try
  AsaSession1.ClientParams := 'start=dbclient.exe -x tcipip';
  AsaSession1.LibraryFile := 'dbl50t.dll';
  AsaSession1.ServerType := stClient;
  AsaSession1.LoginEngineName := 'MyEngine';
  AsaSession1.LoginDatabase := 'MyDB';
  AsaSession1.LoginUser := 'dba';
  AsaSession1.LoginPassword := 'sql';
  AsaSession1.Connected := True;
except
  on E: EAsaException do ShowMessage(E.Message);
end;
```

If you want to connect to a SQL Anywhere 7 server (either running on Linux or Windows) using a Linux desktop machine, consider the following example:

```
AsaSession1: TAsaSession;
.
.
try
  AsaSession1.ClientParams := 'links=tcipip{host=<server IP address>}';
  AsaSession1.LibraryFile := 'libdblib7.so';
  AsaSession1.ServerType := stClient;
  AsaSession1.LoginEngineName := 'MyEngine';
  AsaSession1.LoginDatabase := 'MyDB';
  AsaSession1.LoginUser := 'dba';
  AsaSession1.LoginPassword := 'sql';
  AsaSession1.Connected := True;
except
  on E: EAsaException do ShowMessage(E.Message);
end;
```

#### 4.1.5 Workgroup connection

The workgroup connection occasionally decides which PC in the LAN that becomes the network server. First, it tries to be stClient. If this fails it tries to be stServer. This option is convenient for small LANs with a peer-to-peer network, with no dedicated server. If the application that occasionally becomes the server exits, the running SA server on that PC will not stop until the last client disconnects from it.

Be aware though, that this connection type will potentially run the server engine on a different computer than the one storing the database file. This could lead to a performance problem on

slow LANs with heavy network traffic. But the workgroup connection is still very convenient from a SW distribution point of view.

Also note, that this connection type should be used with a shared network drive or a UNC specified path to the database file. This is required to allow any computer that participates in the workgroup to become the server.

**WARNING!** Sybase does not recommend running the server engine and DB file on separate computers. The authors are not to be held responsible of any data loss due to a workgroup connection.

```
AsaSession1: TAsaSession;
.
.
try
  AsaSession1.ServerParams := 'start=dbsrv7.exe -x tcpip';
  AsaSession1.ServerType := stWorkgroup;
  AsaSession1.LoginEngineName := 'MyDB'; // Optional
  AsaSession1.LoginDatabase := '\\NTServer\c-drive\MyApp\MyDB.db';
  AsaSession1.LoginUser := 'dba';
  AsaSession1.LoginPassword := 'sql';
  AsaSession1.Connected := True;
except
  on E: EAsaException do ShowMessage(E.Message);
end;
```

#### 4.1.6 Transaction handling

Every database application should have a clear strategy in how to handle transaction isolation levels, commits and rollbacks.

Uncommitted updates will potentially block other clients from updating rows fetched and updated by the client doing the updates. It could block other clients, i.e. they seem to "freeze" forever, until the client, that placed the lock, commits the update. The blocking behavior can be controlled by the "BLOCKING" database option.

Due to this behavior, consider "TAsaSession.Autocommit=True", or design your database updates in tight blocks, i.e.

```
// Autocommit is False
try
  // Do database updates here using:
  // SQL.Append, SQL.Modify, SQL.Delete or SQL.Execute
  // These methods could potentially block other clients!
finally
  AsaSession1.Commit; // Force a commit. This will unblock other clients
end;
```

Dataaccess combined with the use of the Borland data-aware components (TDBEdit, TDBGrid), using "TAsaSession.Autocommit=True" could be a convenient strategy.

In other cases when you need to group SQL statements together in a single user transaction, the following code construct will handle this:

```
AsaSession1.StartTransaction;
try
  // Do database updates here with TAsaSQL or TAsaDataset or both.
  // Remember to enable WantExceptions
  AsaSession1.Commit; // Commit the changes on success
except
  AsaSession1.Rollback; // Undo changes on failure
  raise;
end;
```

If one client blocks another client, while it waits for it to commit, you can define a OnServerWait event, which will be repeatedly called by the database engine, while it waits. In this event, you can give the user a chance to stop the database request, or display an option to "Abort, Retry, Ignore".

```

procedure TForm1.AsaSession1ServerWait(Sender: TObject; var AbortRequest: Boolean);
begin
    Application.ProcessMessages; // Let user press a STOP button
end;

procedure TForm1.btnStopClick(Sender: TObject);
begin
    AsaSession1.Stop;
end;

```

### 4.1.7 Backing up the database

The TAsaSession.Backup method is useful to take a full online database backup or an incremental backup by renaming and restarting the transaction log. The backup includes the main database file, the transaction log and the mirror log file if any. To backup you must first be connected to the database.

The backup method returns true if it backed up the database successfully. The first parameter indicates where the backup should be stored. The second parameter allows you to handle the transaction log file in different ways (refer to TAsaSession.Backup for a description on each available backup type).

An additional feature includes the usage of the TAsaSession.OnBackup event to display progress information while the backup executes.

```

procedure TForm1.BackupClick(Sender: TObject);
begin
    AsaSession1.OnBackup := AsaSession1Backup;
    if not AsaSession1.Backup('c:\MyApp\BackupDir', btFull) then
        ShowMessage(SQL.LastError);
end;

```

The code in the OnBackup event used to update a progress bar could be:

```

procedure TForm1.AsaSession1Backup(Sender: TObject; BackupFile: String; CurrPage,
    NumOfPages: Integer; var AbortBackup: Boolean);
begin
    Caption := BackupFile; // Includes both file and path
    ProgressBar1.Max := NumOfPages;
    ProgressBar1.Position := CurrPage;
end;

```

### 4.1.8 Server messages

Introduced with ASA version 6.01, the MESSAGE statement was improved to support the "TO CLIENT" extended syntax. All server messages with this syntax are sent to the TAsaSession.OnServerMessage event. When messages are sent to the client, a properly defined OnServerMessage handler should check the MessageType parameter. This parameter specifies the importance level of the message. Messages in the smtInfo and smtStatus class are notification messages, while smtWarning and smtAction messages, signal more important messages.

The following stored procedure notifies the client with an information message:

```

create procedure dba.notify_client()
begin
    message 'Hello NativeDB!' type info to client
end

```

When executed ASA triggers a callback, and the incoming message could be handled in the following event:

```

procedure TForm1.AsaSession1ServerMessage(Sender: TObject; MessageType:
    TServerMessageType; MessageCode: Integer; Msg: String);
begin
    case MessageType of
        smtInfo: lbInfoListbox.Items.Add('Info message: ' + Msg);
        smtWarning: MessageDlg(Msg, mtWarning, [mbOk], 0);
        smtAction: MessageDlg(Msg, mtInformation, [mbOk], 0);
    end;

```

```

    smtStatus: lbStatListbox.Items.Add('Status message: ' + Msg);
end;
end;

```

The above event handler could also be an interesting starting point for a simple SQL Monitor. By sending client messages from the client itself, one can trace queries as they are sent to the database engine. The following event handler is attached to the TAsaDataset.AfterOpen event, and uses a TAsaSQL instance to supply messages to the client. This sample also shows the strength of TAsaSQL and TAsaDataset, when used together.

```

procedure TForm1.AsaDataset1AfterOpen(DataSet: TDataSet);
begin
    AsaSQL1.Execute('message :stmt type info to client', [(DataSet as
TAsaDataset).SQL.Text]);
end;

```

The event implements a general handler and could easily be attached to every TAsaDataset instance throughout the application.

Client messages could also be a convenient way of setting up a communication link to the current TAsaSession object, in cases when you don't have access to its parent form. In this way, you could simulate the traditional PostMessage and SendMessage Win32 API calls, to communicate with the TAsaSession instance. Further, you could define your own set of commands, and send them as messages to be executed on a global level.

#### 4.1.9 Multi-threading

Many times an application needs to use multiple threads. These threads might also access the database. In cases of multi-tier server applications or when you don't want the user to be locked-up by a long running database operation, multiple threads are a convenient resource. A second thread is also convenient when you want to break a long running database request. But note that a long running database request can also be stopped, in a single-threaded environment, through the server message event OnServerWait.

There are typically two types of configurations for a multi-threaded application.

##### Multiple threads, using a single database session

In this configuration, one singleTAsaSession instance is accessed by multiple threads. Because the SA SQLCA connection handle can only make one database request at the same time, all simultaneous database access must be serialized. When you use a single connection to be accessed by multiple threads, executing SQL statements at the same time, you are restricted to one active request at a time, per connection. Threads can be serialized by using the Win32 thread synchronization API's. Semaphores, events and critical sections are all objects to control thread synchronization.

When threads are synchronized, they block each other when they are accessing the database. In this configuration, additional session synchronization must also take place, when threads are connecting and disconnecting to and from the server.

##### Multiple threads, using multiple database sessions

With this configuration, each thread uses its own separate TAsaSession instance. The client and server can process requests for these threads simultaneously. The result is an optimized use of resources and a non-blocking behavior in the client application.

In both configurations, access to the same TAsaSQL and TAsaDataset instances must always be restricted to a single thread.

Linux note: A thread safe version of the DBLIB client library is available with ASA for Linux. Consider the below replacement versions of DBLIB, if multiple threads and thread-safety is an issue in your Linux/Kylix application.

Product version	Thread-safe client libraries
Adaptive Server Anywhere 6 (ASA6 – Linux)	libdblib6_r.so

Product version	Thread-safe client libraries
Adaptive Server Anywhere 7 (ASA7 – Linux)	libdblib7_r.so
Adaptive Server Anywhere 8 (ASA8 – Linux)	libdblib8_r.so

ActiveX note: The COM classes XAsaSession and XAsaSQL uses the “apartment” threading model.

## 4.2 Working with TAsaSQL

After creating an instance of TAsaSQL, you must assign a valid TAsaSession object to its Session property.

Runtime, you can specify:

```
AsaSQL1: TAsaSQL;

AsaSQL1.Session := AsaSession1;
AsaSQL1.WantExceptions := True;
```

At design time, you use the published Session property to do this assignment.

### 4.2.1 Queries

The TAsaSQL component is typically used to query data from your database. To do this you would typically use either the OpenRead or OpenWrite methods.

```
Listbox1.Clear;
if AsaSQL1.OpenRead('select * from customer where city=:city', ['New York']) then
begin
  repeat
    Listbox1.Items.Add(AsaSQL1['fname'] + ' ' + AsaSQL1['lname']);
  until not AsaSQL1.Next;
end else
  ShowMessage(AsaSQL1.LastError);
```

In this example, exceptions are turned off, so we rely on the boolean return value to indicate success or failure. If the query fails, we display a message box showing the SQL.LastError string.

In the last parameter to OpenRead, we specify the host variable value to replace with the ":city" host indicator. This is optional. We could build "New York" directly into the string passed to OpenRead, and then passed a Null variant in the last parameter (e.g NdbEmptyParam or square brackets).

Note: It's also optional to use question marks as host variable indicators:

```
if AsaSQL1.OpenRead('select * from customer where city=?', ['New York']) then
```

As the example shows, to fill up the listbox, we retrieve the field values from each row using the SQL object's default property (specified by square brackets). This will return the field values as variant types, which in this case consists of strings.

### 4.2.2 Datatypes

NativeDB supports the following datatypes, with the corresponding database and variant datatype listed in the table. These enumerated constants are defined in the NdbBase unit.

Datatype	ASA datatype	Variant type
ntUnknown	<unspecified type>	varNull
ntString	char, varchar	varString
ntInteger	integer, smallint, tinyint, unsigned int, unsigned smallint	varInteger
ntInt64	bigint, unsigned bigint	varInt64
ntFloat	float, double, decimal, numeric, bigint	varDouble

ntCurrency	numeric(10,4), numeric (19,4) or numeric (20,4)	varDouble
ntDate	date	varDate
ntTime	time	varDate
ntDateTime	timestamp	varDate
ntBoolean	bit	varBoolean
ntBinary	binary, varbinary	varByte
ntMemo	long varchar	varString
ntBlob	long binary	varByte

To access field data, for both reading and writing, you use TAsaSQL's default property. The default property is TAsaSQL.FieldValues that you access with square brackets, i.e. SQL['myfield'].

It's the programmer's responsibility to make sure that the left-hand side is assignment compatible with the right-hand side of the expression.

```
var
  S: string;
  I: integer;
begin
  S := AsaSQL1['AStringField'];    // Correct
  I := AsaSQL1['AStringField'];    // Wrong
end;
```

The last assignment results in a "Variant type conversion error".

### 4.2.3 Binary data

The datatype ntBinary is used for binary data with a maximum of 32K in size. The datatype represents SA's binary or varbinary type, and are accessed as a zero-based variant array of bytes (varByte). To access this byte-array with highest possible performance, use the VarArrayLock and VarArrayUnlock functions. To check the size of the value you can use either the VarArrayHighBound function or TAsaSQL.GetFieldLength.

The examples assume that the TAsaSQL object is opened properly with either OpenRead or OpenWrite.

```
var
  BinData: variant;
  PBytes: PByte;
  Size: integer;
begin
  BinData := AsaSQL1['binary_field'];
  Size := VarArrayHighBound(BinData, 1) + 1;
  PBytes := VarArrayLock(BinData);

  // Use PBytes here

  VarArrayUnlock(BinData);
```

... or the short version

```
var
  PBytes: PByte;
begin
  PBytes := VarArrayLock(AsaSQL1['binary_field']);
  // Use PBytes here
```

... or even shorter without VarArrayLock, accepting slower performance

```
var
  TheFirstByte: byte;
begin
  TheFirstByte := Byte(AsaSQL1['binary_field'][0]);
```

To write to a binary field, just assign a variant array of byte to AsaSQL1['binary\_field'].

```

var
  BinData: variant;
  Size: integer;
begin
  Size := AsaSQL1.GetFieldLength('binary_field');
  BinData := VarArrayCreate([0, Size - 1], varByte);
  BinData[0] := ord('A');
  BinData[1] := ord('B');
  BinData[2] := ord('C');
  AsaSQL1['binary_field'] := BinData;

```

To post the changes to the underlying table you must call one of the methods to update or insert the row.

If you prefer to work with PChar's or other buffered binary data types instead, the TAsaSQL.StrToVarByte will do the conversion to a "variant array of byte" automatically for you. The following code shows how to utilize this function to modify a table with a binary field:

```

var
  Bin_Data: array[0..3] of byte;
begin
  if AsaSQL1.OpenWrite('select * from atable where id=100', []) then
    begin
      Bin_Data[0] := 6;
      Bin_Data[1] := 16;
      Bin_Data[2] := 12;
      Bin_Data[3] := 18;
      AsaSQL1['binary_field'] := AsaSQL1.StrToVarByte(@Bin_Data, 4);
      if not AsaSQL1.Modify then
        ShowMessage(AsaSQL1.LastError);
      end else
        ShowMessage(AsaSQL1.LastError);
    end;
end;

```

## 4.2.4 Binary Large Objects (BLOBS)

A BLOB (Binary Large Object) is a convenient data type to store general-purpose data of a large number of bytes. Bitmaps and documents are example of such data types. To access BLOB fields with NativeDB you can use direct field-to-field access, or use some of the BLOB helper methods.

For example to read a picture stored in BLOB field named "picture", you can call:

```

AsaSQL1.BlobToGraphic('picture', Image1.Picture, gtBitmap);
Image1.Picture.SaveToFile('c:\pictures\picture.bmp');

```

... to update a picture, loaded from a file, you can call:

```

Image1.Picture.LoadFromFile('c:\pictures\picture.bmp');
AsaSQL1.GraphicToBlob('picture', Image1.Picture, gtBitmap);

```

To read a long text memo from a ntMemo field, you simply call:

```

Memo1.Text := AsaSQL1['a_memo'];

```

... or to update the same memo you can call:

```

AsaSQL1['a_memo'] := Memo1.Text;

```

You can also optionally use the BLOB helper methods BlobToStream and StreamToBlob to access a memo or an image field.

To post the changes to the underlying table you must call one of the methods to update or insert the row.

All examples assume that a valid cursor is already obtained.



## 4.2.5 Handling NULL values

Un-assigned fields in a table will typically default to the NULL SQL state, unless a default value is given to the field by the database engine.

When the cursor is positioned on a valid row, you can check the current field's NULL state by using the `IsFieldNull` method. Pass the field name as the parameter to this method:

```
if AsaSQL1.OpenRead('select * from employee where emp_id=100', []) then
begin
    if not AsaSQL1.IsFieldNull('birth_date') then
        Caption := DateToStr(AsaSQL1['birth_date'])
    else
        Caption := 'NULL';
end else
    ShowMessage(AsaSQL1.LastError);
```

To append a new row to a table, assigning only a few fields, you can use the following:

```
if AsaSQL1.OpenWrite('select * from employee', []) then
begin
    AsaSQL1.Clear;                                // Set all field buffers to NULL
    AsaSQL1['emp_fname'] := 'Frank';
    AsaSQL1['emp_lname'] := 'Johnsen';
    if not AsaSQL1.Append then
        ShowMessage(AsaSQL1.LastError);
end else
    ShowMessage(AsaSQL1.LastError);
```

The un-assigned fields will potentially be given a default value as defined in your database (e.g. auto-incremental fields).

To change a field from a value to the NULL state, assign a Null variant to the field:

```
if AsaSQL1.OpenWrite('select * from employee', []) then
begin
    AsaSQL1['emp_lname'] := Null;
    if not AsaSQL1.Modify then
        ShowMessage(AsaSQL1.LastError);
end else
    ShowMessage(AsaSQL1.LastError);
```

If you need to check if a given field is allowed to be NULL, use the `GetFieldNulls` method, passing the fieldname as a parameter. A database error will return if you try to assign (and update) a NULL value to a field, which don't allow NULLS.

## 4.2.6 Appending rows

To append new records to a table you need to get a write-able result set first. This is done by a call to `OpenWrite` in contrast to `OpenRead`. When a record is appended, the current row's field values are assigned to the new row. You must explicitly call `Clear` to clear the fields. In the following example, `WantExceptions` is turned off, so we can properly check the boolean return value of `Append` as an indicator of success or failure. Note the use of the `Clear` method to empty the current row buffer before assigning any new values. In many cases, it can be useful to avoid the call to `Clear`, if you want to duplicate many rows, changing only a few fields in a loop.

```
AsaSQL1.OpenWrite('select * from customer', []);
if AsaSQL1.Opened then
begin
    AsaSQL1.Clear;                                // Set all columns to the NULL state
    AsaSQL1['fname'] := 'Frank';
    AsaSQL1['lname'] := 'Johnsen';
    if not AsaSQL1.Append then
        ShowMessage(AsaSQL1.LastError);
end else
    ShowMessage(AsaSQL1.LastError);
```

## 4.2.7 Appending rows containing a auto-incremental column

When a row is appended to the database, columns with NULL states will be added to the database as NULL or by their database default value, if any. An example of a column default value is the auto-incremental type.

To let the SA database engine use the default value, we must ensure that the column initial state is NULL, before the call to Append is made. If the state is non-NULL the value currently assigned to the column will be used.

If the SA database engine uses the default value and assigns the next incremental value to the column, this value is immediately available in the field buffer, without any need to refresh the query, or execute a separate "select @@identity" query. This feature also applies to other type of column-defaults, as well.

The following example uses the customer table in the "asademo" database, to show how auto-incremental values are available to the client, after a call to Append. The customer table defines a field "id" with the auto-incremental default value.

```
AsaSQL1.OpenWrite('select * from customer', []);
if AsaSQL1.Opened then
begin
    AsaSQL1.Clear;                                // Set all columns to the NULL state
    AsaSQL1['fname'] := 'Frank';
    AsaSQL1['lname'] := 'Johnsen';
    if not AsaSQL1.Append then
        ShowMessage(AsaSQL1.LastError)
    else
        ShowMessage('Autoinc value is: ' + IntToStr(AsaSQL1['id']));
end else
    ShowMessage(AsaSQL1.LastError);
```

## 4.2.8 Modifying rows

To modify a record in a table you need to get a write-able result set first. This is done by a call to OpenWrite in contrast to OpenRead.

```
if AsaSQL1.OpenWrite('select * from customer where lname='Johansen'', []) then
begin
    AsaSQL1['lname'] := 'Johnsen';
    if not AsaSQL1.Modify then
        ShowMessage(AsaSQL1.LastError);
end else
    ShowMessage(AsaSQL1.LastError);
```

The current row is the row being modified. You can use the navigational methods (First, Next etc) to make another row, the current.

## 4.2.9 Deleting rows

To delete a record in a table call Delete after a successful call to OpenWrite. This will delete the current row in the above result set.

```
if AsaSQL1.OpenWrite('select * from customer where lname='Johansen'', []) then
begin
    if not AsaSQL1.Delete then
        ShowMessage(AsaSQL1.LastError);
end else
    ShowMessage(AsaSQL1.LastError);
```

## 4.2.10 Executing SQL statements

You can achieve the same result using the Execute method instead of the Append, Modify and Delete methods, except that Execute is not aware of a current row in the result set.

So, to delete a row using Execute you could use the following code:

```
if not AsaSQL1.Execute('delete from customer where fname='John'', []) then
    ShowMessage(AsaSQL1.LastError);
```

With parameters:

```
if not AsaSQL1.Execute('delete from customer where fname=?, ['John']) then
    ShowMessage(AsaSQL1.LastError);
```

In fact, you can use any valid SQL statement, with the Execute method.

Using the second, you can supply host variable values for both input and output.

```
var
    AFirstName: string;

if not AsaSQL1.Execute('delete from customer where fname=?, [AFirstName]) then
    ShowMessage(AsaSQL1.LastError);
```

If you have more than one parameter to bind, separate each value with a comma in the variant array parameter:

```
if not AsaSQL1.Execute('delete from customer where fname=? and lname=?', [AFirstName,
ALastName]) then
    ShowMessage(AsaSQL1.LastError);
```

#### 4.2.11 Using parameters

Using host variables with SQL statements is convenient when using the same statement more than once. Using parameters instead of fixed statements also have some performance benefits with the SA server.

When you use host variables with NativeDB, you are responsible of passing the same number of values as the number of parameters contained in the SQL statement, in a 1 to 1 relationship from left to right.

When you use parameters with your queries, you pass the host variable values in the VarValues open array parameter to OpenRead, OpenWrite, OpenReadPrepared and OpenWritePrepared.

```
if AsaSQL1.OpenRead('select * from customer where fname=:p1 and lname=:p2', ['John',
'Smith']) then
```

When you use parameters with to your DML statements, you pass the values with the Execute and ExecutePrepared methods.

```
if AsaSQL1.Execute('delete from customer where lname=:p1', ['Smith']) then
```

The following table shows the SA datatypes and the corresponding variant types.

ASA type	Variant type
NULL	varNull
char, varchar	varString
integer, smallint, tinyint, unsigned int, unsigned smallint	varInteger
float, double, decimal, numeric, bigint, unsigned bigint	varDouble
numeric(10,4), numeric(19,4) or numeric(20,4)	varCurrency or varDouble
date	varDate
time	varDate
timestamp	varDate
bit	varBoolean
binary, varbinary	varByte
long varchar	varString
long binary	varByte

For example, if you bind a timestamp value to a parameter for a timestamp field, you can pass a TDateTime variable.

```

var
  bdate: TDateTime;
begin
  AsaSQL1.Execute('insert into employee (birth_date) values (?)', [bdate]);
end;

```

Note: It's optional to use a question mark ("?",) instead of host variable name. Also note that when passing strings as host variable values, you can use fixed strings, string variables or general string types.

The following example uses parameters/host variables to update a BLOB field:

```

var
  AStream: TMemoryStream;
  LongBinaryData: variant;
  LongVarcharData: variant;
begin
  AStream := TMemoryStream.Create;
  try
    AStream.LoadFromFile('c:\mypics\sister.bmp');
    LongBinaryData := AsaSQL1.StreamToVarByte(AStream);
    AStream.LoadFromFile('c:\mydocs\sister.doc');
    LongVarcharData := AsaSQL1.StreamToStr(AStream);
    AsaSQL1.Execute('update family_album set picture=:p1, document=:p2 where id=:id',
  [LongBinaryData, LongVarcharData, 1]);
  finally
    AStream.Free;
  end;
end;

```

## 4.2.12 Using prepared statements

The OpenRead, OpenWrite and Execute family of methods are very powerful. These methods, wraps the "prepare-action-unprepare" behavior into one single call. In some situations, you may want to split up this behavior, and control the preparation, execution/action and un-preparation statements separately.

A typical usage of prepared SQL statements and queries are to do the preparation in your application initialization code. Use the action methods with your regular program code flow, and finally un-prepare the statements in your application exit code.

The preparation family of methods consists of PrepareOpen and PrepareExecute. The action methods used together with these methods are OpenReadPrepared, OpenWritePrepared and ExecutePrepared. To un-prepare the statements at program exit, you use UnprepareOpen and UnprepareExecute. Note: If these methods are not called upon exit, they are implicit called internally by NativeDB.

To check if a statement is already prepared, you use the IsPreparedOpen or IsPreparedExecute read-only properties.

After a statement is prepared, it stays prepared throughout the lifetime of your TAsaSQL object, or until an explicit call is made to one of the unprepare methods.

The following code sample shows a typical usage of prepared statements.

```

procedure TForm1.FormCreate(Sender: TObject);
begin
  AsaSQL1.PrepareOpen('select * from customer where id=:id');
  AsaSQL2.PrepareExecute('insert into customer (id, fname, lname) values (?, ?, ?)');
end;

procedure TForm1.btnActionClick(Sender: TObject);
begin
  if AsaSQL1.IsPreparedOpen and
    AsaSQL1.OpenReadPrepared([StrToInt(Edit1.Text)]) then
  begin
    Listbox1.Items.Add(AsaSQL1['fname'] + ' ' + AsaSQL1['lname']);
    if AsaSQL2.IsPreparedExecute then
      if not AsaSQL2.ExecutePrepared([StrToInt(Edit2.Text), Edit3.Text, Edit4.Text])
  then

```

```

        ShowMessage(AsaSQL2.LastError);
    end;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    AsaSQL1.UnprepareOpen;
    AsaSQL2.UnprepareExecute;
end;

```

### 4.2.13 Calling stored procedures

Stored procedures (SPs) are procedures or functions stored in your database, and processed by the server engine. To call SPs, you use either `Execute` or `ExecutePrepared`. If the SP returns a cursor, you use `OpenRead` or `OpenReadPrepared`. To retrieve values returned by OUT or INOUT parameters, you use the `GetOutVar` method, after the execution of the SP.

Assume a SA stored function to return the full name of a customer:

```

create function fullname(in firstname char(30), in lastname char(30)) returns char(61)
begin
    declare name char(61);
    set name = firstname || ' ' || lastname;
    return(name);
end

```

You can call this function, with the following code:

```

begin
    if AsaSQL1.OpenRead('select * from customer where lname=?', ['Smith']) and
        AsaSQL1.Execute('?=call fullname(?,?)', [AsaSQL1['fname'], AsaSQL1['lname']]) then
        ShowMessage('Name: ' + AsaSQL1.GetOutVar('fullname'));
    end;

```

Note that in the above code its required to use a host variable assignment for functions that return a single value. e.g. `'? = call ....'` or `'a_result = call ....'`. Upon the return from the stored function use the NAME of the function to retrieve the returned value.

Stored procedures can supply both IN and OUT parameters (and INOUT). Assume the procedure:

```

create procedure greater(in a integer, in b integer, out c integer)
begin
    if a>b then
        set c=a
    else
        set c=b
    end if
end

```

You can call it with:

```

if AsaSQL1.Execute('call greater(?,?,?)', [10,20]) then
    ShowMessage(FloatToStr(AsaSQL1.GetOutVar('c')));

```

You only need to supply host variable values for the IN and INOUT parameters. If the stored procedure you are calling returns a result set, use the `OpenRead` or `OpenReadPrepared` method instead. Your SA database includes many system procedures. To retrieve useful information about the current connection you can use the following stored procedure:

```

if AsaSQL1.OpenRead('call sa_db_info', []) then
repeat
    // Read the fields here
until not AsaSQL1.Next;

```

If you want to call a stored procedure the same way as "Interactive SQL" you could:

```

if AsaSQL1.Execute('create variable c int', []) then
begin
  if AsaSQL1.Execute('call greater(100,150,c)', []) and
    AsaSQL1.OpenRead('select c', []) then
    ShowMessage(IntToStr(AsaSQL1['c']));
  AsaSQL1.Execute('drop variable c', []);
end;

```

The following sample shows how you use the GetOutVar method to retrieve values for INOUT, OUT and function result variables:

```

create procedure sp_get_customer(in i_id integer, out o_fname char(15), out o_lname
char(20), out o_city char(20))
begin
  select fname, lname, city
  into o_fname, o_lname, o_city
  from customer
  where id = a_id
end

```

You can call this SP from your application with the following code:

```

if AsaSQL1.Execute('call sp_get_customer(?,?,?,?)', [100]) then
begin
  edCust_fname.Text := AsaSQL1.GetOutVar('r_fname');
  edCust_fname.Text := AsaSQL1.GetOutVar('r_lname');
  edCust_fname.Text := AsaSQL1.GetOutVar('r_last_updated');
end else
  ShowMessage(AsaSQL1.LastError);

```

A stored procedure can return multiple result sets, or code that follows the first result set. To be able to continue execution of a SP after the processing of cursor, the Resume method is used:

```

create procedure multi_cust()
begin
  select id,fname from customer order by id asc;
  select id,lname,address from customer order by id asc
end

```

This SP returns multiple result sets. To handle this properly you can use the following code:

```

Listbox1.Clear;
if AsaSQL1.OpenRead('call multi_cust', []) then
begin
  repeat
    repeat
      S := '';
      for I := 0 to AsaSQL1.GetFieldCount - 1 do
        S := S + VarToStr(AsaSQL1 [I]) + ' ';
      Listbox1.Items.Add(S);
    until not AsaSQL1.Next;
  until not AsaSQL1.Resume;
end else
  ShowMessage(AsaSQL1.LastError);

```

The following SP returns a result set depending on a condition, which is passed to the SP as a parameter.

```

create procedure conditional_cust(in who integer)
begin
  if who = 1 then
    select id,fname from customer order by id asc
  else
    select id,lname,address from customer order by id asc
  end if
end

```

And the code to call it:

```

Listbox1.Clear;
if AsaSQL1.OpenRead('call conditional_cust(?)', [StrToInt(Edit1.Text)]) then

```

```

begin
  repeat
    Listbox1.Items.Add(IntToStr(AsaSQL1 ['id']));
  until not AsaSQL1.Next;
  AsaSQL1.Resume;
end else
  ShowMessage(AsaSQL1.LastError);

```

#### 4.2.14 Using compound statements

In contrast to stored procedures, compound statements are executed "on the fly" by the client application. Compound statements can also be used in batches or groups of operations. A compound statement starts with the keyword BEGIN and ends with the keyword END. In cases where you don't want to write stored procedures, groups of statements can be very convenient.

```

ABatch := 'begin' +
  ' declare cust_id int;' +
  ' set cust_id = 101;' +
  ' select * from customer where id=cust_id;' +
  'end';
AsaSQL1.OpenRead(ABatch, []);

```

Or using an atomic compound statement that doesn't return a cursor:

```

ABatch := 'begin atomic' +
  ' declare cust_id int;' +
  ' declare phone_no char(12);' +
  ' set cust_id = 101;' +
  ' set phone_no = ''3175558414'';' +
  ' update customer set phone=phone_no where id=cust_id;' +
  'end';
AsaSQL1.Execute(ABatch, []);

```

#### 4.2.15 Breaking a running stored procedure

The TAsaSession.Stop method is convenient to stop or break a running server request. This method works the same way as the "Stop-button" in the Command window of the Interactive SQL utility (ISQL), as part of the SQL Anywhere product.

This method can be called asynchronously by a second thread. The method is typically used to allow the user break a long running stored procedure, currently executed by the SA server engine. To use the Stop method you could define two threads. One thread could be used to execute a stored procedure and a second to break it.

The following example illustrates this scenario by allowing the main thread to create and stop a stored procedure, while a worker thread runs it:

```

procedure TForm1.ExecuteClick(Sender: TObject);
var
  AThread: TTestThread;
begin
  if AsaSQL1.Execute(
    'create procedure run_for_awhile() ' +
    'begin'+
    ' declare I integer;' +
    ' set I = 0;' +
    ' while I < 50000 loop' +           // Dummy loop
    '   set I = I + 1;' +
    ' end loop;' +
    'end', []) then
    begin
      // Let second thread execute it
      AThread := TTestThread.Create;

      // Let second thread run uninterrupted for 2 seconds
      Sleep(2000);

      // Main thread tries to stop the running stored procedure
    end;
  end;
end;

```

```

    if not AsaSession1.Stop then
        ShowMessage(AsaSession1.LastError);

    end else
        ShowMessage(AsaSQL1.LastError);
    end;
end;

```

And the code for the TTestThread class that executes the procedure:

```

constructor TTestThread.Create;
begin
    FreeOnTerminate := True;
    inherited Create(False);
end;

procedure TTestThread.Execute;
begin
    // Execute procedure created by the main thread
    Form1.AsaSQL1.Execute('call run_for_awhile', []);
end;

```

The code to break the running stored procedure would normally be executed by a “Stop-button” OnClick event, as in Interactive SQL. Then we can, as specified in the previous example, omit the “Sleep” Win32 API function, and have the following declaration instead:

```

procedure TForm1.StopClick(Sender: TObject);
begin
    // Main thread tries to stop the running stored procedure
    if not AsaSession1.Stop then
        ShowMessage(AsaSession1.LastError);
    end;
end;

```

... or if TAsaSession.WantExeptions are turned on:

```

procedure TForm1.StopClick(Sender: TObject);
begin
    // Main thread tries to stop the running stored procedure
    AsaSession1.Stop;
end;

```

A second technique, to stop a running server process, is to use the event TAsaSession.OnServerWait instead of a second thread. This event is called repeatedly while the database server or client library is busy processing a database request.

```

procedure TForm1.AsaSession1ServerWait(Sender: TObject; var AbortRequest: Boolean);
begin
    if StopButtonIsPressed then
        AsaSession1.Stop;
    end;
end;

```

Or you could also decide to use the AbortRequest parameter to achieve the same result:

```

procedure TForm1.AsaSession1ServerWait(Sender: TObject; var AbortRequest: Boolean);
begin
    if StopButtonIsPressed then
        AbortRequest := True;
    end;
end;

```

### 4.3 Working with TAsaDataset

The TAsaDataset component is typically used to replace Borland VCL's TQuery database component. In most cases you should expect compatibility between the two. The typical usage of TAsaDataset includes:

1. Drop a TAsaSession on the form.
2. Drop a TAsaDataset on the form
3. Drop a TDataSource on the form.
4. Drop data-aware components on the form.



5. Connect the TAsaDataset to the TAsaSession.
6. Connect the TDataSource to the TAsaDataset.
7. Connect the data-aware components to the TDataSource.

### 4.3.1 Queries

To execute a query, you need to assign it to the SQL property.

```
AsaDataset1.Close;
AsaDataset1.SQL.Clear;
AsaDataset1.SQL.Add('select * from customer order by id');
AsaDataset1.Open;
```

### 4.3.2 Parameters

A query/dataset can contain parameters prefixed by a colon. The following example assumes that the ParamCheck property is True.

```
AsaDataset1.Close;
AsaDataset1.SQL.Clear;
AsaDataset1.SQL.Add('select * from customer where id=:id');
AsaDataset1.ParamByName('id').AsInteger := 101;
AsaDataset1.Open;
```

### 4.3.3 Searching

To search for a row containing specified values, you use either Locate or Lookup.

```
if AsaDataset1.Locate('id', 110, []) then
  ShowMessage('Row found!');
```

To search for a row containing more than one value:

```
if AsaDataset1.Locate('id;lname', VarArrayOf(110, 'Agliori'), []) then
  ShowMessage('Row found!');
```

To search for a row without moving the current row pointer, you use Lookup.

```
AResult := AsaDataset1.Lookup('id', 110, 'lname');
if not VarIsNull(AResult) then
  ShowMessage('Customer last name is: ' + AResult);
```

TAsaDataset.FindKey, FindNearest and LocateNext can also be used for searching. Refer to the description of these methods for further details.

### 4.3.4 Appending rows

To append or insert a row to a table, you can use ExecSQL or one of the Insert or Append methods.

To insert a row using ExecSQL:

```
AsaDataset1.Close;
AsaDataset1.SQL.Clear;
AsaDataset1.SQL.Add('insert into customer (fname, lname)');
AsaDataset1.SQL.Add('values (''John'', ''Smith'')');
AsaDataset1.ExecSQL;
```

To insert a row using Insert:

```
AsaDataset1.Close;
AsaDataset1.SQL.Clear;
AsaDataset1.SQL.Add('select * from customer order by id');
AsaDataset1.Open;
AsaDataset1.Insert;
```

```
AsaDataset1.FieldByName('fname').AsString := 'John';
AsaDataset1.FieldByName('lname').AsString := 'Smith';
AsaDataset1.Post;
```

Do note that newly inserted row has no position in the result set, and will cause the row to disappear from view when using data-aware controls. To avoid this affect, set the TAsaDataset property SyncInsersts to True. For further details refer to the TAsaDataset. SyncInsersts property.

### 4.3.5 Modifying rows

To modify an existing row of a table, you can use ExecSQL or the Edit method.

To modify a row using ExecSQL:

```
AsaDataset1.Close;
AsaDataset1.SQL.Clear;
AsaDataset1.SQL.Add('update customer set fname='Johnny'');
AsaDataset1.SQL.Add('where lname='Smith'');
AsaDataset1.ExecSQL;
```

To modify a row using Edit:

```
AsaDataset1.Close;
AsaDataset1.SQL.Clear;
AsaDataset1.SQL.Add('select * from customer where lname='Smith'');
AsaDataset1.Open;
AsaDataset1.Edit;
AsaDataset1.FieldByName('fname').AsString := 'Johnny';
AsaDataset1.Post;
```

### 4.3.6 Deleting rows

To delete a row of a table, you can use ExecSQL or the Delete method.

To delete a row using ExecSQL:

```
AsaDataset1.Close;
AsaDataset1.SQL.Clear;
AsaDataset1.SQL.Add('delete from customer where lname='Smith'');
AsaDataset1.ExecSQL;
```

To delete a row using Delete:

```
AsaDataset1.Close;
AsaDataset1.SQL.Clear;
AsaDataset1.SQL.Add('select * from customer where lname='Smith'');
AsaDataset1.Open;
AsaDataset1.Delete;
```

### 4.3.7 Updating BLOBs

To update a BLOB using ExecSQL:

```
AsaDataset1.Close;
AsaDataset1.SQL.Clear;
AsaDataset1.SQL.Add('update pictures set pic=:pic where id=:id');
AsaDataset1.ParamByName('id').AsInteger := 101;
AsaDataset1.ParamByName('pic').Assign(Image1.Picture);
AsaDataset1.ExecSQL;
```

... or

```
AsaDataset1.Close;
AsaDataset1.SQL.Clear;
AsaDataset1.SQL.Add('update pictures set pic=:pic where id=:id');
AsaDataset1.ParamByName('id').AsInteger := 101;
```

```
AsaDataset1.ParamByName('pic').LoadFromFile('c:\picture.bmp, ntBlob);
AsaDataset1.ExecSQL;
```

... Or

```
AsaDataset1.Close;
AsaDataset1.SQL.Clear;
AsaDataset1.SQL.Add('update memos set a_memo=:a_memo where id=:id');
AsaDataset1.ParamByName('id').AsInteger := 101;
AsaDataset1.ParamByName('a_memo').Assign(Memo1.Lines);
AsaDataset1.ExecSQL;
```

To modify a row using Edit:

```
AsaDataset1.Close;
AsaDataset1.SQL.Clear;
AsaDataset1.SQL.Add('select * from pictures where id=:id');
AsaDataset1.ParamByName('id').AsInteger := 101;
AsaDataset1.Open;
AsaDataset1.Edit;
AsaDataset1.FieldByName('pic').Assign(Image1.Picture);
AsaDataset1.Post;
```

### 4.3.8 Concurrency issues

In most cases its convenient and wise to let the database server handle the situations when two or more users concurrently update a table. Handling row-locks and simultaneous data access is an issue normally left to the database server to safely deal with. In SA, this is the default behavior controlled by the database option BLOCKING=ON. This setting will cause the client thread to block until conflicting updates, protected by locks, are released by the other transaction holding these locks. However in some cases it could be necessary and handy to control row-locks and simultaneous data access yourself. Handling concurrency issues, from the client-side, is what this chapter is about.

In a typical TAsaDataset Edit-Post scenario, when two users attempt to edit the same record, one of them will get a "Couldn't perform the edit because another user changed the record"-exception. This is the standard concurrency behavior in both Borland's BDE (TQuery) and NativeDB's TAsaDataset.

But in some cases you would rather want to prevent the second user from even starting to edit the same row. So instead of the "Couldn't perform the edit..." error, you want the current user to be fully locked-out from editing the row and instead being notified with a "This row is currently being worked on by another user" kind of message. The "Couldn't perform the edit..." error will be raised on the attempt to place the dataset in edit-mode (TAsaDataset.Edit), BUT ONLY AFTER the second user posts his update (TAsaDataset.Post). But this is not soon enough if you want to totally prevent concurrent row-access in your multi-user application.

To solve this requirement, you would need to place an explicit lock on the current row on the attempt to place the dataset in edit-mode (TAsaDataset.Edit).

Among our first considerations are to minimize implicit database locks and allow all users to read any rows. To meet this requirement we decide to use an optimistic concurrency-schema at isolation level 0 (e.g. TAsaSession.TransIsolation := atReadUncommitted). Also to get any row-lock errors immediately returned back to the client, we need to set the database option BLOCKING to OFF.

```
AsaSession1.SetOption('blocking', 'off');
```

The next step is to code a TAsaDataset.BeforeEdit event to properly handle the concurrency issue. The event handler should both lock the current row and check if another user already has placed a lock on that row.

```

procedure TForm1.AsaDataset1BeforeEdit(DataSet: TDataSet);
var
    CurrCustID: Extended;
    TempCursor: TAsaDataset;
begin
    if not AsaSession1.InTransaction then
        AsaSession1.StartTransaction;

    CurrCustID := DataSet.FieldByName('custno').AsFloat;
    TempCursor := TAsaDataset.Create(nil);
    try
        TempCursor.Session := TAsaDataset(DataSet).Session;
        TempCursor.SQL.Clear;

        // Post a "dummy" change to force the current row to be locked.
        TempCursor.SQL.Add('update customer set custno=custno where custno=:custno');
        TempCursor.ParamByName('custno').AsFloat := CurrCustID;
        try
            TempCursor.ExecSQL;
        except
            on E: ENativeDatasetError do
                begin
                    if E.ErrorCode = -210 then
                        // Show a more "human-readable" error-message !
                        DatabaseError('This row is currently being worked on by another user.' +
                            'The full error message reads:' + #10 +
                            TempCursor.Session.LastError + #10 +
                            IntToStr(TempCursor.Session.LastErrorCode))
                    else
                        if E.ErrorCode = 100 then
                            DatabaseError('This row has been deleted.' +
                                'The full error message reads:' + #10 +
                                TempCursor.Session.LastError + #10 +
                                IntToStr(TempCursor.Session.LastErrorCode))
                        else
                            raise;
                    end;
                end;
            finally
                TempCursor.Free;
            end;
        end;
    end;

```

Note: Instead of posting the above “dummy” update to force the row-lock, we could have used the TAsaSQL.Lock method and achieved the same result.

We are now preventing a second user from editing the current row if it’s in use by the other transaction. So at this point we could also decide to ignore the “Couldn’t perform the edit...”-exception by the following code.

```

procedure TForm1.AsaDataset1EditError(DataSet: TDataSet; E: EDatabaseError;
    var Action: TDataAction);
begin
    Action := daRetry;
end;

```

As a concluding note, the recommended usage of the above event handlers is to design the edit-post code in such a way that it avoids any user-interaction between the physical locking of the current record and the final commit (or rollback). Ignoring this could potentially leave the locks on “forever” if the user (holding the locks) leaves the office. Keep this in mind if you plan to use VCL data-aware controls (i.e. DBGrid’s, DBEdit’s etc.) with this solution. If you still want/need to use data-aware user-input controls, at least consider the SA “-ti” database switch as a safeguard. This switch will cause inactive connections to be disconnected from the database when a certain amount of time has elapsed and will in turn cause any active locks to be released.

The above example more or less assumes that the client-side lock protection schema is executed through VCL data-aware controls. But in some situations it could also be convenient to use the same row-lock technique to protect a critical block of code in an “outermost” level before any attempt is made to execute the code in the “innermost” parts. The following

example shows a very tight and protected code-block that uses a very simple table to protect other transactions from reaching the "inner-most" code, if it's in use by a current transaction.

```

create table seat_table
(
    seat    integer not null,    ; Seat number
    status  integer null,       ; 0=Vacant, 1=Occupied
    primary key (seat)
)

procedure TForm1.ReserveSeat(ASeat: Integer);
// Ticket reservation.
var
    szBlocking: string;
begin
    if AsaSession1.GetOption('blocking', szBlocking) and (szBlocking = 'ON') then
        AsaSession1.SetOption('blocking', 'OFF');

    if not AsaSession1.InTransaction then
        AsaSession1.StartTransaction;
    try
        // The following line(s) (OpenWrite) will not "freeze" other clients
        // (e.g. BLOCKING=OFF) when accessing the same seat, but will
        // immediately show a "Seat # is taken/occupied"-message instead.
        // The call to TAsaSQL.Lock will put a write-lock on the current row,
        // causing a second user's request for the same seat, to fail.
        // ESQL: "EXEC SQL OPEN CURSOR FOR UPDATE"
        // ESQL: "EXEC SQL FETCH RELATIVE 0 FOR UPDATE".
        if AsaSQL1.OpenWrite('select * from seat_table where seat=? and status=0',
[ASeat]) and
            AsaSQL1.Lock then
            begin

                // To make the client-side lock handling worth all the efforts
                // (e.g. BLOCKING=OFF), the code to add here would typically
                // take some time to execute. Thus, the above write-lock will
                // avoid to do the credit-card validation and the ticket printout
                // for a second user, trying to reserve the same seat. Say f.ex.
                // that both operations take 30-60 secs. to execute. i.e. this
                // code would prevent a "conflicting" user's workstation from being
                // blocked/"frozen" in 30-60 secs. Also remember that any blocked
                // clients (e.g. in BLOCKING=ON mode) shouldn't attempt to do these
                // operations on the occupied seat anyway.

                ValidateCreditCard;
                PrintTicket;

                // For testing purposes, comment in the following line if you
                // want to simulate a code-flow that will take some time to execute.
                // This will allow you to experience what happens if a second user
                // try to reserve the same seat.

                // ShowMessage(Format('Seat %d is write-locked', [ASeat]));

                // And finally update the seat_table
                AsaSQL1['status'] := 1;

                // ESQL: "EXEC SQL UPDATE"
                AsaSQL1.Modify;

                // ESQL: "EXEC SQL CLOSE"
                AsaSQL1.Close;

                // ESQL: "EXEC SQL COMMIT"
                AsaSession1.Commit;

            end else
            begin
                if AsaSQL1.LastErrorCode = 100 then
                    raise Exception.Create(Format('Seat %d is occupied or doesn't exist !',
[ASeat]))
                else
                    if AsaSQL1.LastErrorCode = -210 then
                        raise Exception.Create(Format('Seat %d is (about to be) taken !', [ASeat]))
                    else
                        raise Exception.Create(AsaSQL1.LastError)
            end
        end
    end
end

```

```

        end;
    except
        // ESQ: "EXEC SQL ROLLBACK"
        AsaSession1.Rollback;
        raise;
    end;
end;

```

## 4.4 Working with TAsaStoredProc

The TAsaStoredProc component is typically used to replace Borland VCL's TStoredProc database component. In most cases you should expect compatibility between the two.

Although you can use TAsaDataset with most stored procedures, TAsaStoredProc is handy if your SP includes OUT or INOUT parameters.

### 4.4.1 Stored Procedures with INOUT and OUT parameter types

Consider the following SP:

```

create procedure greater(in a integer, in b integer, out c integer)
begin
    if a>b then
        set c=a
    else
        set c=b
    end if
end

```

You can call it with the following code:

```

AsaStoredProc1.StoredProcName := 'greater';
AsaStoredProc1.Prepare;
AsaStoredProc1.ParamByName('a').AsInteger := StrToInt(Edit1.Text);
AsaStoredProc1.ParamByName('b').AsInteger := StrToInt(Edit2.Text);
AsaStoredProc1.ExecProc;
ShowMessage(AsaStoredProc1.ParamByName('c').AsString);

```

### 4.4.2 Stored Procedures returning Result sets

Your SA database ships with many useful system procedures. The sa\_conn\_info procedure returns a result-set (cursor) with information about current active database connections. The following code uses TAsaStoredproc to call it:

```

AsaStoredProc1.StoredProcName := 'sa_conn_info';
AsaStoredProc1.Prepare;
AsaStoredProc1.ParamByName('connidparm').Clear;
AsaStoredProc1.Open;

```

### 4.4.3 Resuming a Stored Procedure

Stored procedures that return one or more result-sets (cursors) can be instructed to resume execution after the client has finished processing the result set.

Consider the following SP:

```

create procedure multi_cust()
begin
    select id,fname from customer order by id asc;
    select id,lname,address from customer order by id asc
end

```

When first called, the SP makes the first cursor available to the client. When the client has finished processing the result-set, it should allow the SP to continue execution. The SP must be explicitly told to continue by the call to TAsaStoredProc.Resume.

```
AsaStoredProc1.StoredProcName := 'multi_cust';
AsaStoredProc1.Open;
repeat
    while not AsaStoredProc1.EOF do
        // Process the rows here
        AsaStoredProc1.Resume;
    until not AsaStoredProc1.StillInProc;
```

## 5. Appendices

### 5.1 Converting from BDE to NDB

This chapter discusses some of the concerns when converting an existing BDE based application to NativeDB.

A typical Borland BDE based application consists of one or more datamodules and forms. On these forms one have a TDatabase and one or more TQuery's and TTable's. We assume that this is a single-threaded application, and that the default TSession is used. To feed data-aware components with data, a TDataSource is also used as the link between the controls and the dataset.

#### 5.1.1 Step 1 - Replacing TDatabase or a BDE Alias

- Replace the TDatabase instance with a TAsaSession instance. If you use a fixed BDE alias instead of a TDatabase alias, to represent the connection, you still need a TAsaSession instance.
- Name the TAsaSession (Name property) the same as the TDatabase.DatabaseName property. This will later ease the understanding of the link between your TAsaDataset's and the TAsaSession.
- If you use TDatabase's default login prompt to provide login information, change to the TAsaLogin component instead for the same purpose.
- Your current BDE alias is linked to a ODBC alias that contain the different SQL Anywhere database parameters. Look up the ODBC alias. Assign the "Server name" parameter, if any, to TAsaSession.LoginEngineName. Assign the "Start line" parameter, if any, to TAsaSession.ServerParams. Insert a "start=" in front of the string. Assign either "Database name" or "Database file" to TAsaSession.LoginDatabase. This depends on whether you are setting up a client connection or a personal engine (server) connection. Set TAsaSession.ServerType accordingly. Assign the "Network" options to TAsaSession.ClientParams.
- Verify TAsaSession.LibraryFile with your current SA version.
- Verify that TAsaSession.LibraryPath is pointing to the location of were LibraryFile resides. (N/A on Linux), or leave it empty to rely on the system PATH.
- Consider TAsaSession.OEMConvert if your database contains OEM based characters.
- Lookup up your BDE alias setting SHARED PASSTHROUGHMODE. If the current setting is SHARED AUTOCOMMIT the set TAsaSession.AutoCommit = True. Otherwise False.
- Compare TDatabase.TransIsolation to TAsaSession.TransIsolation.
- If your project relies heavily on exception handlers, set TAsaSession.WantExceptions to True.
- Set TAsaSession.Connected equal to TDatabase.Connected.

### 5.1.2 Step 2 - Replacing TQuery

- Replace the TQuery components with TAsaDataset components.
- Name them equally.
- Compare TQuery.DatabaseName to TAsaDataset.Session. Given equal database names at step 1, they should be equal.
- Assign TQuery.SQL to TAsaDataset.SQL.
- Change any TQuery.Filter into a WHERE clause with TAsaDataset.SQL.
- Compare TQuery.ParamCheck and TQuery.Params to TAsaDataset.ParamCheck and TAsaDataset.Params.
- Assign TQuery.Datasource to TAsaDataset.Datasource, to feed parameter values based on another query.
- Assign TQuery.AutoCalcFields to TAsaDataset.AutoCalcFields.
- TQuery.RequestLive=False is the same as TAsaDataset.ReadOnly=True. Note: If RequestLive is True and you are inserting rows into this query, also set TAsaDataset.SyncInserts to True.
- Verify all events.
- Double-click the TAsaDataset component and verify that all TField descendants are the same as defined with TQuery.
- For persistent lookup-fields, it's recommended to enable the LookupCache property of these fields.
- Set TAsaQuery.Active equal to TAsaDataset.Active.
- Verify that all TDataSource components are pointing to the new TAsaDataset components.

### 5.1.3 Step 3 - Replacing TTable

- TTable's are not designed for use with RDBMS based databases. It's highly recommended to replace them with true query based classes. In other words, replace the TTable components with TAsaDataset components. Do note that the TTable's searching capabilities FindKey and FindNearest are available with TAsaDataset.
- Name them equally.
- Compare TTable.DatabaseName to TAsaDataset.Session. Given equal database names at step 1, they should be equal.
- Assign to TAsaDataset.SQL a query like "select \* from <TTable.TableName> order by <TTable.IndexFieldNames>".
- Change any TTable.Filter into a WHERE clause with TAsaDataset.SQL.
- Set TAsaDataset.Datasource equal to TTable.MasterSource.
- Consider TTable.MasterFields, and replace the master fields by updating TAsaDataset.SQL's WHERE clause to include parameters as placeholders for the master fields.
- Assign TTable.AutoCalcFields to TAsaDataset.AutoCalcFields.
- Assign TTable.ReadOnly to TAsaDataset.ReadOnly. Note: If ReadOnly is False and you are inserting rows into this table, also set TAsaDataset.SyncInserts to True.
- Verify all events.
- Double-click the TAsaDataset component and verify that all TField descendants are the same as defined with TTable.
- For persistent lookup-fields, it's recommended to enable the LookupCache property of these fields.
- Set TAsaQuery.Active equal to TAsaDataset.Active.
- Verify that all TDataSource components are pointing to the new TAsaDataset components.

### 5.1.4 Step 4 - Replacing TStoredProc

- Replace the TStoredProc components with TAsaStoredProc components.
- Name them equally.



- Compare TStoredProc.DatabaseName to TAsaStoredProc.Session. Given equal database names at step 1, they should be equal.
- Assign TStoredProc.StoredProcName to TAsaStoredProc.StoredProcName.
- Compare parameters assigned to TStoredProc.Params with TAsaStoredProc.Params.
- Assign TStoredProc.AutoCalcFields to TAsaStoredProc.AutoCalcFields.
- Verify all events.
- Double-click the TAsaStoredProc component and verify that all TField descendants are the same as defined with TStoredProc.
- Set TStoredProc.Active equal to TAsaStoredProc.Active.

### 5.1.5 Step 5 - Changing Sourcecode

- Remove any USES references to the BDE and DBTables units.
- Replace any direct BDE function calls (dbiXXXX).
- Remove any reference to the Session VCL global variable.
- Replace any TDatabase, TQuery, TStoredProc class references or type casts, to either the generic TDataset class, to TAsaSession, TAsaDataset or TAsaStoredProc.
- Coded usage of TQuery.Filter is not supported by TAsaDataset, and should be changed into parameter values assigned through TAsaDataset.ParamByName instead. This will feed a WHERE clause with the given parameter values.
- Change any TQuery cached updates into regular SA transactions.
- The TDatabase EDBEngineError exceptions must be updated to reference the ENativeException and EAsaException classes instead.
- The TQuery and TStoredProc EDBEngineError must be updated to reference the ENativeDatasetError instead.
- TQuery.RequestLive=False is the same as TAsaDataset.ReadOnly=True.
- Consider BLOBs back into the project, if they were removed caused by some known BDE BLOB issues (e.g. BLOBs larger than 1MB). Large BLOBs are fully supported by TAsaDataset.

Another approach to consider, if you want to ease the conversion process, is to implement your own derived data access classes. This will allow you to handle most of the above requirements inside your own class. The "MyNdb.pas" unit found in the \Bonus directory will be a convenient starting-point to implement your own descendant classes. These classes will also allow you to switch back and forth between BDE and NDB throughout your conversion process.

## 5.2 Using NativeDB with C++Builder

The following sections show some code samples by using Borland C++Builder instead of Delphi. The samples focus on the generic class TAsaSQL and assume you have the basic knowledge of the VCL class TDataset (i.e. TAsaDataset) already.

### 5.2.1 Connecting to a running database engine

The following code sample connects the client to a running ASA server engine on the LAN.

```
TAsaSession *AsaSession1;
.
.
.
try
{
    AsaSession1->ClientParams = "links=tcipip";
    AsaSession1->ServerType = stClient;
    AsaSession1->LoginEngineName = "MyEngine";
    AsaSession1->LoginDatabase = "MyDB";
    AsaSession1->LoginUser = "dba";
    AsaSession1->LoginPassword = "sql";
    AsaSession1->Connected = True;
}
```

```

}
catch(const EAsaException &E)
{
    ShowMessage(E.Message);
}

```

### 5.2.2 A simple query

The sample below uses TAsaSQL to open a cursor using a simple query without any host variables.

```

if (AsaSQL1->OpenRead("select * from customer where city='New York'", &NdbEmptyParam,
0))
{
    do
        ListBox1->Items->Add(AsaSQL1->FieldValues["fname"] + String(" ") + AsaSQL1->FieldValues["lname"]);
    while (AsaSQL1->Next());
}
else
    ShowMessage(AsaSQL1->LastError);

```

### 5.2.3 A query with host variables

A query can also include host variables or parameters bound to the statement.

```

if (AsaSQL1->OpenRead("select * from customer where city=:city", OPENARRAY(Variant,
("New York"))))
{
    do
        ListBox1->Items->Add(AsaSQL1->FieldValues["fname"] + String(" ") + AsaSQL1->FieldValues["lname"]);
    while (AsaSQL1->Next());
}
else
    ShowMessage(AsaSQL1->LastError);

```

More host variables can be passed in the second parameter.

```

if (AsaSQL1->OpenRead("select * from customer where id=:id and city=:city",
OPENARRAY(Variant, (103, "New York"))))

```

You can also use the short abbreviation form (?) when specifying the host variables.

```

if (AsaSQL1->OpenRead("select * from customer where id=? and city=?",
OPENARRAY(Variant, (103, "New York"))))

```

### 5.2.4 Appending rows

You can use an open cursor to append new rows.

```

AsaSQL1->OpenWrite("select * from customer", &NdbEmptyParam, 0);
if (AsaSQL1->Opened)
{
    AsaSQL1->Clear(); // Set all columns to the NULL state
    AsaSQL1->FieldValues["fname"] = "Frank";
    AsaSQL1->FieldValues["lname"] = "Johnsen";
    if (!AsaSQL1->Append())
        ShowMessage(AsaSQL1->LastError);
}
else
    ShowMessage(AsaSQL1->LastError);

```

You can also add a new row to an empty cursor (no rows in the result set). In this case it's no need to call Clear to empty the row.

```

AsaSQL1->OpenWrite("select * from customer where A<>A", &NdbEmptyParam, 0);
if (AsaSQL1->Opened)

```

```

{
  AsaSQL1->FieldValues["fname"] = "Frank";
  AsaSQL1->FieldValues["lname"] = "Johnsen";
  if (!AsaSQL1->Append())
    ShowMessage(AsaSQL1->LastError);
}
else
  ShowMessage(AsaSQL1->LastError);

```

### 5.2.5 Modifying rows

An open cursor can also be used when modifying existing rows.

```

if (AsaSQL1->OpenWrite("select * from customer where lname='Johnsen'", &NdbEmptyParam,
0))
{
  AsaSQL1->FieldValues["lname"] = "Johnston";
  if (!AsaSQL1->Modify())
    ShowMessage(AsaSQL1->LastError);
}
else
  ShowMessage(AsaSQL1->LastError);

```

### 5.2.6 Deleting rows

As required when modifying and appending rows, you must obtain a write-able result set to delete a record in a table.

```

if (AsaSQL1->OpenWrite("select * from customer where lname='Johnston'", &Null, 0))
{
  if (!AsaSQL1->Delete())
    ShowMessage(AsaSQL1->LastError);
}
else
  ShowMessage(AsaSQL1->LastError);

```

### 5.2.7 Executing SQL statements

You can use TAsaSQL.Execute to execute any general purpose (DML) SQL statements. So, to delete a row using Execute you could use the following code.

```

if (!AsaSQL1->Execute("delete from customer where fname='John'", &NdbEmptyParam, 0))
  ShowMessage(AsaSQL1->LastError);

```

### 5.2.8 Calling Stored Procedures

The following sample demonstrates an SP that returns a value (Stored Function).

```

create function fullname(in firstname char(30), in lastname char(30)) returns char(61)
begin
  declare name char(61);
  set name = firstname || ' ' || lastname;
  return(name);
end

```

And the code to call it.

```

if (AsaSQL1->Execute("=?=call fullname(?,?)", OPENARRAY(Variant, ("Frank",
"Johnsen"))))
  ShowMessage("Name: " + AsaSQL1->GetOutVar("fullname"));

```

Stored procedures can supply both IN and OUT parameters (and INOUT).

```

create procedure greater(in a integer, in b integer, out c integer)
begin
  if a>b then
    set c=a

```

```

else
    set c=b
end if
end

```

And the code to call it.

```

if (AsaSQL1->Execute("call greater(?,?,?)", OPENARRAY(Variant, (10,20))))
    ShowMessage(FloatToStr((float)AsaSQL1->GetOutVar("c")));

```

## 5.3 Using NativeDB ActiveX with Visual Basic

After you have registered the NativeDB ActiveX automation server according to the installation instructions in chapter 2, you are ready to begin development with NativeDB (NDBAX) and Microsoft Visual Basic. The following sections demonstrate the most commonly used features in the NDBAX library. The library is named "NDBASA" and includes two main dual-class interfaces to access Sybase SQL Anywhere. These are "XAsaSession" to control the connection and transaction handling against the database, and "XAsaSQL" to execute any valid SQL statement. Refer to section 3.1 and 3.2 for the complete reference, including property and methods instruction details.

### 5.3.1 Running the VB demo

Start Visual Basic and open the VBDemo project found in the directory "Demos\VBDemo", relative to the NativeDB home directory. Compile and execute the application. Work with this example for a while to get familiarized with the most basic NDBAX features.

### 5.3.2 Getting started

Create a new blank VB project or open an existing. Select Project | References and include the NativeDB ActiveX automation server with your application. The two main classes are declared in VB with the following code:

```

Dim AsaSession1 As NDBASA.XAsaSession
Dim AsaSQL1 As NDBASA.XAsaSQL

```

Then create the object instances:

```

Set AsaSession1 = New NDBASA.XAsaSession
Set AsaSQL1 = New NDBASA.XAsaSQL

```

### 5.3.3 Client connection

The following code sample connects to an already running ASA server engine on the LAN.

```

AsaSession1.ClientParams = "links=tcPIP"
AsaSession1.ServerType = stClient
AsaSession1.LoginEngineName = "MyEngine"
AsaSession1.LoginDatabase = "MyDB"
AsaSession1.LoginUser = "dba"
AsaSession1.LoginPassword = "sql"
AsaSession1.Connected = True 'Optionally use AsaSession1.Open

```

### 5.3.4 Starting and connecting to the ASA personal or runtime engines

The below code demonstrates how to launch and connect to an ASA engine implicitly from your application.

```

AsaSession1.ServerParams = "start=dbeng7.exe"
AsaSession1.ServerType = stServer
AsaSession1.LoginDatabase = "c:\MyApp\MyDB.db"
AsaSession1.LoginUser = "dba"
AsaSession1.LoginPassword = "sql"

```

```
AsaSession1.Connected = True
```

### 5.3.5 A simple query

Before you can execute any SQL statements you must first link the statement object (XAsaSQL) to the connection object (XAsaSession).

```
AsaSQL1.Session = AsaSession1
```

The example below uses XAsaSQL to open a simple query without any host variables.

```
If AsaSQL1.OpenRead("select * from customer where city='New York'") Then
    Do While Not AsaSQL1.EOF
        List1.AddItem(AsaSQL1("fname") & " " & AsaSQL1("lname "))
        AsaSQL1.Next
    Loop
Else
    MsgBox(AsaSQL1.LastError)
End If
```

Slightly more compact code can be written when checking the return value of XAsaSQL.Next instead of watching the EOF state.

```
If AsaSQL1.OpenRead("select * from customer where city='New York'") Then
    Do
        List1.AddItem (AsaSQL1("fname") & " " & AsaSQL1("lname "))
    Loop Until Not AsaSQL1.Next
Else
    MsgBox(AsaSQL1.LastError)
End If
```

It's also optional to use a subroutine-based coding style, which relies on proper error handling through exceptions, instead of the function-base style (True/False) we did in the first two examples.

```
AsaSQL1.WantExceptions = True
AsaSQL1.OpenRead "select * from customer where city='New York'"
Do While Not AsaSQL1.EOF
    List1.AddItem(AsaSQL1("fname") & " " & AsaSQL1("lname "))
    AsaSQL1.Next
Loop
```

### 5.3.6 A query with host variables

A query can also contain host variables or parameters bound to the statement.

```
If AsaSQL1.OpenRead("select * from customer where city=:city", "New York") Then
    Do While Not AsaSQL1.EOF
        List1.AddItem(AsaSQL1("fname") & " " & AsaSQL1("lname "))
        AsaSQL1.Next
    Loop
Else
    MsgBox(AsaSQL1.LastError)
End If
```

If the statement contains multiple host variables a VB Variant Array is required to pass the parameters.

```
If AsaSQL1.OpenRead("select * from customer where id=:id and city=:city", _
    Array(103, "New York")) Then
    List1.AddItem(AsaSQL1("fname") & " " & AsaSQL1("lname "))
End If
```

It's optional to use the short abbreviation form, using a question mark (?) as the parameter placeholder, when specifying the host variables.

```

If AsaSQL1.OpenRead("select * from customer where id=? and city=?", _
    Array(103, "New York")) Then
    .
    .
End If

```

### 5.3.7 Executing SQL statements

The XAsaSQL.Execute method can be used to execute any general purpose (DML) statement. The below examples illustrate a call to delete a customer record.

```

If Not AsaSQL1.Execute("delete from customer where fname='John'")
    MsgBox(AsaSQL1.LastError)
End If

```

Or ...

```

If Not AsaSQL1.Execute("delete from customer where fname=?", "John")
    MsgBox(AsaSQL1.LastError)
End If

```

### 5.3.8 Calling Stored Procedures

The following example demonstrates an SP that returns a value (Stored Function) and how to call it.

```

create function fullname(in firstname char(30), in lastname char(30)) returns char(61)
begin
    declare name char(61);
    set name = firstname || ' ' || lastname;
    return(name);
end

```

```

If AsaSQL1.Execute("?=call fullname(?,?)", Array("Frank","Johnsen")) Then
    MsgBox("Name: " & AsaSQL1.GetOutVar("fullname"))
End If

```

A Stored procedure can supply both IN and OUT parameters (and INOUT).

```

create procedure greater(in a integer, in b integer, out c integer)
begin
    if a>b then
        set c=a
    else
        set c=b
    end if
end

```

And the code to call it.

```

If AsaSQL1.Execute("call greater(?,?,?)", Array(10,20)) Then
    MsgBox("Result: " & AsaSQL1.GetOutVar("c"))
End If

```

### 5.3.9 Appending rows

In addition to the regular INSERT, UPDATE and DELETE SQL statements executed by XAsaSQL.Execute, you can insert new rows into an open cursor. This becomes handy when you want to retrieve database assigned column-defaults or auto-incremental values directly into your application.

```

AsaSQL1.OpenWrite "select * from customer"
If AsaSQL1.Opened Then
    AsaSQL1.Clear          'Set all columns to the NULL state
    AsaSQL1("fname") = "Frank"
    AsaSQL1("lname") = "Johnsen"
    If Not AsaSQL1.Append Then

```

```

        MsgBox(AsaSQL1.LastError)
    Else
        'Column defaults or auto-incremental values are available here by regular
        'field retrieval methods.
        MsgBox("The new auto-inc value is: " & AsaSQL1("id"))
    End If
Else
    MsgBox(AsaSQL1.LastError)
End If

```

### 5.3.10 Modifying rows

An open cursor can also be used to update an existing row.

```

If AsaSQL1.OpenWrite("select * from customer where lname='Johansen'") Then
    AsaSQL1("lname") = "Johnsen"
    If Not AsaSQL1.Modify Then
        MsgBox(AsaSQL1.LastError)
    End If
Else
    MsgBox(AsaSQL1.LastError)
End If

```

### 5.3.11 Deleting rows

As required when modifying or appending rows, you must first obtain a write-able result set to delete a record in the table.

```

If AsaSQL1.OpenWrite("select * from customer where lname='Johansen'") Then
    If Not AsaSQL1.Delete Then
        MsgBox(AsaSQL1.LastError)
    End If
Else
    MsgBox(AsaSQL1.LastError)
End If

```

### 5.3.12 Using Events

In many situations SQL Anywhere notifies the client of an event through a callback function. These callback functions are translated into ActiveX events and are immediately available to your application. Sections 3.1.3 and 3.2.3 lists all available events. The ASA MESSAGE TO CLIENT statement is a nice example of a useful event to handle in your end-user application. Note: The MESSAGE statement is available in ASA 6.01 or later releases.

To enable event support in your VB project, you must first include the keyword "WithEvents" in your object declaration.

```
Dim WithEvents AsaSession1 As NDBASA.XAsaSession
```

The following stored procedure notifies the client-application with an information message:

```

create procedure dba.notify_client()
begin
    message 'Hello NativeDB!' type info to client
end

```

When executed, ASA triggers a callback, and the incoming message could be handled in an event routine similar to the following:

```

Private Sub AsaSession1_OnServerMessage(ByVal Sender As IXAsaSession, ByVal
MessageType As ServerMessageType, ByVal MessageCode As Long, ByVal Msg As String)
    Select Case MessageType
        Case smtInfo
            List1.AddItem("Event OnServerMessage raised = smtInfo: " & Msg)
        Case smtWarning
            List1.AddItem("Event OnServerMessage raised = smtWarning: " & Msg)
        Case smtAction
    
```

```
List1.AddItem("Event OnServerMessage raised = smtAction: " & Msg)
Case smtStatus
    List1.AddItem("Event OnServerMessage raised = smtStatus: " & Msg)
End Select
End Sub
```