

## BDEDoRxS unit

A unit that provides you with a more friendly interface to some DBI\*-Functions, including table and index creation, restructuring, setting and retrieving RI and valchecks. There are also functions to work with table definition files, enabling you to very easily save the properties of your tables to file and create or restructure tables from those files at your client's site. See '[wrappers](#)' or the source of the [sample project](#) 'Restruc' for ready-to-paste methods you can include with your own custom restructure/table creation project.

### Miscellaneous

[Concept](#)

[Definition files](#)

[Sample projects](#)

[General hints and trap wires](#)

[Limitations on restructuring \(sorry...\)](#)

[Using custom dialogs with BDEDoRxS](#)

[Download areas / address information](#)

[Disclaimer](#)

### Registration

[Register the Restructure Toolkit](#)

### Wrappers

[Easy wrapper methods](#)

### Routines

[Routines by Group](#)

### Components

[Px7Table component](#)

[DBFTable component](#)

### Types

[TDoRxList](#)

[EDoRxError](#)

[EDoRxKeyViol](#)

[EDoRxProblem](#)

### Constants

[szBLOCKSIZE](#)

[szLANGDRV](#)

[szLEVEL](#)

[szMDXBLOCKSIZE](#)

[szMEMOBLOCKSIZE](#)

[szSTRICTRI](#)

## EDoRxProblem type

### Unit

BDEDoRxS

### Declaration

```
EDoRxProblem = class(EDoRxError)
```

### Description

An exception of this type will be raised if data has been transferred to a problem table during a restructure. Property TableName will hold the complete path and name of that table while the message only displays the name.

## EDoRxKeyViol type

### Unit

BDEDoRxS

### Declaration

```
EDoRxKeyViol = class(EDoRxError)
```

### Description

An exception of this type will be raised if data has been transferred to a keyviol table during a restructure. Property TableName will hold the complete path and name of that table while the message only displays the name.

## TDoRxList type

### Unit

BDEDoRxS

### Declaration

```
TDoRxList = class(TList)
```

### Description

This is a simple TList descendant that adds a method FreeAll to free it's items.

The FieldDescList used in all methods that do a field restruct is a List of TStringList. Each stringlist holds the info for one field in for following form just the same as it appears in the stringgrid of the BDEDoRx demo):

stringlist[0] = fieldname,

stringlist[1] = type, using the single char abbreviations from DBD, e.g. 'F' is float, 'D' is date and so on

stringlist[2] = size, for dbase float and num seperated by a comma

stringlist[3] = field no.

**!!!MOST IMPORTANT!!!:** in order to determine whether a field has been moved or modified, you will have to supply each field no in the \*old\* table when restructuring a table. Remember that you can change name, type, size and position in one single action - there has to be some sort of identifier. so for restructuring a table, first get the field data with BDEGetFieldStructure, modify it while keeping the field no, and feed the modified data back to BDEFieldRestruct.

## szLEVEL constant

### Unit

BDEDoRxS

### Declaration

```
szLEVEL = 'LEVEL';
```

### Description

For use with BDETableCreate, BDEOptRestruct and BDEFieldAndOptRestruct.

## szLANGDRV constant

### Unit

BDEDoRxS

### Declaration

```
szLANGDRV = 'LANGDRIVER';
```

### Description

For use with BDETableCreate, BDEOptRestruct and BDEFieldAndOptRestruct.

## szBLOCKSIZE constant

### Unit

BDEDoRxS

### Declaration

```
szBLOCKSIZE = 'BLOCK SIZE';
```

### Description

For use with BDETableCreate, BDEOptRestruct and BDEFieldAndOptRestruct.

## szSTRICTRI constant

### Unit

BDEDoRxS

### Declaration

```
szSTRICTRI = 'STRICTINTEGRITY';
```

### Description

For use with BDETableCreate, BDEOptRestruct and BDEFieldAndOptRestruct.



## szMDXBLOCKSIZE constant

### Unit

BDEDoRxS

### Declaration

```
szMDXBLOCKSIZE = 'MDX BLOCK SIZE';
```

### Description

For use with BDETableCreate, BDEOptRestruct and BDEFieldAndOptRestruct.

## szMEMOBLOCKSIZE constant

### Unit

BDEDoRxS

### Declaration

```
szMEMOBLOCKSIZE = 'MEMO FILE BLOCK SIZE';
```

### Description

For use with BDETableCreate, BDEOptRestruct and BDEFieldAndOptRestruct.

## BDEGetDBPath routine

### Unit

BDEDoRxS

### Declaration

```
function BDEGetDBPath(AliasName: string): TFileName;
```

### Description

This function returns the path of a global or local alias. Pass either the name of the alias (if it's a global alias) or the DataBaseName of a TDataBase (if it's a local alias).

## **BDEGetTablePath routine**

### **Unit**

BDEDoRxS

### **Declaration**

```
function BDEGetTablePath(ATable: TTable): TFileName;
```

### **Description**

This function returns the path of the TTable being passed in.

## **BDESaveTableDefsToFile routine**

### **Unit**

BDEDoRxS

### **Declaration**

```
procedure BDESaveTableDefsToFile(ATable: TTable;  
                                const AFileName: TFileName);
```

### **Description**

Writes a table definition file to disk. You need to pass an active TTable and a fully qualified file name.

## BDERestructTableFromFile routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDERestructTableFromFile(ATable: TTable;  
                                   const AFileName: TFileName);
```

### Description

This will do a field restructure as well as a optional parameter restructure of ATable, using the entries of the table definition file AFileName.

## BDECreateTableFromFile routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDECreateTableFromFile (ADataBase: TDataBase;  
                                   const AFileName: TFileName);
```

### Description

Used to create local tables from a definition file. Needs nothing but a TDataBase and a valid filename for a def file.

## BDEAddIndicesFromFile routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEAddIndicesFromFile(ATable: TTable;  
                                const AFileName: TFileName);
```

### Description

This adds/creates all indices included in the passed in definition file. You can use files created by either BDESaveTableDefsToFile or BDESaveIndexDefsToFile.



## BDERecoverIndicesFromFile routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDERecoverIndicesFromFile (ADB: TDataBase;  
                                     const ATableName, AFileName: TFileName);
```

### Description

This routine first calls BDEEmergencyDropAllIndices (what a name...) and will then re-create them from either a table or index definition file.

## BDESaveIndexDefsToFile routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDESaveIndexDefsToFile(ATable: TTable;  
                                const AFileName: TFileName);
```

### Description

Writes an index definition file for ATable to disk. You need to pass an active TTable and a fully qualified file name.

## BDEDropAllIndices routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEDropAllIndices(ATable: TTable);
```

### Description

This method drops all indices for the table passed in. If you encounter or suspect damaged indices, uses BDEEmergencyDropAllIndices.

## BDEEmergencyDropAllIndices routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEEmergencyDropAllIndices (ADB: TDataBase;  
                                       ATableName: string);
```

### Description

It does basically the same as [BDEDropAllIndices](#), but it works with damaged indices as well. Use this method to write a handler for 'Index out of date' errors. You might also want to have a look at [BDERecoverIndicesFromFile](#) which combines deleting and re-creating indices in 'Index out of date' situations.

See '[Wrappers](#)' as well.

## BDEGetFieldStructure routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEGetFieldStructure(ATable: TTable;  
                               AFieldDescList: TDoRxList);
```

### Description

Used to retrieve a list of field properties of ATable. For a description of the 'AFieldDescList' parameter refer to TDoRxList.

## BDEFieldRestruct routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEFieldRestruct(ATable: TTable;  
                           AFieldDescList: TDoRxList;  
                           AskForTrim: boolean;  
                           ShowProgrThreshold: longint);
```

### Description

This one does the restructuring. AskForTrim = True brings up a dialog when data might be truncated, ShowProgrThreshold determines the minimum recordcount that will bring up a progress form (-1 means never show progress).

For a description of the 'AFieldDescList' parameter refer to TDoRxList.

## BDEOptRestruct routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEOptRestruct(ATable: TTable;  
                        const OptNames,OptValues: array of string);
```

### Description

Used for restructuring a table with optional parameters. Field structure etc will be unaffected by this method. You can for example use it to set your tables to a higher table level or to increase Blocksize if your table is full.

You can use the following constants for OptNames:

```
const  
  {for use with both dBase and Paradox:}  
  szLEVEL           = 'LEVEL';  
  szLANGDRV         = 'LANGDRIVER';  
  {for use with Paradox only:}  
  szBLOCKSIZE       = 'BLOCK SIZE';  
  szSTRICTRI        = 'STRICTINTEGRITY'; {note the spelling!}  
  {for use with dBase only:}  
  szMDXBLOCKSIZE    = 'MDX BLOCK SIZE';  
  szMEMOBLOCKSIZE   = 'MEMO FILE BLOCK SIZE';
```

Valid OptValues are

- with szLEVEL: Paradox '3' to '7', dBase '3' to '5'
- with szLANGDRV: the *short* name of the language driver (i.e. 'INTL' instead of 'Paradox Intl')
- with szSTRICTRI: 'TRUE' or 'FALSE'
- with any sz\*BLOCKSIZE: block size in bytes (i.e. '4096' for a 4k setting)

## BDEFieldAndOptRestruct routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEFieldAndOptRestruct(ATable: TTable;  
                                AFieldDescList: TDoRxList;  
                                AskForTrim: boolean;  
                                ShowProgrThreshold: longint,  
                                const OptNames,  
                                OptValues: array of string);
```

### Description

Used for restructuring a table's field structure plus optional parameters. You can for example use it to restructure your table and set your tables to a higher table level or to increase Blocksize if your table is full at the same time. This method is used by BDERestructTableFromFile.

AskForTrim = True brings up a dialog when data might be truncated, ShowProgrThreshold determines the minimum recordcount that will bring up a progress form (-1 means never show progress).

You can use the following constants for OptNames:

```
const  
    {for use with both dBase and Paradox:}  
    szLEVEL          = 'LEVEL';  
    szLANGDRV        = 'LANGDRIVER';  
    {for use with Paradox only:}  
    szBLOCKSIZE      = 'BLOCK SIZE';  
    szSTRICTRI       = 'STRICTINTEGRITY'; {note the spelling!}  
    {for use with dBase only:}  
    szMDXBLOCKSIZE   = 'MDX BLOCK SIZE';  
    szMEMOBLOCKSIZE  = 'MEMO FILE BLOCK SIZE';
```

Valid OptValues are

- with szLEVEL: Paradox '3' to '7', dBase '3' to '5'
- with szLANGDRV: the *short* name of the language driver (i.e. 'INTL' instead of 'Paradox Intl')
- with szSTRICTRI: 'TRUE' or 'FALSE'
- with any sz\*BLOCKSIZE: block size in bytes (i.e. '4096' for a 4k setting)

For a description of the 'AFieldDescList' parameter refer to TDoRxList.



## BDEPdoxCreate routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEPdoxCreate(ADataBase: TDataBase;  
                        const ATableName: TFileName;  
                        AFieldDescList: TDoRxList;  
                        const ATableLevel, ABlockSize: word;  
                        const StrictRI: boolean;  
                        const LangDrv: string);
```

### Description

Used to create Paradox tables with optional parameters. This method uses BDETableCreate for creating tables which you might want to do as well. You have to additionally specify the tabletype and you can supply any optional parameters in the same way as described with BDEOptRestruct (see above).

Advantages are that you dont have to supply any values if not needed (just empty brackets then...). For a description of the 'AFieldDescList' parameter refer to TDoRxList.

## BDEdBaseCreate routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEdBaseCreate(ADataBase: TDataBase;  
                        const ATableName: TFileName;  
                        AFieldDescList: TDoRxList;  
                        const ATableLevel, AMDXBlockSize,  
                        AMemoBlockSize: word;  
                        const LangDrv: string);
```

### Description

Used to create dBase tables with optional parameters. This method uses BDETableCreate for creating tables which you might want to do as well. You have to additionally specify the tabletype and you can supply any optional parameters in the same way as described with BDEOptRestruct (see above). Advantages are that you dont have to supply any values if not needed (just empty brackets then...). For a description of the 'AFieldDescList' parameter refer to TDoRxList.

## BDETableCreate routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDECreateTable(ADataBase: TDataBase;  
                        const ATableName: TFileName;  
                        const ATableType: TTableType;  
                        AFieldDescList: TDoRxList;  
                        const OptNames, OptValues: array of string);
```

### Description

Used to create local tables (either dBase or Paradox) with optional parameters. For the names of those params you can use predefined constants:

```
const  
    {for use with both dBase and Paradox:}  
    szLEVEL           = 'LEVEL';  
    szLANGDRV         = 'LANGDRIVER';  
    {for use with Paradox only:}  
    szBLOCKSIZE       = 'BLOCK SIZE';  
    szSTRICTRI        = 'STRICTINTEGRITY'; {note the spelling!}  
    {for use with dBase only:}  
    szMDXBLOCKSIZE    = 'MDX BLOCK SIZE';  
    szMEMOBLOCKSIZE   = 'MEMO FILE BLOCK SIZE';
```

Valid OptValues are

- with szLEVEL: Paradox '3' to '7', dBase '3' to '5'
- with szLANGDRV: the *short* name of the language driver (i.e. 'INTL' instead of 'Paradox Intl')
- with szSTRICTRI: 'TRUE' or 'FALSE'
- with any sz\*BLOCKSIZE: block size in bytes (i.e. '4096' for a 4k setting)

NOTE StrictRI with Paradox will not have any effect as long as no RI is set up. Methods BDEAddRIConstraint and BDEAddRIFromFile will both take care that this flag is correctly set up so you might want to ignore it on table creation.

Here's an example on how to create a Paradox table with table level 7 and language driver 'Intl850':

```
BDETableCreate(MyDataBase, MyTableName,  
              MyFieldDescList,  
              [szLEVEL, szLANGDRIVER],  
              ['7', 'INTL850']);
```

For a description of the 'AFieldDescList' parameter refer to TDoRxList.

## BDECloneTable routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDECloneTable(ASourceTable: TTable;  
                        ADataBase: TDataBase;  
                        const ANewTableName: TFileName;  
                        const APassword: TCaption;  
                        KeepRI: boolean);
```

### Description

Used to create an exact 'clone' of the source table including indices, valchecks, even security descriptors and optional RI as well. KeepRI determines whether RI constraints of the source table will be set up for the target table. You can pass in a password if you want the target table to be encrypted.

## BDEAddRIConstraint routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEAddRIConstraint(ATable: TTable;  
                             const OtherTable: TFileName;  
                             const RName: TCaption;  
                             const RIModeOp, RIDelOp: RINTQual;  
                             const StrictRI: boolean;  
                             const ThisTableFields,  
                             OtherTableFields: string);
```

### Description

NOTE RI is only set up on the dependent table, the 'master' link will be automatically created by the BDE!

Sets up a referential integrity check with the values supplied. The last two parameters are equal to a TTable's AddIndex FieldNames parameter, i.e. it's a string holding field names separated by semikola.

Possible values of RIDelOp and RIModeOp are: either rintRESTRICT or rintCASCADE.

## BDEDropRIConstraint routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEDropRIConstraint(ATable: TTable;  
                               const ARIName: string);
```

### Description

NOTE this method will only work as expected if you keep your RI names unique, which they don't have to be as far as the BDE is concerned.

Will drop the referential integrity check that matches ARIName.

## **BDEDropAllRIConstraints routine**

### **Unit**

BDEDoRxS

### **Declaration**

```
procedure BDEDropAllRIConstraints(ATable: TTable);
```

### **Description**

Will drop any referential integrity checks on ATable.

## **BDEGetRIList routine**

### **Unit**

BDEDoRxS

### **Declaration**

```
procedure BDEGetRIList(ATable: TTable; AList: TStrings);
```

### **Description**

This will first clear AList and then add all RI names for ATable to AList.



## BDEGetRIDEfsByName routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEGetRIDEfsByName(ATable: TTable;  
                             const ARIName: string;  
                             ARIDEfs: TStrings);
```

### Description

NOTE this method will only work as expected if you keep your RI names unique, which they don't have to be as far as the BDE is concerned.

This function returns INI-like entries in ARIDEfs. You can best get at them using a TStringList as ARIDEfs and make use of it's Values property. Use the following 'entries' to retrieve info on a certain RI:

Name

OtherTbl

FldCount

Type {cf below for possible values}

ModOp {cf below for possible values}

DelOp {cf below for possible values}

ThisTabFlds

OthTabFlds

Possible values of 'Type' include: either rintMASTER or rintDEPENDENT.

Possible values of DelOp and ModeOp are: either rintRESTRICT or rintCASCADE.

Using a TStringList you would retrieve an RI type like this:

```
MyType := ARIDEfs.Values('Type');
```

NOTE that all Values will be strings so you'll have to write some code to get them to be types, integers etc again.

## BDEGetRIDEfsByNumber routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEGetRIDEfsByNumber(ATable: TTable;  
                               const ARINumber: smallint;  
                               ARIDEfs: TStrings);
```

### Description

You'd most probably rather use BDEGetRIDEfsByName in your apps. This method here is used internally for writing definition files. It's surfaced nevertheless, maybe you can make use of it for certain tasks.

This function returns INI-like entries in ARIDEfs. You can best get at them using a TStringList as ARIDEfs and make use of it's Values property. Use the following 'entries' to retrieve info on a certain RI:

Name

OtherTbl

FldCount

Type

ModOp

DelOp

ThisTabFlds

OthTabFlds

Possible values of 'Type' include: either rintMASTER or rintDEPENDENT.

Possible values of DelOp and ModeOp are: either rintRESTRICT or rintCASCADE.

Using a TStringList you would retrieve an RI type like this:

```
MyType := ARIDEfs.Values('Type');
```

NOTE that all Values will be strings so you'll have to write some code to get them to be types, integers etc again.

## BDEAddRIFromFile routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEAddRIFromFile(ATable: TTable;  
                           const AFileName: TFileName);
```

### Description

Adds a referential integrity check from a table definition file. See also 'Wrappers' for an example on how to use it.

## BDEAddValchecksFromFile routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEAddValchecksFromFile(ATable: TTable;  
                                   const AFileName: TFileName);
```

### Description

Adds valchecks to a table from a definition file. See also the 'Wrappers' section for an example on how to use it.

## BDEGetValchecksByNumber routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEGetValchecksByNumber(ATable: TTable;  
                                   const AVchkNumber: smallint;  
                                   AValDefs: TStrings);
```

### Description

You'd most probably rather use BDEGetValChecksForField in your apps. This method here is used internally for writing definition files. It's surfaced nevertheless, maybe you can make use of it for certain tasks.

This function returns INI-like entries in AValDefs. You can best get at them using a TStringList as AValDefs and make use of it's Values property. Use the following 'entries' to retrieve info on a field's valchecks:

FieldNo

Required

Picture

LkUpTblName

LkUpType {cf below for possible values}

DefVal

MinVal

MaxVal

possible values of LkUpType are:

Lookup Type	Description
-------------	-------------

IkupNONE	The table has no lookup.
IkupPRIVATE	Only current field private.
IkupALLCORRESP	All corresponding no help.
IkupHELP	Only current field help and fill.
IkupALLCORRESPHELP	All corresponding help

Using a TStringList you would retrieve DefVal like this:

```
MyDefVal := AValDefs.Values('DefVal');
```

NOTE that all Values will be strings so you'll have to write some code to get them to be types, integers etc again.

## BDEDropValcheckByNumber routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEDropValcheckByNumber(ATable: TTable;  
                                   AValNumber: integer);
```

### Description

Will drop the valcheck in question.

NOTE due to a bug in BDE4.0 this can cause a GPF. Please get hold of 4.01 as fast as possible and/or think of implementing a version check before calling this routine.

See BDEGetIdapi32Version for a version check and BDEDropValFile for a possible workaround.

## **BDEDropAllValchecks routine**

### **Unit**

BDEDoRxS

### **Declaration**

```
procedure BDEDropAllValchecks(ATable: TTable);
```

### **Description**

Will drop any valcheck on ATable but will leave any RI intact.

NOTE due to a bug in BDE4.0 this can cause a GPF. Please get hold of 4.01 as fast as possible and/or think of implementing a version check before calling this routine.

See BDEGetIdapi32Version for a version check and BDEDropValFile for a possible workaround.



## BDEDropValFile routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEDropValFile(ATable: TTable);
```

### Description

This will simply delete the \*.val file of ATable. This can be used to work around a bug in BDE4.0 that causes GPF's when dropping valchecks. Make sure that you re-create any RI ATable might have had as the val file holds both valcheck and RI constraints.

## BDEAlterPrimary routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEAlterPrimary(ATable: TTable;  
                           const AIndexFields: string;  
                           const AIdxOptions: TIndexOptions);
```

### Description

This routine can be used for altering primary indices of Paradox tables. While you can easily drop and recreate a secondary index if you want to change, you cannot do so for primaries.

AIndexFields holds field names separated with semikola, like with AddIndex of TTable.

AIdxOptions parameter is used here to be compatible with evt newer versions of the BDE which might offer more flexible primary indices. At present the only valid options are ixPrimary and ixUnique, which is redundant but won't hurt.

## BDEAsciiImport routine

### Unit

BDEDoRxS

### Declaration

```
function BDEAsciiImport(ADestTable: TTable;  
                        const ASrcTableName, ASchemaFile: TFileName;  
                        const ASection: string;  
                        var ACount: LongInt): boolean;
```

### Description

This method has been originally written on demand, it's included here as I like the option it provides. For this import facility the schema file (cf ASCII.DRV.TXT somewhere in your Delphi dirs) doesn't need to be in the same directory as the text file to import from. This is useful if for example you want to import from floppy without copying the schema file to the floppy first (which can be a hassle with space checking etc).

It uses DBIOpenTable and DBIBatchmove internally, so you don't have to use TBatchMove and instead of a second TTable just it's file name is passed in. ASection refers to the section name in the schema file which by default matches the text file name w/o extension. Var ACount is used to return the number of records imported and to specify the number of records to be imported. If you want them all, ACount has to be set to 0 before calling this routine.

## BDEGetIdapi16Version routine

### Unit

BDEDoRxS

### Declaration

```
function BDEGetIdapi16Version: string;
```

### Description

This function returns the version of the 16bit BDE as a string. It uses the timestamp of IDAPI01.DLL to retrieve the version.

## **BDEGetIdapi32Version routine**

### **Unit**

BDEDoRxS

### **Declaration**

```
function BDEGetIdapi32Version: string;
```

### **Description**

This function returns the version of the 32bit BDE as a string. It uses the timestamp of IDAPI32.DLL to retrieve the version.

## CalcControlSize routine

### Unit

BDEDoRxS

### Declaration

```
procedure CalcControlSize(Sender: TComponent);
```

### Description

This is a simple routine that tries to calculate a form's or container control's size so that the right and lower borders fit the left border. That way you can have a form wider at designtime (for example to not clutter it with invisible controls but have them at an extended right border) without worrying about the forms appearance at runtime. It is used internally for the progress and ask-for-trim dialog, and it is used in the sample projects as well.

## CalcCenterPos routine

### Unit

BDEDoRxS

### Declaration

```
Procedure CalcCenterPos(PF, CF: TForm);
```

### Description

This routine calculates a form's position so that it centers on another form. If ParentForm is nil, Screen.ActiveForm is assumed to be the one to center on. It is used internally for the progress and ask-for-trim dialog, and it is used in the sample projects as well.

## Registering the Restructure Toolkit

By Mail

By CIS SWREG

Online from the Internet

### Why Register

Thank you for your interest in BDEDoRxS. If you find it useful in your own projects, please support the shareware concept. Much time and effort was put into providing a quality component along with adequate documentation, help, and demos.

Registered users will receive the latest registered version of BDEDoRxS, free on-line support, and - if they choose to - the source code.

### Prices

Prices depend on the way you decide to register. There are four different ways to register:

1) by CIS SWREG.

(the fastest way, but a bit more expensive as CIS takes it's fees)

2) by Mail with check included

(check **must** be drawn on a german bank!)

3) by Mail with IPMO included

(check your local post office for info on International Postal Money Orders)

4) by Mail and a bank transfer

(drop me mail first to get my bank account)

5) online from the Internet

Prices for option 1 are:

- w/o source: \$69.95

- with source: \$119.95

Prices for options 2 to 5 are:

- w/o source: \$65.95

- with source: \$112.95

German residents and people who want to choose option 4 please drop me mail prior to registering.

### Questions & Comments

Please refer questions or comments about BDEDoRxS to:

Reinhard Kalinke, [r\\_kalinke@compuserve.com](mailto:r_kalinke@compuserve.com)

For the latest version of my Delphi components have a look at my homepage at



[http://ourworld.compuserve.com/homepages/r\\_kalinke](http://ourworld.compuserve.com/homepages/r_kalinke)

## Registration by Mail

Select **Print Topic** from the **File** menu to print this form.

**Item** BDEDoRxS

**Price** See Registration for prices

Please register my copy of BDEDoRxS,

I am sending a check (drawn on agerman bank!) or international postal money  
orders for \$ \_\_\_\_\_

**Name:** \_\_\_\_\_

**Company:** \_\_\_\_\_

**Address1:** \_\_\_\_\_

**Address2:** \_\_\_\_\_

**City:** \_\_\_\_\_

**State:** \_\_\_\_\_ **Zip:** \_\_\_\_\_

**Country:** \_\_\_\_\_

**Phone:** \_\_\_\_\_ optional

**Email:** \_\_\_\_\_ optional

Please send completed form with payment to:

Reinhard Kalinke  
Jahnstr 45  
D-29549 Bad Bevensen  
Germany

## Registration by CIS SWREG

SWREG numbers are:

w/o source:     # **16541** (\$69,95)

with source:    # **16542** (\$119,95)

To register by SWREG start WinCim, GO SWREG and type in one of the above numbers (the appropriate one, of course...). Fill out the forms and you're done.

CIS will bill the registration fees with your next invoice.

## Registration online from the Internet

An Internet registration facility is set up at

<http://www.shareit.com>

They accept all major credit cards, money order and cash or Eurocheques (only in Europe).

The lower prices apply to this form of registration as well (\$65.95 resp \$112.95).

If you would like to register Restructure Toolkit for Delphi, you can do the registration online on the Internet at <http://www.shareit.com/programs/100813.htm> (w/o source) or /100814 (with source). Alternatively, you can go to <http://www.shareit.com> and enter the program number there: 100813 (w/o source) or 100814 (with source).

If you do not have access to the Internet, you can register via phone, fax or postal mail. Please print out the following form, and fax or mail it to:

Reimold&Schumann Internet Services  
ShareIt!  
Habsburgerring 3  
50674 Koeln  
Germany

Phone: +49-221-2407279  
Fax: +49-221-2407278  
E-Mail: [register@shareit.com](mailto:register@shareit.com)

Registration form for Restructure Toolkit for Delphi

Program No.: 100813

Last name: \_\_\_\_\_

First name: \_\_\_\_\_

Company: \_\_\_\_\_

Street and #: \_\_\_\_\_

City, State, postal code: \_\_\_\_\_

Country: \_\_\_\_\_

Phone: \_\_\_\_\_

Fax: \_\_\_\_\_

E-Mail: \_\_\_\_\_

How would like to receive the registration key/full version?

- e-mail - fax                      - postal mail

How would you like to pay the registration fee of \$65.95:

- credit card      - wire transfer    - EuroCheque - cash

Credit card information (if applicable)

Credit card: Visa - Eurocard/Mastercard - American Express - Diners Club

Card holder: \_\_\_\_\_

Card No.: \_\_\_\_\_

Date of Expiration : \_\_\_\_\_

Date / Signature \_\_\_\_\_

## Easy wrapper methods

Here are a few ready-to-paste 'wrapper' methods for use with your own apps. They are also included with the RESTRUC.DPR sample project. For the latest incarnations please always look at the sample project RESTRUC!

First one is for full-featured restructure and table creation. It handles tables, indices, RI and valchecks.

Second one is for 'simple' restructure and table creation. It only handles tables and indices.

A third one is for handling 'Index out of date' errors.

Number four and five handle the creation of table or index definition files.

And there is a number six, simply scanning a directory for def files.

## BDEDoRxS - Routines by Group

### Table Creation

[BDEPdoxCreate](#)  
[BDEdBaseCreate](#)  
[BDECreateTable](#)  
[BDECloneTable](#)  
[BDECreateTableFromFile](#)  
[BDESaveTableDefsToFile](#)

### Restructuring

[BDEGetFieldStructure](#)  
[BDEFieldRestruct](#)  
[BDEOptRestruct](#)  
[BDEFieldAndOptRestruct](#)  
[BDESaveTableDefsToFile](#)  
[BDERestructTableFromFile](#)

### Indices

[BDEDropAllIndices](#)  
[BDEEmergencyDropAllIndices](#)  
[BDEAddIndicesFromFile](#)  
[BDERecoverIndicesFromFile](#)  
[BDESaveIndexDefsToFile](#)  
[BDEAlterPrimary](#)  
[BDEPatchdBaseHeader](#)

### Valchecks

[BDEGetValchecksForField](#)  
[BDEGetValChecksByNumber](#)  
[BDEAddValchecksForField](#)  
[BDEAddValchecksFromFile](#)  
[BDEDropValFile](#)  
[BDEDropAllValchecks](#)  
[BDEDropValchecksForField](#)  
[BDEDropValchecksByNumber](#)

### Referential Integrity

[BDEGetRIDefsByName](#)  
[BDEGetRIDefsByNumber](#)  
[BDEAddRIConstraint](#)  
[BDEAddRIFromFile](#)  
[BDEDropAllRIConstraints](#)  
[BDEDropRIConstraint](#)

### Export/Import

[BDEAsciiImport](#)  
[BDEWriteMailMerge](#)

### Little Helpers

[BDEGetDBPath](#)  
[BDEGetTablePath](#)  
[BDEGetIdapi16Version](#)  
[BDEGetIdapi32Version](#)

[BDEGetLevel](#)  
[BDEGetLangDriver](#)  
[BDEGetPDXBlockSize](#)  
[BDEGetMDXBlockSize](#)  
[BDEGetMemoBlockSize](#)  
[BDEGetCFGDefaults](#)  
[BDEGetPDXUserList](#)  
[BDEGetPDXUserCount](#)  
[BDECopyTable](#)  
[BDERenameTable](#)  
[BDEPackTable](#)  
[BDEProtectTable](#)  
[BDEEmptyTable](#)  
[BDEIsTableEmpty](#)  
[BDEGetRecordNo](#)

**Form methods**

[CalcControlSize](#)  
[CalcCenterPos](#)



## EDoRxError type

### Unit

BDEDoRxS

### Declaration

```
EDoRxError = class(Exception);
private
  FTableName: TFileName;
  procedure SetTableName(Value: TFileName);
public
  property TableName: TFileName
    read FTableName
    write SetTableName;
end;

EDoRxProblem = class(EDoRxError);
EDoRxKeyViol = class(EDoRxError);
```

### Description

An exception of this type is raised when a problem table was created either due to data trimming with BDEFieldRestruct (will be EDoRxProblem then) or with records failing the RI constraint being set up with BDEAddRIConstraint (EDoRxKeyViol). The property TableName will hold the name of that table for further processing.

There are methods to write and process definition files so most of the time you would use a tool like 'Restructor' (sample project 'Restruc'). However, if you need or want to manually edit them you need to know the structure and some rules to obey.

Definition files are simply INI-files. The 'Table' section holds general information:

```
[Table]
Name=telekom.db
Type=PARADOX
FieldCount=6
RICount=1
ValCount=1
Level=5
LangDrv=intl
BlockSize=2048
StrictRI=0
IndexCount=1
RIDepCount=1
```

The 'Name' entry has to match the TableName property of the ATable parameter of all BDE\*FromFile methods at the time the file is processed, otherwise an exception will be raised.

When adding / removing any fields, indices, RI or valchecks make sure the corresponding '\*Count' entries match the new counts.

Although RI defs are written to the definition file for the master table as well (for information purposes), it is only processed for dependent tables (master link is established automatically by the BDE). So the entry 'RIDepCount' is important for RI creation, not 'RICount', which again is for information only.

BlockSize entries have to be bytes.

Field sections look like this:

```
[Field1]
Name=ANr
Type=A
Size=6
Pos=2
FieldID=2
```

Other than you would probably think it is not 'Pos' that determines the position of the fields in the new or restructured table but the counter added to the section name 'Field'. So after processing this definition file 'ANr' would be field 1 in the table, not field 2.

Field processing on restructure is done like this: as the BDE needs to know the *old* fieldposition on DBIDoRestructure input in order to find fields to move etc (remember you can alter any field property, BDE has to have some means to identify the fields internally) and as we don't know the old positions as the definition file is written (there might have been multiple restructures etc), the definition file is 'pre-processed'.

For Paradox everything is fine, the developers decided early to keep an invariant field ID that you can use to identify a field if you don't know about it's current name, position etc. So we use it to find the corresponding fields in the original table and write their current position to the 'Pos' entry. (You might ask why the BDE does not use field ID's with Paradox restructures instead of old field positions. Well, you will have to ask BI...)

For dBase at the time being we have to use the field name to identify the field. Thus the little limitation for dBase users.

There is nothing particular special with indices, RI checks and valchecks. Here's what they look like:

[Index4]

Name=ixAdrNr

Fields=AdrNr

Options=ixCaseInsensitive

[RiCheck1]

Name=AdrTelekom

OtherTbl=ADRESSE.DB

FldCount=1

Type=rintDEPENDENT

ModOp=rintCASCADE

DelOp=rintRESTRICT

ThisTabFlds=1

OthTabFlds=1

[ValCheck1]

FieldNo=2

Required=1

Picture={##[.##]}

LkUpTblName=

LkUpType=lkupNONE

DefVal=1

MinVal=

MaxVal=

To make a long story short: you currently cannot change field names when restructuring a dBase table. You can do anything else, moving, adding, deleting fields, change type and size, but you have to stick to the field names you once created. I guess that in most cases you wouldn't want to change field names anyway, in order not to have trouble with persistent fields

and FieldByName calls in your apps. So I hope this limitation is not too severe... now here's the reason why this limitation currently exists:

As the BDE needs to know the *old* fieldposition on DBIDoRestructure input in order to find fields to move etc (remember you can alter any field property, BDE has to have some means to identify the fields internally) and as we don't know the old positions as the definition file is written (there might have been multiple restructures etc), the definition file is 'pre-processed'.

For Paradox everything is fine, the developers decided early to keep an invariant field ID that you can use to identify a field if you don't know about its current name, position etc. So we use it to find the corresponding fields in the original table and write their current position to the 'Pos' entry. (You might ask why the BDE does not use field ID's with Paradox restructures instead of old field positions. Well, you will have to ask BI...)

For dBase at the time being we have to use the field name to identify the field. Thus this little limitation for dBase users. (Quote of a Beta-tester re this limitation: 'Well, they're used to limits anyway, aren't they? <g,d&r>'). I would start setting sth up to overcome this limit if anyone comes and says: 'Hey, I would really like to register full source, but this one really sucks...'. I would

consider it even more if that person came up with a neat idea on how to do it (oh, I do have some ideas but they either require some work from the user or they are not safe enough. In any case, at the moment I don't really like one of them...).

HOWEVER, using field ID's with Paradox can be dangerous as well. It's only safe if the table to be restructured all descend from the same original table or if it was created in exactly the same way.

Consider a simple table, created by the BDEDoRx sample, DBD or whatever:

No     I

Item A10

'No' will have the field ID 1, 'Item' ID 2. Now you decide to update your app and insert a field 'Pos'. It will have ID 3 in the restructured table. However, for new customers, when the tables are created at runtime, 'Pos' will have ID 2 and 'Item' ID 3. With the next update you might for example want to change the size of 'Item' to A20. Everything works fine for your 'old' customers, but for customers who joined at your first update the restructure routine will try and change type and size of 'Pos' instead of 'Item'! They will either be prompted with an 'invalid field transformation', or, if the types are compatible, the wrong field is changed and your app will most probably crash or not work as designed.

A few scenarios:

You distribute your apps with tables and don't make use of runtime creation. You keep a copy of those

tables at your site and use them for all restructures. You did never delete a table and recreated it. Compare by field ID's will work.

You distribute your apps without tables and make use of runtime creation. You keep a copy of those tables - created using the same tools as at your client's site! - at your site and use them for all restructures. You did never delete a table and recreated it. Compare by field ID's will work.

One of the above scenarios applies. Your app is being updated one or more times and new customers have their tables created in a different way the earlier customers. Compare by field ID's will only work if you keep sets of all updates and provide different def files depending on the version the customers first created their tables with.

Of course you can have some of the tables in your database use field ID compare and some field name compare. I would recommend that you only use field ID compare when you want to change the name of a field. That's why the Restruct sample defaults to field name compare.

## Sample projects

There are four sample projects included with this unit, BDEDoRx, Restructor and two enduser oriented samples. The first one is focused on the 'core' routines while 'Restructor' is a demo on the 'file based' routines, implementing table and index definition files.

NOTE When compiling the samples or a project of your own using BDEDoRxS methods with Delphi 1 tests seem to indicate that you better increase stack size to 24 or even 32k.

### BDEDoRx

Resides in the BDEDoRx directory of the archive file. It's an interactive tool for restructuring tables as well as retrieving and setting index and RI properties. Don't miss all the little PopUp's!

In the structure grid, Ctrl-Ins inserts a row, Ctrl-Del deletes and Enter, Ctrl-Down or Ctrl-PgDn adds one. Moving fields is done by mouse drag and drop. There's not too much validation coded here, you will have to make sure yourself to supply valid field data. The interface field data is taken from the DBD concept here but it should be easy to rewrite the FLDDescToList and ListToFLDDDesc procs to fit any other terminology - if you have the source, that is...

### Restructor

RESTRUC.DPR can be found in the RestDev1 subdir of the archive file.

'Restructor' is an interactive tool for creating and processing table and index definition files. It's interface should be sort of self-explaining, although perhaps the index checkbox at the right of the form needs some explanation. If checked, Restructor creates and processes index definition files only, if not checked, 'complete' table definition files will be created/processed.

There are two sorts of definition files in case you don't want the 'complete' ones to remain at your clients site. Index definition files enable you to handle 'Index out of date' errors nevertheless.

Restructor uses .DBI as extension for table definition files and .DBX for index definition files, but this is by no means a must. You can safely use any extension you like with your apps.

You might want to use Restructor as a sample or base for your own custom client site restructor.

NOTE: You are not allowed to give it away as a whole, that is including any method to write def files!

### UserRest

Source is in archive subdir RestUser. This is a lite version of Restruc that processes any def files it finds in a certain subdir. This one may be given away.

### AutoRest

Source is in archive subdir AutoRest. It's an even 'liter' version that needs no interaction with the end user. You can use it as a standalone exe, called by the main app with WinExecAndWait or you can change the way it takes parameters and include the form in your app. I prefer first option since you need to deploy it just once and your app is not blown up.

## BDEGetValchecksForField routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEGetValchecksForField(ATable: TTable;  
                                   const AFieldNo: smallint;  
                                   AValDefs: TStrings);
```

### Description

This function returns INI-like entries in AValDefs. You can best get at them using a TStringList as AValDefs and make use of it's Values property. Use the following 'entries' to retrieve info on a field's valchecks:

FieldNo  
Required  
Picture  
LkUpTblName  
LkUpType {cf below for possible values}  
DefVal  
MinVal  
MaxVal

possible values of LkUpType are:

Lookup Type	Description
-----	
IkupNONE	The table has no lookup.
IkupPRIVATE	Only current field private.
IkupALLCORRESP	All corresponding no help.
IkupHELP	Only current field help and fill.
IkupALLCORRESPHELP	All corresponding help

Using a TStringList you would retrieve DefVal like this:

```
MyDefVal := AValDefs.Values['DefVal'];
```

NOTE that all Values will be strings so you'll have to write some code to get them to be types, integers etc again.



## BDEWriteMailMerge routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEWriteMailMerge(ATable: TTable;  
    const AMergeFile: TFileName;  
    const ASeparator, ADelimiter, ALineEnd: string;  
    BDelimitEmpty, BWriteHeader, BOEM: boolean;  
    AFieldList: TStrings;  
    {$IFDEF WIN32}  
    AProgressBar: TProgressBar);  
    {$ELSE}  
    AProgressBar: TGauge);  
    {$ENDIF}
```

### Description

This routine gives you a lot more of control than the text driver offers when you want to write an acci export esp. for mail merge puposes. You can control not only delimiter and separator but a special char or string at a lines end as well. You can determine whether empty fields will be delimited as well, whether a header (containg the field names) will be written and whether any text will be OEM-converted for use with DOS apps. You can also pass in a progressbar/gauge that will display the progress. If you don't want progress display the simply pass **nil** for this parameter.

## General hints and trap wires

### BDE4.0 Valcheck Bug

First the most dangerous gotcha: there is an incredible bug in BDE4.0 concerning valchecks. Dropping valchecks most likely will cause a GPF which of course is not desirable. There is an internal workaround in BDEDoRxS but it is a crude one (the only one I could find, though). All methods dropping valchecks check for BDE4.0 and if the check is positive they don't drop the valcheck but 'null' it (with FillChar(0)). That way they are not dropped but don't have any effect either. However, your val file will never shrink but only grow that way. I have not discovered any other drawback of this 'solution' yet.

If you want to implement a custom workaround that I think is 'cleaner' have a look at method [BDEDropValFile](#).

### Index issues

BDEDoRxS offer some methods not only for 'normal' index processing but for handling 'Index out of date' errors as well. These don't need to open the table. Have a look at methods [BDEEmergencyDropAllIndices](#), [BDERecoverIndicesFromFile](#) and [BDESaveIndexDefsToFile](#). Use an exception handler like this for catching 'Index out of date' errors:

```
on E:EDBEngineError do
begin
  for i:=0 to pred(E.ErrorCount) do
    if (E.Errors[i] = DBIERR_INDEXOUTOFDATE) then
      begin
        {steps to fix the error}
        Break;
      end;
  end;
end;
```

### dBase index issues

There are two ways to deal with missing or corrupted MDX files. First one is using [BDEPatchdBaseHeader](#) which will access the DBF directly and clear the bit that indicates the presence of the MDX. Second, which is more elegant from my point of view, is using [TDBFTable](#) instead of TTable in your apps. This will set up a cbINPUTREQ callback that gets triggered when you try to open a DBF with missing or corrupt MDX and will then tell the BDE to detach the MDX file.

Note that with dBase index corruption is detected very late: create a single field table with let's say a char(50) field. Create an index on the field. Copy the (empty) index file to another name or location. Enter some 20 values in the table. Now close it and copy the old MDX back to the table, overwriting the existing one. Open the table - no error. Switch to the index - the table appears to be empty. So much for data integrity with dBase tables... try this with Paradox and the famous 'Index out of date' will appear when having entered just a single value.

### Use TDataBases

I recommend using a TDataBase component when dealing with BDEDoRxS methods. I have found that things run smoother that way and, of course, some methods need a TDataBase as a parameter anyway.

### Use at least a separate form / data module

... if not a separate application to do the restructuring etc. That way the methods don't interfere with active TTables in other forms and your app doesn't carry code around that is seldom used.

### **Network issues**

As almost any non-retrieval routines needs exclusive access to the tables, you should try and make sure that noone else uses these tables when a restructure is about to be performed.

The whole package implement sort of a three layer concept. Layer one has all the 'core' methods that implement the restructure etc for Delphi by dealing with low level BDE API calls. Most of these methods are 'surfaced' to give you the ability to use them, too. Some of them are fairly easy to use, like BDEAddRIConstraint, but others, like BDEFieldRestruct, are easier than DBIDoRestructure but still require some skill and care and are still sort of a mess to code.

Layer two uses these methods to implement all the 'BDE\*FromFile' and 'BDE\*ToFile' methods. These are easier to use and provide you with the ability to do full-featured restructure, table creation, index, valcheck and RI handling at your and your client's site.

Layer three consist of the 'wrappers' that you can find in the sample project RESTRUC.DPR and in this help file as well (they may differ slightly, so please have a look at both locations). These are ready-to-paste procedures you can use to create your custom restruct / table creation tool and you custom index error tool. You might as well use them as templates to write your own '3rd layer' wrappers.

Layer one methods have been around for some time now and have been used in 'real world' applications, too. I always wanted to write a component based on them but now I find that the current approach is even easier to use and requires less coding from the developer than a component approach. I hope you can share this view; if not, drop me a mail...

## BDEAddValcheckForField

### Unit

BDEDoRxS

### Declaration

```
procedure BDEAddValcheckForField(ATable: TTable;  
                                AFieldNo: integer;  
                                AValList: TStrings);
```

### Description

Will add the constraints supplied with AValList to the field determined by AFieldNo. For the syntax requirements of AValList have a look at [BDEGetValcheckForField](#).

## BDEDropValcheckForField

### Unit

BDEDoRxS

### Declaration

```
procedure BDEDropValchecksForField(ATable: TTable;  
                                   AFieldNo: integer;
```

### Description

Drops all current valchecks for the field specified by AFieldNo. AFieldNo represents the field's physical fieldno. You can get it by for example calling

```
MyFieldNo := TableX.FindField('MyField').FieldNo
```

## BDECopyTable routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDECopyTable(ATable: TTable;  
                       const ADestTableName: TFileName;  
                       DoOverwrite: boolean);
```

### Description

Copies a table including all 'family members' (indices, val file etc) to a new name / location. DoOverWrite determines whether the table will be overwritten if it already exists.

K BDERenameTable

## BDERenameTable routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDERenameTable(ATable: TTable;  
                        const ANewTableName: TFileName);
```

### Description

Renames a table. Adds nothing to TTable.RenameTable, in here only for completeness...



## BDEPackTable routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEPackTable(ATable: TTable);
```

### Description

Packs a table, removing deleted records and reusing their space.

## BDEProtectTable routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEProtectTable(ATable: TTable;  
                           const APassword: string);
```

### Description

Protects a table with the given password. If APassword is an empty string, it will unprotect a protected table (current password must be provided by Session.AddPassword).

## **BDEEmptyTable routine**

### **Unit**

BDEDoRxS

### **Declaration**

```
procedure BDEEmptyTable(ATable: TTable);
```

### **Description**

Empties a table; adds recommended DBISaveChanges to TTable.EmptyTable.

## DoSimpleRestructureFromFile routine

You have to set up a list of files to process in AFileList. If you don't need any progress bars or status panels please delete the lines referencing them.

```
{takes care of updating the progressbar}
procedure SetProgress({$IFDEF WIN32}AProgressBar: TProgressBar;
                     {$ELSE}AProgressBar: TRect;{$ENDIF}
                     const Count: integer; Reset: boolean);
begin
  if (AProgressBar = nil) then Exit;
  if Reset then
    begin
      {$IFDEF WIN32}
      AProgressBar.Position := 0;
      AProgressBar.Max := Count;
      {$ELSE}
      {AProgressBar.Progress := 0;
      AProgressBar.MaxValue := Count; {}
      {$ENDIF}
    end
  else
    {$IFDEF WIN32}
    AProgressBar.Position := Count;
    {$ELSE}
    {AProgressBar.Progress := Count; {}
    {$ENDIF}
  end;

{processes table defs for field restructure and indices only
(no RI or Val processing)}
procedure DoSimpleRestructureFromFile(AFileList: TStringList;
                                     ADataBase: TDataBase;
                                     ATable: TTable;
                                     {$IFDEF WIN32}
                                     AProgressBar: TProgressBar;
                                     {$ELSE}
                                     AProgressBar: TRect;
                                     {$ENDIF}
                                     AStatusPanel: TPanel;
                                     const DoCreateTables,
                                     DoCreateIndices,
                                     DoDeleteDefs: boolean);
var i,j,iProg: integer;
    DefFile, DBFile: TFileName;
    DoIndex: boolean;
    ExcValue: DBIResult;
begin
  Screen.Cursor := crHourGlass;
  try
    DoIndex := DoCreateIndices;
    iProg := 0;
    SetProgress(AProgressBar,AFileList.Count*(1+ord(DoIndex)),True);
    for i:=0 to pred(AFileList.Count) do
      begin
        DefFile := AFileList.Strings[i];
```

```

with TIniFile.Create(DefFile) do
try
  ATable.TableName := ReadString('Table','Name','');
  AStatusPanel.Caption := 'processing: '+ATable.TableName;
  AStatusPanel.Update;
  try
    ATable.Open;
    BDERestructTableFromFile(ATable, DefFile);
    inc(iProg);
    SetProgress(AProgressBar,iProg,False);
  except
    on E:EDBEngineError do
      begin
        DoIndex := DoCreateIndices and DoCreateTables;
        {if table does not exist:}
        for j:=0 to pred(E.ErrorCount) do
          begin
            if DoCreateTables
              and (E.Errors[j].ErrorCode = DBIERR_NOSUCHTABLE) then
              begin
                BDECreateTableFromFile(ADatabase, DefFile);
                inc(iProg);
                SetProgress(AProgressBar,iProg,False);
                ATable.Open;
                DBISaveChanges(ATable.Handle);
                Break;
              end
            else raise;
          end;
        end;
        else raise;
      end;
    end;
    if DoIndex then
      BDEAddIndicesFromFile(ATable, DefFile);
      inc(iProg);
      SetProgress(AProgressBar,iProg,False);
    finally
      Free;
      ATable.Close;
    end;
  end;
  AStatusPanel.Caption := 'Done!';
  AStatusPanel.Update;
finally
  Screen.Cursor := crDefault;
end;
if DoDeleteDefs then
  begin
    for i:=0 to pred(AFileList.Count) do
      SysUtils.DeleteFile(AFileList.Strings[i]);
    end;
  end;
end;
end;

```

## DoRecoverIndicesFromFile routine

You have to set up a list of files to process in AFileList. If you don't need any progress bars or status panels please delete the lines referencing them.

```
{takes care of updating the progressbar}
procedure SetProgress({$IFDEF WIN32}AProgressBar: TProgressBar;
                     {$ELSE}AProgressBar: TRect;{$ENDIF}
                     const Count: integer; Reset: boolean);
begin
  if (AProgressBar = nil) then Exit;
  if Reset then
    begin
      {$IFDEF WIN32}
      AProgressBar.Position := 0;
      AProgressBar.Max := Count;
      {$ELSE}
      {AProgressBar.Progress := 0;
      AProgressBar.MaxValue := Count; {}
      {$ENDIF}
    end
  else
    {$IFDEF WIN32}
    AProgressBar.Position := Count;
    {$ELSE}
    {AProgressBar.Progress := Count;
    {$ENDIF}
  end;

{processes index defs for a list of files in case of
index errors ('Index out of date')}
procedure DoRecoverIndicesFromFile(AFileList: TStringList;
                                  ADB: TDataBase;
                                  ATable: TTable;
                                  {$IFDEF WIN32}
                                  AProgressBar: TProgressBar;
                                  {$ELSE}
                                  AProgressBar: TRect;
                                  {$ENDIF}
                                  AStatusPanel: TPanel;
                                  const DoDeleteDefs: boolean);
var i,iProg,iPass,iPasses: integer;
    DefFile, DBFile: TFileName;
    Res: integer;
    FileRec: TSearchRec;
begin
  Screen.Cursor := crHourGlass;
  try
    iProg := 0;
    SetProgress(AProgressBar,AFileList.Count,True);
    for i:=0 to pred(AFileList.Count) do
      begin
        DefFile := AFileList.Strings[i];
        with TIniFile.Create(DefFile) do
          try
            ATable.TableName := ReadString('Table','Name','');
```

```

        AStatusPanel.Caption := 'recovering indices: '+ATable.TableName;
        AStatusPanel.Update;
        BDERecoverIndicesFromFile(ADB, ATable.TableName, DefFile);
        inc(iProg);
        SetProgress(AProgressBar,iProg,False);
    finally
        Free;
    end;
end;
end;
AStatusPanel.Caption := 'Done!';
AStatusPanel.Update;
finally
    Screen.Cursor := crDefault;
end;
if DoDeleteDefs then
begin
    for i:=0 to pred(AFileList.Count) do
        SysUtils.DeleteFile(AFileList.Strings[i]);
    end;
end;
end;

```

## DoWriteTableDefsToFile routine

You have to set up a list of tables to process in AFileList. If you don't need any progress bars or status panels please delete the lines referencing them.

```
{takes care of updating the progressbar}
procedure SetProgress({$IFDEF WIN32}AProgressBar: TProgressBar;
                     {$ELSE}AProgressBar: TRect;{$ENDIF}
                     const Count: integer; Reset: boolean);
begin
  if (AProgressBar = nil) then Exit;
  if Reset then
    begin
      {$IFDEF WIN32}
      AProgressBar.Position := 0;
      AProgressBar.Max := Count;
      {$ELSE}
      {AProgressBar.Progress := 0;
      AProgressBar.MaxValue := Count; {}
      {$ENDIF}
    end
  else
    {$IFDEF WIN32}
    AProgressBar.Position := Count;
    {$ELSE}
    {AProgressBar.Progress := Count; {}
    {$ENDIF}
  end;

{writes table defs for a list of tables}
procedure DoWriteTableDefsToFile(AFileList: TStrings;
                                ATable: TTable;
                                const AVersion: string;
                                DoUseFieldIDs: boolean;
                                {$IFDEF WIN32}
                                AProgressBar: TProgressBar;
                                {$ELSE}
                                AProgressBar: TRect;{}
                                {$ENDIF}
                                AStatusPanel: TPanel);
var i,iProg: integer;
    DBFile, DefFile: TFileName;
begin
  Screen.Cursor := crHourGlass;
  try
    iProg := 0;
    SetProgress(AProgressBar,AFileList.Count,True);
    for i:=0 to pred(AFileList.Count) do
      begin
        DBFile := AFileList.Strings[i];
        DefFile := ChangeFileExt(DBFile, '.dbi');
        with TIniFile.Create(DefFile) do
          try
            ATable.TableName := ExtractFileName(DBFile);
            ATable.Open;
            AStatusPanel.Caption := 'creating table def: '
```



```

+DefFile;
AStatusPanel.Update;
BDESaveTableDefsToFile(ATable, DefFile);
if (AVersion > '') then
    WriteString('Table','Version',AVersion);
if DoUseFieldIDs then
    WriteString('Table','FieldCompare','ByFieldID')
else
    WriteString('Table','FieldCompare','ByFieldName');
finally
    Free;
    ATable.Close;
end;
inc(iProg);
SetProgress(AProgressBar,iProg,False);
end;
AStatusPanel.Caption := 'Done!';
AStatusPanel.Update;
finally
    Screen.Cursor := crDefault;
end;
end;

```

## DoWriteIndexDefsToFile routine

You have to set up a list of tables to process in AFileList. If you don't need any progress bars or status panels please delete the lines referencing them.

```
{takes care of updating the progressbar}
procedure SetProgress({$IFDEF WIN32}AProgressBar: TProgressBar;
                     {$ELSE}AProgressBar: TRect;{$ENDIF}
                     const Count: integer; Reset: boolean);
begin
  if (AProgressBar = nil) then Exit;
  if Reset then
    begin
      {$IFDEF WIN32}
      AProgressBar.Position := 0;
      AProgressBar.Max := Count;
      {$ELSE}
      {AProgressBar.Progress := 0;
      AProgressBar.MaxValue := Count; {}
      {$ENDIF}
    end
  else
    {$IFDEF WIN32}
    AProgressBar.Position := Count;
    {$ELSE}
    {AProgressBar.Progress := Count; {}
    {$ENDIF}
  end;

{writes index defs only for a list of tables}
procedure DoWriteIndexDefsToFile(AFileList: TStrings;
                                ATable: TTable;
                                const AVersion: string;
                                {$IFDEF WIN32}
                                AProgressBar: TProgressBar;
                                {$ELSE}
                                AProgressBar: TRect;
                                {$ENDIF}
                                AStatusPanel: TPanel);

var i,iProg: integer;
    DBFile, DefFile: TFileName;
begin
  Screen.Cursor := crHourGlass;
  try
    iProg := 0;
    SetProgress(AProgressBar,AFileList.Count,True);
    for i:=0 to pred(AFileList.Count) do
      begin
        DBFile := AFileList.Strings[i];
        DefFile := ChangeFileExt(DBFile, '.dbx');
        with TIniFile.Create(DefFile) do
          try
            ATable.TableName := ExtractFileName(DBFile);
            ATable.Open;
            AStatusPanel.Caption := 'creating index def: '
                                   +DefFile;
          finally
            ATable.Close;
          end;
        end;
        iProg := i + 1;
        SetProgress(AProgressBar,iProg,True);
      end;
    end;
  finally
    Screen.Cursor := crDefault;
  end;
end;
```

```
    AStatusPanel.Update;
    BDESaveIndexDefsToFile(ATable, DefFile);
    if (AVersion > '') then
        WriteString('Table','Version',AVersion);
    finally
        Free;
        ATable.Close;
    end;
    inc(iProg);
    SetProgress(AProgressBar,iProg,False);
end;
AStatusPanel.Caption := 'Done!';
AStatusPanel.Update;
finally
    Screen.Cursor := crDefault;
end;
end;
```

## DoRestructureFromFile routine

You have to set up a list of files to process in AFileList. If you don't need any progress bars or status panels please delete the lines referencing them. If you don't want Progressbars you can simply pass nil or delete the lines as well.

```
{takes care of updating the progressbar}
procedure SetProgress({$IFDEF WIN32}AProgressBar: TProgressBar;
                     {$ELSE}AProgressBar: TRect;{$ENDIF}
                     const Count: integer; Reset: boolean);
begin
  if (AProgressBar = nil) then Exit;
  if Reset then
    begin
      {$IFDEF WIN32}
      AProgressBar.Position := 0;
      AProgressBar.Max := Count;
      {$ELSE}
      {AProgressBar.Progress := 0;
      AProgressBar.MaxValue := Count; {}
      {$ENDIF}
    end
  else
    {$IFDEF WIN32}
    AProgressBar.Position := Count;
    {$ELSE}
    {AProgressBar.Progress := Count; {}
    {$ENDIF}
  end;

{processes table defs with the whole range of current
 options (indices, RI, Val)}
procedure DoRestructureFromFile(AFileList: TStrings;
                               ADataBase: TDataBase;
                               ATable: TTable;
                               {$IFDEF WIN32}
                               AProgressBar: TProgressBar;
                               {$ELSE}
                               AProgressBar: TRect;
                               {$ENDIF}
                               AStatusPanel: TPanel;
                               const DoCreateTables,
                               DoCreateIndices,
                               DoCreateRefInt,
                               DoCreateValchecks,
                               DoDeleteDefs: boolean);
var i,j,iProg,iPass,iPasses: integer;
    DefFile, DBFile: TFileName;
    DoIndex: boolean;
    ActionStr: string;
    ExcValue: DBIResult;
begin
  Screen.Cursor := crHourGlass;
  try
    DoIndex := DoCreateIndices;
    if (DoCreateRefInt or DoCreateValchecks) then
```

```

    iPasses := 2 else iPasses := 1;
SetProgress(AProgressBar,AFileList.Count*(1+ord(DoIndex)),True);
for iPass:=1 to iPasses do
begin
    iProg := 0;
    SetProgress(AProgressBar,iProg,False);
    if (iPass = 1) then
        ActionStr := 'processing: '
    else
        ActionStr := 'creating RI and/or ValChecks: ';
    for i:=0 to pred(AFileList.Count) do
    begin
        DefFile := AFileList.Strings[i];
        with TIniFile.Create(DefFile) do
        try
            ATable.TableName := ReadString('Table','Name','');
            AStatusPanel.Caption := ActionStr+ATable.TableName;
            AStatusPanel.Update;
            if (iPass = 2) then
            begin
                ATable.Open;
                {'Bugfix' BDE4.0:}
                if MainForm.FDeleteVals then
                    BDEDropValFile(ATable);
                if DoCreateRefInt then
                    {dropping existing RI is included
                     with below function}
                    BDEAddRIFromFile(ATable, DefFile);
                inc(iProg);
                SetProgress(AProgressBar,iProg,False);
                if DoCreateValchecks then
                    {dropping existing val is included
                     with below function}
                    BDEAddValchecksFromFile(ATable, DefFile); {}
                inc(iProg);
                SetProgress(AProgressBar,iProg,False);
                Continue;
            end
            else
            try
                ATable.Open;
                BDERestructTableFromFile(ATable, DefFile);
                inc(iProg);
                SetProgress(AProgressBar,iProg,False);
            except
                on E:EDBEngineError do
                begin
                    DoIndex := DoCreateIndices and DoCreateTables;
                    {if table does not exist:}
                    for j:=0 to pred(E.ErrorCount) do
                    begin
                        if DoCreateTables
                        and (E.Errors[j].ErrorCode = DBIERR_NOSUCHTABLE) then
                        begin
                            BDECreateTableFromFile(ADataBase, DefFile);
                            inc(iProg);
                            SetProgress(AProgressBar,iProg,False);

```

```

        ATable.Open;
        DBISaveChanges(ATable.Handle);
        Break;
    end
    else raise;
end;
end;
else raise;
end;
if DoIndex then
    {dropping existing indices is included
    with below function}
    BDEAddIndicesFromFile(ATable, DefFile);
    inc(iProg);
    SetProgress(AProgressBar,iProg,False);
finally
    Free;
    ATable.Close;
end;
end;
end;
AStatusPanel.Caption := 'Done!';
AStatusPanel.Update;
finally
    Screen.Cursor := crDefault;
end;
if DoDeleteDefs then
begin
    for i:=0 to pred(AFileList.Count) do
        SysUtils.DeleteFile(AFileList.Strings[i]);
    end;
end;
end;

```

## DoScanDirForFiles routine

Used to scan a directory for tables or def files. Use 'DB' or 'DBF' as AExt when scanning for tables. All the demos use 'DBI' for table and 'DBX' for index def files, however, this is not a must. You can use any extension you like but don't forget to change all occurrences in your source if you are using any of these wrapper routines.

```
{scans a dir for files with extension AExt and writes them
 into a list for further processing}
function DoScanDirForFiles(const ADir,AExt: TFileName;
                           AList: TStrings): integer;
var FileRec: TSearchRec;
    Res: integer;
begin
    AList.Clear;
    Res := SysUtils.FindFirst(ADir+'*.'+AExt, 0, FileRec);
    while Res = 0 do
    begin
        AList.Add(FileRec.Name);
        Res := SysUtils.FindNext(FileRec);
    end;
    SysUtils.FindClose(FileRec);
    Result := AList.Count;
end;
```

## BDEPatchdBaseHeader routine

### Unit

BDEDoRxS

### Declaration

```
procedure BDEPatchdBaseHeader(const AFileName: TFileName);
```

### Description

Used to patch a dBase table's header if the MDX index file has become corrupt. Is used internally by the index recovering routines, so in most cases you do not need to employ it yourself. The routine needs a fully qualified filename. However, using [TDBFTable](#) is a much cleaner way of detaching the MDX file.



This unit is using two dialog forms (BDEDoRxT for getting the trim options if field data may have to be truncated and BDEDoRxP for showing progress info when restructuring large tables). You can modify the text of BDEDoRxT by directly entering it in it's memo vomponent. Set the const 'FieldLineIndex' to the index of the line that is to hold the name of the field in question. The text for BDEDoRxP is directly supplied by the BDE, however you can set up const 'ProgressDlgLastMessage' to display some custom text when the restructuring is finished.

## BDEGetCFGDefaults

### Unit BDEDoRxS

#### Declaration

```
procedure BDEGetCFGDefaults;
```

#### Description

A call to this procedure opens the CFG file for the current session and retrieves the default values for creating and restructuring tables. It has no parameters since the default values are written to typed constants which are included with BDEDoRxS. That way we only need to call it once and have the values at hand nevertheless. It is called automatically when BDEDoRxS is initialized, however, if the session is for some reason not active at that time you might have to call once from your app before doing any restructure / table creation. Without a successful call the constants are initialized as follows:

```
{cfg default values}
szDBFLevel: string      = '5';
szDBFMDXBlockSize: string = '1024';
szDBFMemoBlockSize: string = '1024';
szDBFLangDrv: string    = 'ASCII';

szPDXLevel: string      = '5';
szPDXBlockSize: string  = '2048';
szPDXStrictRI: string   = 'TRUE';
szPDXLangDrv: string    = 'ASCII';
```

## BDEGetMemoBlockSize

### Unit

BDEDoRxS

### Declaration

```
function BDEGetMemoBlockSize(ATable: TTable): integer;
```

### Description

A call to this function returns the current block size of the DBT file associated with that table in bytes. As the BDE does not supply a proper way to retrieve this we have to open the file and have a look at the header. When the table is opened exclusive at the time of the call we first need to close it, set Exclusive to false and close and reopen it in old mode again. All this and a Disable-/EnableControls is included but you might want to take some extra steps to avoid any flicker on a status panel for instance.

## BDEGetMDXBlockSize

### Unit

BDEDoRxS

### Declaration

```
function BDEGetMDXBlockSize(ATable: TTable): integer;
```

### Description

Returns the current block size settings for the MDX file associated with the table in bytes.

## BDEGetLevel

### Unit

BDEDoRxS

### Declaration

```
function BDEGetLevel(ATable: TTable): integer;
```

### Description

Returns the table level or version of ATable.

## BDEGetPDXBlockSize

### Unit

BDEDoRxS

### **Declaration**

```
function BDEGetPDXBlockSize(ATable: TTable): integer;
```

### **Description**

Returns the current block size of ATable. Valid for Paradox tables only.

## BDEGetLangDriver

### Unit

BDEDoRxS

### Declaration

```
function BDEGetLangDriver(ATable: TTable): string;
```

### Description

Returns the short name of the current language driver for ATable.

## **BDEGetPDXUserList**

### **Unit**

[BDEDoRxS](#)

### **Declaration**

```
procedure BDEGetPDXUserList (AList: TStrings);
```

### **Description**

This procedure will clear AList and then add all the users that share a common net file at the moment. Unfortunately there is no equivalent with dBase for this functionality.



**Unit**

BDEDoRxS

**Declaration**

```
function BDEIsTableEmpty(ADataBase: TDataBase;  
    const ATableName, ATableType: string): boolean;
```

**Description**

Returns whether a table has records or not without opening the table within the VCL which would cause a performance hit (creation of runtime field objects etc).

## Download Areas

On the Internet there might be several places you can download my tools and components. However, some of those places might offer out-of-date versions. For the latest version of my tools and components always check

CompuServe Forum GO DELPHI

CompuServe Forum GO BDELPHI

and, on the Internet, my homepage at

[http://ourworld.compuserve.com/homepages/r\\_kalinke](http://ourworld.compuserve.com/homepages/r_kalinke)

If you want to drop me a letter here's my address:

Reinhard Kalinke  
Jahnstr 45  
D-29549 Bad Bevensen  
Germany

## Disclaimer

...and here's the inevitable:

### DISCLAIMER

The author cannot and does not warrant that any functions contained in the software will meet your requirements, or that its operations will be error free. The entire risk as to the software performance or quality, or both, is solely with the user and not the author. You assume responsibility for the selection of the component to achieve your intended results, and for the installation, use, and results obtained from the software.

The author makes no warranty, either implied or expressed, including without limitation any warranty with respect to this software documented here, its quality, performance, or fitness for a particular purpose. In no event shall the author be liable to you for damages, whether direct or indirect, incidental, special, or consequential arising out the use of or any defect in the software, even if the author has been advised of the possibility of such damages, or for any claim by any other party.

All other warranties of any kind, either express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose, are expressly excluded.

Delphi and Paradox are trademarks of Borland International.

## **BDEGetRecordNo routine**

### **Unit**

[BDEDoRxS](#)

### **Declaration**

```
function BDEGetRecordNo (ADataset: TDataSet): longint;
```

### **Description**

Returns the record no of the current record for the given dataset. This is of interest with Delphi v1.x only as 32bit Delphi added the property RecordNo to it's datasets.

Note that with dBase tables the result might be misleading in respect to the current position of the record as dBase has static record no's that don't reflect index orders.

## **BDEGetPDXUserCount routine**

### **Unit**

[BDEDoRxS](#)

### **Declaration**

```
function BDEGetPDXUserCount: integer;
```

### **Description**

Returns the number of users currently sharing the same net file.

