TTUtility Component

Properties Methods Events Tasks Support Etc.

Unit
TU

Description

TTUtility is a Delphi component that implements the functionality in Borland's TUtility.DLL, the same DLL that comes with Paradox for Windows. The primary purpose of this component is to give the Delphi or Paradox developer an easy to implement tool for validating and fixing corrupt Paradox tables from inside delivered applications. The TUtility DLL on which this component is based will work on Paradox tables up to and including level 5 tables.

Select a table to verify by assigning a value to the TableName property. Assigning a value to TableName has two side effects. TheTblInfo property is given value and the header of the table is verified. The TblInfo structure contains all kinds of valuable information about the table. Check TblInfo.bValidInfo and TblInfo.iRecords to get hints about the table header integrity.

You can also assign values to the tErrTableName, tBkUpTableName, tKeyVTableName, and tProbTableName properties. These are all tables that Verify and Rebuild generate as a side effect of their execution.

Next you would assign a value to the AltStructName property. This is the name of a known good table that rebuild can borrow the structure from. After assigning the AltStructName It is also a good idea to check the AltTblInfo.bValidInfo to make sure the alternate table does not have a corrupted header.

Assign a value to the Password property if the table has a master password.

Either drop the Verify Status Dialog and Rebuild Status Dialog Objects into your form or define the onInfoVerify and onInfoRebuild events to respond to the status messages from the verify and rebuild processes.

Finally, execute the ExecuteVerify public method. If errors are found, ExecuteVerify will create an error table with the name specified in the tErrTableName property. Check the iErrorLevel public property to the highest error level found in the error table. Use iErrorLevel, and the information in the TblInfo property to determine whether to use the Tables own structure or to borrow the structure from a similar table. To use the tables own structure run ExecuteRebuild passing it the value of the pCurrentTblDesc Property . To rebuild the table by borrowing a structure from a different table pass ExecuteRebuild the value in the pAltTblDesc Property .

You can even execute both the verify and rebuild in one step using the ExecuteVerifyRebuild Method .

See Also TUtility API , Strategies for corrupt file recovery and TIdxUtl

Properties

Run-time only

Key properties

iErrorLevel

Options

TblInfo	Pack
AltTblInfo	
Password	
pCurrentTblDesc	
TableName	
pAltTblDesc	Tag
AtStructAlways	tBkUpTableName
AltStructName	
tErrTableName	
AlwaysRebuild	tKeyVTableName
CBActive	tProbTableName
CBRebuildDialog	Table
CBVerifyDialog	
Name	

Methods

- ExecuteVerify
- ExecuteRebuild
- ExecuteVerifyRebuild

Events

▶ Run-time only

🔑 Key properties

🔑 OnInfoRebuild

🔑 OnInfoVerify

🔑 OnInfoVerReb

Using TTUtility Component

Here are 3 rules and one consideration you should be aware of when designing an application that incorporates the TTUtility component.

RULES

1. Any table that will be verified or rebuilt using the TUtility component must be set to inactive (Active = False) at design time if you want to run the application under Delphi. For a discussion on the reasons see the TUtility API section later in this doc. If the table is active you will receive a run time error.

2. We highly recommend that you never run ExecuteRebuild on a table without first running ExecuteVerify. ExecuteVerify discovers things about the table that ExecuteRebuild needs to know for a safe rebuild of the corrupt table.

3. If the table under consideration has a master password then it must be assigned correctly to the Password property. The TUtility component has no way of knowing if this password has been assigned incorrectly. In fact it will rebuild your table without the correct password, however, the resulting table will have no records in it. THIS DOES NOT RAISE AN ERROR. So... Make sure to assign the password correctly.

CONSIDERATION

Your application should deal with the side effect tables created by the Verify and Rebuild processes. These include the Error Table created by ExecuteVerify and the Problems and Key Violation Tables created by ExecuteRebuild. At some point the tables should be deleted (especially the Error Table). The first demo project automatically deletes the Error Table when it's done with it.

See Also Strategies for corrupt file recovery

iErrorLevel Property

Example

Applies To - TTUtility Component   Readonly and run time only.

Declaration

property iErrorLevel : Word;

Description
The iErrorLevel property contains the status of the table being worked on. This property gets set whenever the TableName or AltStructName properties are assigned. This property is also set when the ExecuteVerify procedure is run. Use the value in iErrorLevel to make decisions on how best to proceed through the verify/rebuild process. This property along with the bValidInfo field in the TblInfo (or AltTblInfo) structures are keys to successful table maintenance.

Then possible values are;

0 : No structure problems. Everythings OK.
1 : Table is damaged but verification can continue.
2 : Table is damaged and verification stops.
3 : Table must be rebuilt manually with a user supplied table description.
4 : Table cannot be rebuilt. Use your last backup.

Note that the Table Repair utility in Paradox for Windows will allow for an auto-rebuild (structure is not specified by
the user) on Level 2 errors. Experience shows that this is often a bad idea. The problem is that since the verify aborts
with a level 2 before completion, there is no way to tell if there is a level 3 or 4 error beyond the point where verify
aborts. We suggest that the user always specifies an alternate file structure on level 2 errors, especially if the bValidInfo
field is false.   (See the descriptions for TblInfo and AltTblInfo.).

TblInfo Property

Applies To - TTUtility Component   Readonly and run time only.

Declaration

property TblInfo: TTableInfo; (ReadOnly)

Description

The TblInfo record contains useful information on the structure of the table being verified and/or rebuilt. TblInfo is filled with data whenever the TableName property is assigned. Here is a brief description of the fields that make up the TTableInfo Structure.

sTableType	: String[32];	Driver type - Should always be "Paradox"
iFields	: Word;	Number of fields in Table
iRecSize	: Word;	Record size in bytes
iKeySize	: Word;	Key size (Primary key)
iIndexes	: Word;	Number of indexes on the table
iValChecks	: Word;	Number of val checks on the table
iRefIntChecks	: Word;	Number of Ref Integrity constraints on the table
iRestrVersion	: Word;	Restructure version number
iPasswords	: Word;	Number of Aux passwords on the table
bProtected	: Bool;	True if the table is protected by a password
sLangDriver	: String[32];	Language driver name
iBlockSize	: Word;	Physical file blocksize in K
iRecords	: Longint;	Number of records in table
bValidInfo	: Bool;	Is the header information reliable.

The last to fields in this data structure need special mention. The bValidInfo field specifies whether TTUtility was able to read the header information reliably. If this value is False then the chances are good that the header is corrupt. The safest thing to do in this case is to borrow the table description from another good table rather than use the corrupt tables description

AltTblInfo Property

Applies To - TTUtility Component   Readonly and run time only.

Declaration

property AltTblInfo: TTableInfo;

Description
The AltTblInfo record contains useful information on the structure of the table being used to borrow a table description from. AltTblInfo is filled with data whenever the AltStructName property is assigned. The value of AltTblInfo.bValidInfo should be checked after assigning AltStructName. If the value is False do not use the AltStructName table to specify the structure used for the rebuild. Here is a brief description of the fields that make up the TTableInfo Structure.

sTableType	: String[32];	Driver type - Should always be "Paradox"
iFields	: Word;	Number of fields in Table
iRecSize	: Word;	Record size in bytes
iKeySize	: Word;	Key size (Primary key)
iIndexes	: Word;	Number of indexes on the table
iValChecks	: Word;	Number of val checks on the table
iRefIntChecks	: Word;	Number of Ref Integrity constraints on the table
iRestrVersion	: Word;	Restructure version number
iPasswords	: Word;	Number of Aux passwords on the table
bProtected	: Bool;	True if the table is protected by a password
sLangDriver	: String[32];	Language driver name
iBlockSize	: Word;	Physical file blocksize in K
iRecords	: Longint;	Number of records in table
bValidInfo	: Bool;	Is the header information reliable.

The last to fields in this data structure need special mention. The bValidInfo field specifies whether TTUtility was able to read the header information reliably. If this value is False then the chances are good that the header is corrupt and this table should probably not be used as a source for a table structure.

pCurrentTblDesc Property

Example

Applies To - TTUtility Component   Readonly and run time only.

Declaration

property pCurrentTblDesc: pCRTblDesc

Description
Pass this value to ExecuteRebuild if you want to use the table specified in TableName to determine the structure of the rebuilt table. You can assign this value with the pointer to a CRTblDesc that you created yourself (Not Recommended). See the discussions in ExecuteRebuild for more information.

ALSO NOTE - The user should not destroy or modify the Table Desc structure that is passed back by pCurrentTblDesc it is to be looked at and passed to Rebuild table only.

ADVANCED NOTE - If you create and populate a pCRTblDesc structure yourself (not recommended) and assign it to TUtility's pCurrentTblDesc property it then becomes owned by the TUtility component. This means that you must not destroy it yourself. Any existing pCurrentTblDesc structure is destroyed (the memory is freed) when ever a new value is assigned or when the component itself is destroyed. If you destroy it yourself you run a good chance of GPFing your app.

ADVANCED   The following describes the pCRTblDesc (Create Table Description) for rebuilding a table. This structure is also documented in the Borland Database Engine User's Guide. The CRTblDesc structure defines the general attributes of the table and supplies pointers to arrays of field, index, and other descriptors.

Field	**Type**	**Description**
szTblName	DBITBLNAME	Table name, including path.
szTblType	DBINAME	Driver type.
szErrTblName	DBIPATH	Name of the Error table created by Execute Verify (including path)
szUserName	DBINAME	Not currently used.
szPassword	DBINAME	Master password (if bProtected is TRUE).
bProtected	BOOL	TRUE if table is encrypted.
bPack	BOOL	If TRUE, specifies packing for the rebuild. Assigned by the Pack property.
iFldCount	UINT16	The number of field descriptors supplied.
pecrFldOp	pCROpType	Not used by rebuild. Must be zero.
pfldDesc	pFLDDesc	An array of field descriptors.
iIdxCount	UINT16	The number of index descriptors supplied.
pecrIdxOp	pCROpType	Not used by rebuild. Must be zero.
pidxDesc	pIDXDesc	An array of index descriptors.
iSecRecCount	UINT16	The number of security descriptors given.
pecrSecOp	pCROpType	Not used by rebuild. Must be zero.
psecDesc	pSECDesc	An array of security descriptors
iValChkCount	UINT16	The number of validity checks
pecrValChkOp	pCROpType	Not used by rebuild. Must be zero.
pvchkDesc	pVCHKDesc	An array of validity check descriptors.
iRintCount	UINT16	The number of referential integrity specifications.
pecrRintOp	pCROpType	Not used by rebuild. Must be zero.
printDesc	pRINTDesc	An array of referential integrity specifications.
iOptParams	UINT16	The number of optional parameters.
pfldOptParams	pFLDDesc	An array of field descriptors for optional parameters.
pOptData	pBYTE	The values of optional parameters.

In order to populate this structure correctly you must also create pointers to the pfldDesc, pIDXDesc, pSECDesc, pVCHKDesc and pRINTDesc. For information on these record structures refer to the DbiTypes.Int file in you Delphi\Doc directory or to the BDE User's guide if you have it.

Also, be advised, that the last three fields of the structure mentioned above (iOptParams, pfldOptParams, pOptData) must contain information specific to the Paradox table that is to be rebuilt. The required information is not documented anywhere outside of Borland. We suggest that if you really must create and populate this structure yourself that you first populate it a few times by borrowing the structure from a known table and then study the data in the borrowed structure.

The authors of this component can not support technical questions relating the to manual populating of this structure.

pAltTblDesc Property

Example

Applies To - TTUtility Component   Readonly and run time only.

Declaration

property pAltTblDesc: pCRTblDesc

Description

Pass this value to ExecuteRebuild if you want to use the table specified in AltStructName to determine the structure of the rebuilt table.   The description returned in the pAltTblDesc represents the complete description of the table named in the AltStructName property.

This value is readonly. It exists only as a convenient way to specify a table to borrow a structure from for the rebuild process.

NOTE - The user should not destroy or modify the Table Desc structure that is passed back by pAltTblDesc it is to be looked at and passed to Rebuild table only. See pCurrentTblDesc for a description of the pCRTblDesc structure.

AtStructAlways Property

Applies To - TTUtility Component   Read write, and both design and   time only.

Declaration

property AltStructAlways: Boolean

Description

Set this value to true if you always want the table specified in AltStructName to determine the structure of therebuilt table. This is used only by ExecuteVerifyRebuild.

AltStructName Property

Applies To - TTUtility Component   Read write, and both design and   time only.

Declaration

property AltStructName: TFileName

Description

Assign this property the name of the table to use as the structure for the rebuild. This should be a completelyqualified file name including the path. Assigning a value to AltStructName has two side effects. The AltTblInfo property is given value and the header of the AltStructName table is verified. The values of in iErrorLevel and AltTblInfo.bValidInfo should be checked after assigning a value to AltStructName.

AlwaysRebuild Property

Applies To - TTUtility Component   Read write, and both design and   time only.

Declaration

property AlwaysRebuild: Boolean

Description

Used by ExecuteVerifyRebuild. If this property is true then the table mentioned in the TableName property is always rebuilt even when verify shows that it has no errors.

CBActive Property

Applies To - TTUtility Component   Read write, and both design and   time only.

Declaration

property CBActive : Boolean

Description

Set this value to False if you don't want the installed Callback functions activated. You make completely define the callbacks using CBVerifyDialog and CBRebuildDialog or OnInfoVerify and OnInfoRebuild and then choose to not use them at run time by setting this value to false. You would do this for performance reasons.

Testing has shown that turning the information callbacks on has an adverse effect on performance. The CBActive can be turned on or off at runtime. This is especially useful for creating programs that do multiple verifies and rebuilds and where there will be no one around to watch the gauges move anyway.

NOTE : Do not try to change CBActive from True to False from inside any of the onInfoXXXX events. CBActive must be set prior to ExecuteVerify,   ExecuteRebuild or ExecuteVerifyRebuild.

CBRebuildDialog Property

Applies To - TTUtility Component   Read write, and both design and   time only.

Declaration

property CBRebuildDialog: TRebuildDlg

Description

Assign this the value of the TRebuildDlg component on your form. This is the easiest way to get status information during ExecuteRebuild. If you do not want to use the canned TRebuildDlg component you may define your own status dialog by implementing the OnInfoRebuild Event

CBVerifyDialog Property

Applies To - TTUtility Component   Read write, and both design and   time only.

Declaration

property CBVerifyDialog: TVerifyDlg

Description

Assign this the value of the TVerifyDlg component on your form. This is the easiest way to get status information during ExecuteVerify.

If you do not want to use the canned TVerifyDlg component you may define your own status dialog by implementing the OnInfoVerify Event .

Options Property

Applies To - TTUtility Component   Read write, and both design and   time only.

Declaration

property Options: TVerifyOptions

Description

This property specifies various behaviors of ExecuteVerify. The default is all options are false. Here are the available options:

TU_Append_Errors	Append errors to an existing error table
TU_No_Secondary	Bypass secondary indexes
TU_No_Warnings	Prevent warnings of secondary errors
TU_Header_Only	Verify table header only
TU_Dialog_Hide	Reserved for future expansion. Do not use
TU_No_Lock	Do not lock table being verified

If you are going to create an application that verifies a number of files in a batch, then you will want to set the TU_Append_Errors to TRUE unless you specify a different error table for each table in your batch.

Pack Property

Applies To - TTUtility Component   Read write, and both design and   time only.

Declaration

property Pack :   Boolean

Description

The way this property is supposed to work is that if the Pack property equals TRUE then ExecuteRebuild packs the rebuilt table. If Pack is set to FALSE the records are not packed. This is part of the TUtility API and is exposed in the component for completeness. However, it is not supported by the current version of the TUtility.DLL. The records are always packed.

Password Property

Applies To - TTUtility Component   Read write, and both design and   time only.

Declaration

property Password :   String

Description

Assign the master password of the table to this property. If the table is password protected then this property MUST be correctly assigned, otherwise, when ExecuteRebuild is run, the table will appear to be rebuilt but the resulting table will have no records in it. See the discussion later in this document under the TUtility API section for more information about passwords.

Passwords and the TUtility API are rather tricky. Since the TUtility does not use a standard BDE session it has no knowledge of what passwords are available. For this reason you must specify the table's master password in order for the ExecuteRebuild procedure to work correctly.

Warning, and very important,   there is no way to check if the password entered is valid. If a table has a password assigned to it, ExecuteRebuild will run with no errors even if the wrong password (or no password) is entered. When this happens the resulting rebuild table will have no records. Note that this is even true in the Rebuild performed by Paradox for Windows. PFW will ask for a master password but if you key in an incorrect one there is no error message. The table is rebuild and the record count is zero.

The TUtility component offers some ways to insure that the correct password is entered. Heres the strategy. When the TableName property is assigned, as a side effect,   it checks to see if a password is required for the table. If a password is required it checks TUtility's password property to see if it has been assigned. If no password as been assigned a message box asks the user to assign the password. Now with the password in hand, TUtility attempts to get an extended description of the table without opening a cursor on the table. If this is successful then TUtility's TblInfo record property will show a positive value in the iRecords field. It's a good bet that if iRecords is zero than the password is incorrect but it's still no guarantee since a corrupt table header can also return an iRecords count of zero even when the password is correct.

Of course you could always try to open the table but if the table is corrupt then this may not be possible. Remember that the TUtility component never opens a cursor on the table being worked on.

So the bottom line is, make sure the value assigned to the password property is correct.

Table Property

Applies To - TTUtility Component   Read write, and both design and   time only.

Declaration

property Table :   TTable

Description

This property was added as an optional way of specifying the TableName . You can drop a TTable component into your form and then use this property to select that TTable component. Assigning this component automatically assigns a value to the TableName property. This may be easier than assigning the TableName property directly if the path name is very long.

Note that this property is completely optional and is ment only as a way of assisting in the assigning of the TableName property. Also not that using this method to assign Tablename uses slightly more resources than assigning TableName directly,

TableName Property

Applies To - TTUtility Component   Read write, and both design and   time only.

Declaration

property TableName : TFileName

Description

Assign this property the name of the table to be verified and/or rebuilt. This should be a completely qualified file name, including the path. Assigning a value to TableName has two side effects.   The TblInfo property is given a value and the header of the table is verified. The values in iErrorLevel and TblInfo.bValidInfo should be checked after assigning a value to TableName.

Assigning a value to the Table property will automatically assign a value to the TableName property. Also assigning a value to TableName directly will clear the Table property.

tBkUpTableName Property

Applies To - TTUtility Component Read write, and both design and   time only.

Declaration

property tBkUpTableName: TFileName

Description

Assign this property the name of the backup table created by ExecuteRebuild. The default is TableName + '_'. For example, if the TableName was CUSTOMER.DB then the backup name would be CUSTOME_.DB If no path name is specified then the table will be created in the same directory as TableName .

tErrTableName Property

Applies To - TTUtility Component   Read write, and both design and   time only.

Declaration

property tErrTableName : TFileName

Description

Assign this property the name of the Error Table to be created by ExecuteVerify. The default is __TUERR.DB. If no path name is specified, the table will be created in whatever directory Delphi believes is your Private Directory (Session.PrivDirectory).

tKeyVTableName Property

Applies To - TTUtility Component   Read write, and both design and   time only.

Declaration

property tKeyVTableName: TFileName

Description

Assign this property the name of the key violation table created by ExecuteRebuild. The default is KEYVIOL.DB. If no path name is specified, the table will be created in whatever directory Delphi believes is your Private Directory (Session.PrivDirectory).

tProbTableName Property

Applies To - TTUtility Component   Read write, and both design and   time only.

Declaration

property tProbTableName: TFileName

Description

Assign this property the name of the problems table created by ExecuteRebuild. The default is PROBLEMS.DB. If no path name is specified, the table will be created in whatever directory Delphi believes is your Private Directory (Session.PrivDirectory).

ExecuteVerify Method

Example

Applies To - TTUtility Component

Declaration

procedure ExecuteVerify

Description

ExecuteVerify performs the verify step. At a minimum the TableName property must be set for this procedure to operate. Depending on how options are set, ExecuteVerify creates or appends to the table specified by the tErrTableName property. You can view this table for an in-depth analysis of the table's problems. On completion ExecuteVerify sets iErrorLevel to the highest error encountered. Use iErrorLevel and the value in TblInfo.bValidInfo to determine the best way to run ExecuteRebuild.

The structure of the error table created by ExecuteVerify looks like this.

Field Name	Type	Size	Description
Drive	A	2	Disk Drive
Directory	A	65	Path to the Table
Table Name	A	8	Paradox Table Name
Extension	A	4	Should always be .DB
Error Code	S		Code used to get the Error Message (Appendix C)
Error Level	S		Rating of error severity
Error Message	A	150	Textual description of error
Date	D		The tables file date
Time	T		The tables file time

The Error Level is the most important field since it rates the importance of the error. See the description of the iErrorLevel property. ExecuteVerify reports the highest value found in the Error Level field in the iErrorLevel property.

ExecuteRebuild Method

Example

Applies To - TTUtility Component

Declaration

Procedure ExecuteRebuild(pTableDesc : pCRTblDesc)

Description

ExecuteRebuild attempts to fix the table. It creates a backup of the original table in the table specified by the tBkUpTableName property. A problem table,tProbTableName, and key violation table tKeyVTableName may also be created.

ExecuteRebuild's minimum requirement is that the TableName property be specified and that a value be passed in the method's pTableDesc parameter.

The pTableDesc parameter specifies a complete description of the table to be rebuilt. This table description can be created in any of three ways;

1. Use the value in the pCurrentTblDesc property which is the description of the table named in the TableName property.

2. Use the value in the pAltTblDesc property which is the description of the table named in the AltStructName property. (This is borrowing the structure from another table).

3. **(NOT RECOMMENDED)** The user can create the table description itself.. If this is what you want to do you must study and completely understand the CRTTblDesc. This is by far the most complicated record structure we have ever encountered and we highly recommend that you avoid attempting the creation of this structure from scratch.

Note, that once you pass this pointer to ExecuteRebuild it becomes "owned" by the TTUtility object. Do not attempt to destroy it yourself, TTUtility will take care of the destruction of this structure.

Recommendation - For the safest, most successful rebuilds we recommend that you pass pCurrentTblDesc as the parameter to ExecuteRebuild only when the iErrorLevel is 2 or less (see iErrorLevel above) and when TblInfo.bValidInfo is True. Otherwise, specify a table to borrow the structure from in AltStructName and use pAltTblDesc as the parameter for ExecuteRebuild.

ExecuteVerifyRebuild Method

Example

Applies To - TTUtility Component

Declaration

procedure ExecuteVerifyRebuild

Description

ExecuteVerifyRebuild combines the verify and rebuild processes into a single convenient procedure call. First, the table mentioned in the TableName property is verified. If the header is not damaged (TblInfo.bValidInfo = True) and the error level is less than 3 then the table's own structure is used for the rebuild, otherwise the table named in the AltStructName property is used. If AltStructAlways is true than the table named in the AltStructName property is always used to get the rebuild structure. The AltStructName property should always be assigned prior to executing this procedure.

If the AlwaysRebuild property is set to true then the table will always be rebuilt even it verify returns an iErrorLevel of zero. If AlwaysRebuild is false (the default) then the table is not rebuilt if the verify showed no table errors. **Note, it has been reported that the the verify portion of the TUtility.Dll can show no when there are errors that rebuild can fix.** We can not prove or disprove this claim. Also Verify will not report out of date secondary indexes. Use the TIdxUtl component to check and regenerate secondary indexes.

Use the companion OnInfoVerReb event plus onInfoVerify and onInfoRebuild to monitor and record the execution of this procedure.

OnInfoRebuild Event

Example

Applies To - TTUtility Component

Declaration

event OnInfoRebuild: TInfoRebuildEvent

Description

Define the OnInfoRebuild event if you want to create your own Rebuild Status Dialog box. The TInfoRebuildEvent looks like this.

```
     TInfoRebuildEvent = procedure(
           Sender: Tobject;        {Where the message came from}
           RebuildCBRec: TRebuildCBData)       { Message to display }
           of object;
```

Where TRebuildCBData is

```
  TRebuildCBData = record
    iPercentDone     : Integer;           { Percentage done. }
    sMsg             : String[128];       { Message to display }
  end;
```

If sMsg is blank then use the information in iPercentDone other us the information in sMsg.

NOTE : This is VERY important. DO NOT MAKE ANY DATABASE CALLS FROM THIS METHOD. This event is actually part of a BDE Callback response. The rules for Callback responses are clear. The BDE is not re-entrant, that means that you can not do anything here that would call the BDE. So.... No database calls. Just make pictures.

OnInfoVerify Event

Example

Applies To - TTUtility Component

Declaration

event OnInfoVerify: TInfoVerifyEvent

Description

Define the OnInfoVerify event if you want to create your own Verify Status Dialog box. The TInfoVerifyEvent looks like this.

```
  TInfoVerifyEvent = procedure(
     Sender: Tobject;        {Where the message came from}
     VerifyCBRec: TVerifyCBData    {The data to be acted on}
     ) of object;
```

Where TVerifyCBData is

```
  TVerifyCBData = record
    PercentDone: word;             The Percent Completed
    TableName: String[82];         Passed only with Process = 
TUVerifyTableName
    Process: TUVerifyProcess;      Changes with the various verify steps 
below.
    CurrentIndex: word;            Increments with each secondary index 
checked.
    TotalIndex: word;              Number of Secondary Indexes
  end;
```

and The TUVerifyProcess is

```
  TUVerifyProcess = (
     TUVerifyHeader,         Header is verified, PercentDone increments.
     TUVerifyIndex,          Primary Index is verified, PercentDone 
increments.
     TUVerifyData,           Primary Index Data is verified, PercentDone 
increments.
     TUVerifySXHeader,       A Secondary Index Header verified, PercentDone 
incs.
     TUVerifySXIndex,        A Secondary Index verified, PercentDone 
increments.
     TUVerifySXData,         A Secondary Index data verified, PercentDone 
incs.
     TUVerifySXIntegrity,    A Secondary Index integrity is verified. Ditto.
     TUVerifyTableName       Passes the Table Name in 
TVerifyCBData.TableName
     );
```

You need to watch the process field to determine which gauge to adjust with the amount delivered in PercentDone.

NOTE : This is VERY important. DO NOT MAKE ANY DATABASE CALLS FROM THIS METHOD. This event is actually part of a BDE Callback response. The rules for Callback responses are clear. The BDE is not re-entrant, that means that you can not do anything here that would call the BDE. So.... No database calls. Just make pictures.

OnInfoVerReb Event

Example

Applies To - TTUtility Component

Declaration

event OnInfoVerReb: TInfoVerRebEvent;

Description

OninfoVerReb sends textual messages back to your application as ExecuteVerifyRebuild runs. These messages are in addition to those sent by onInfoVerify and onInfoRebuild. The text messages sent by ExecuteVerifyRebuild and received in onInfoVerReb are general in nature and allow you to monitor the VerifiyRebuild sequence. The TInfoVerRebEvent looks like this:

```
TInfoVerRebEvent = procedure(
    Sender: TObject;
    AMessage : String;
    Process : TUVerRebProcess;
    var Abort : Boolean) of object;
```

Where the Process field is

```
TUVerRebProcess = (TUVerifying, TURebuilding);
```

As ExecuteVerifyRebuild runs it sends general status information back to the application by firing onInfoVerReb. The status information arrives in the AMessage field. This information can be written to the screen or printer for later review.

The Process field indicates which process is running (verify or rebuild). You can abort onInfoVerReb by setting Abort to true.

Example

Use iErrorlevel, pCurrentTblDesc and pAltTblDesc to properly rebuild a table.

```
procedure TFormTUMain.ButtonRebuildClick(Sender: TObject);
begin
  TUtilityVerReb.AltStructName:= C:\Data\GoodInfo.DB;
  TUtility1.tBkUpTableName := C:\Data\BadInfo_.DB
  TUtility1.tKeyVTableName := C:\Data\KyVInfo.DB
  TUtility1.tProbTableName:= C:\Data\ProbInfo.DB

  If (TUtility1.iErrorLevel < 3) then
    Tutility1.ExecuteRebuild(TUtility1.pCurrentTblDesc);
  else if (TUtility1.iErrorLevel < 4) then
    Tutility1.ExecuteRebuild(TUtility1.pAltTblDesc);
  else
  begin
    MessageDlg('BAD NEWS! The cannot be rebuilt.' + #10#13 +
               'Reload from backups.',  mtInformation, [mbOK], 0);
    exit;               {Can't rebuild so Bail out }
  end;
  MessageDlg('Table Successfully rebuild!', mtInformation, [mbOK], 0);
end;
```

Example

```
procedure TFormTUMain.ButtonVerifyClick(Sender: TObject);
begin
  TUtility1.TableName := C:\Data\BadInfo.DB;
  TUtility1.tErrTableName := C:\Data\ErrInfo.DB
  TUtility1.ExecuteVerify;
  if TUtility1.ierrorLevel <> 0 then
  begin
    MessageDlg('The table is corrupt and must be repaired!,
                 mtWarning, [mbYes, mbNo], 0) = mrYes then
  end
  else
  begin
    MessageDlg('GOOD NEWS!' + #10#13 + 'Header and Data are O.K.',
                 mtInformation, [mbOK], 0);
  end;
end;
```

Example

```
procedure TFormBatchMain.ButtonFixAllClick(Sender: TObject);
begin
  TUtilityVerReb.TableName := C:\Data\BadInfo.DB;
  TUtilityVerReb.AltStructName:= C:\Data\GoodInfo.DB;
  TUtilityVerReb.tBkUpTableName := C:\Data\BadInfo_.DB
  TUtilityVerReb.tErrTableName := C:\Data\ErrInfo.DB
  TUtilityVerReb.tKeyVTableName := C:\Data\KyVInfo.DB
  TUtilityVerReb.tProbTableName:= C:\Data\ProbInfo.DB
  Try
    TUtilityVerReb.ExecuteVerifyRebuild;
  except
   {report the error to the log  so it doesn't stop the process}
   on E:Exception do
       SendToLog(E.Message);
  end;
end;
```

Example

```
procedure TFormTUMain.TUtility1InfoRebuild(Sender: TObject;
  RebuildCBRec: TRebuildCBData);
begin
  with RebuildCBRec do
  begin
    if sMsg = '' then
      FormRebuildStatus.GaugeRebuild.Progress := iPercentDone
    else
    begin
      FormRebuildStatus.LabelNumPacked.Caption := sMsg;
      FormRebuildStatus.refresh;
    end;
  end;
end;
```

Example

```
procedure TFormTUMain.TUtility1InfoVerify(Sender: TObject;
  VerifyCBRec: TVerifyCBData);
begin
  with VerifyCBRec do
  begin
    Case Process of
      TUVerifyTableName : FormVerifyStatus.LabelStatus.Caption := TableName;
      TUVerifyHeader : FormVerifyStatus.GaugeHeader.Progress := PercentDone;
      TUVerifyIndex : FormVerifyStatus.GaugeIndex.Progress := PercentDone;
      TUVerifyData : FormVerifyStatus.GaugeData.Progress := PercentDone;
      TUVerifySXHeader : FormVerifyStatus.GaugeHeaderIdx.Progress :=
PercentDone;
      TUVerifySXIndex : FormVerifyStatus.GaugeIndexIdx.Progress :=
PercentDone;
      TUVerifySXData : FormVerifyStatus.GaugeDataIdx.Progress :=
PercentDone;
      TUVerifySXIntegrity :
        begin
          FormVerifyStatus.GaugeIntegrity.Progress := PercentDone;
          FormVerifyStatus.LabelZeroOf.Caption := IntToStr(CurrentIndex);
          FormVerifyStatus.LabelOfZero.Caption := IntToStr(TotalIndex);
          FormVerifyStatus.refresh;
        end;
    end; {Case}
  end;
end;
```

Example

```
procedure TFormBatchMain.TUtilityRestInfoVerReb(Sender: TObject;
  AMessage: String; Process: TUVerRebProcess; var Abort: Boolean);
begin
  SendToLog(AMessage);
  if process <> CurProcess then
  begin
    Case Process of
    TUVerifying  :
      begin
        FormStatus.GroupBoxVerify.Font.Color := clRed;
        FormStatus.GroupBoxRebuild.Font.Color := clBlack;
      end;
    TURebuilding :
      begin
        FormStatus.GroupBoxVerify.Font.Color := clBlack;
        FormStatus.GroupBoxRebuild.Font.Color := clRed;
      end;
    end; {case}
    FormStatus.GroupBoxVerify.refresh;
    FormStatus.GroupBoxRebuild.refresh;
    CurProcess := Process;
  end;
end;
```

TU Unit

The TU unit contains the declarations for the TTUtility component, as well as, the declarations for the associated fields. When you add a component declared in this unit to a form, the unit is automatically added to the uses clause of that form's unit. The following items are declared in the TU unit:

Components
TTUtility

Types

ETUtilityError
TVerifyOption
TVerifyCBData
TInfoVerifyEvent
TRebuildCBData
TInfoRebuildEvent
TInfoVerRebEvent
TTableInfo

To see a listing of items declared in this unit including their declarations, use the ObjectBrower.

ETUtilityError Type

Unit

TU

Declaration

ETUtilityError = class(Exception)

Description

Specialized TUtility error class that knows about the errors and error messages supported in the TUtility API.

TVerifyOption Type

Unit

TU

Declaration

```
   TVerifyOption = (
      vTU_Append_Errors,
      vTU_No_Secondary,
      vTU_No_Warnings,
      vTU_Header_Only,
      vTU_Dialog_Hide,
      vTU_No_Lock);

also
   TVerifyOptions = Set of TVerifyOption;
```

Description

Type used bt the Tutility.Options property.

TU_Append_Errors	Append errors to an existing error table
TU_No_Secondary	Bypass secondary indexes
TU_No_Warnings	Prevents warnings of secondary errors
TU_Header_Only	Verify table header only
TU_Dialog_Hide	Reserved for future expansion. Do not use
TU_No_Lock	Do not lock table being verified

TVerifyCBData Type

Unit

TU

Declaration

```
TVerifyCBData = record
  PercentDone: word;
  TableName: String[82];
  Process: TUVerifyProcess;
  CurrentIndex: word;
  TotalIndex: word;
end;
```

where

```
TUVerifyProcess = (TUVerifyHeader, TUVerifyIndex, TUVerifyData, TUVerifySXHeader,
                   TUVerifySXIndex, TUVerifySXData, TUVerifySXIntegrity,
                   TUVerifyTableName);
```

Description

The TVerifyCBData record is one of the paramenters of the TInfoVerifyEvent which is the type of the TTUtility.OnInfoVerify event.

PercentDone	The Percent Completed
TableName	Passed only with Process = TUVerifyTableName
Process	Changes with the various verify steps below.
CurrentIndex	Increments with each secondary index checked.
TotalIndex	Number of Secondary Indexes

The Process field of TVerifyCBData is of type TUVerifyProcess its members are described as

TUVerifyHeader	Header is verified, PercentDone increments.
TUVerifyIndex	Primary Index is verified, PercentDone increments.
TUVerifyData	Primary Index Data is verified, PercentDone increments.
TUVerifySXHeader	A Secondary Index Header verified, PercentDone incs.
TUVerifySXIndex	A Secondary Index verified, PercentDone increments.
TUVerifySXData	A Secondary Index data verified, PercentDone incs.
TUVerifySXIntegrity	A Secondary Index integrity is verified. Ditto.
TUVerifyTableName	Passes the Table Name in TVerifyCBData.TableName

TInfoVerifyEvent Type

Unit

TU

Declaration

```
TInfoVerifyEvent = procedure(
        Sender: TObject;
        VerifyCBRec: TVerifyCBData) of object;
```

Description

The OnInfoVerify event is defined as type TInfoVerifyEvent. The Sender field specifices the object that fired the event. VerifyCBRec is of type TVerifyCBData and contains all the information needed to created RAD status displays for the verify process.

TRebuildCBData Type

Unit

TU

Declaration

```
TRebuildCBData = record
  iPercentDone      : Integer;
  sMsg              : String[128];
end;
```

Description

TRebuildCBData defines the type of the information record passed as one of the parameters in in OnInfoRebuild event which is of type TInfoRebuildEvent.

PercentDone contains the percentage complete of the data move part of the rebuild process while sMsg contains a verbal description of the pack part of the rebuild process.

TInfoRebuildEvent Type

Unit

TU

Declaration

```
TInfoRebuildEvent = procedure(
      Sender: TObject;
      RebuildCBRec: TRebuildCBData) of object;
```

Description

The TInfoRebuildEvent type is used to define the OnInfoRebuild event. RebuildCBRec contains the information needed to display RAD status dialogs.

TInfoVerRebEvent Type

Unit

TU

Declaration

```
TInfoVerRebEvent = procedure(
    Sender: TObject;
    AMessage : String;
    Process : TUVerRebProcess;
    var Abort : Boolean) of object;
```

where Process is defined as

```
TUVerRebProcess = (TUVerifying, TURebuilding);
```

Description

TInfoVerRebEvent is the type of the ExecuteVerifyRebuild event.

As ExecuteVerifyRebuild runs it sends general status information back to the application by firing onInfoVerReb. The status information arrives in the AMessage field. This information can be written to the screen or printer for later review.

The Process field indicates which process is running (verify or rebuild). You can abort onInfoVerReb by setting Abort to true.

TTableInfo Type

Unit

TU

Declaration

```
TTableInfo = Record
  sTableType                   : String[32];
  iFields                      : Word;
  iRecSize              : Word;
  iKeySize              : Word;
  iIndexes              : Word;
  iValChecks            : Word;
  iRefIntChecks         : Word;
  iRestrVersion         : Word;
  iPasswords            : Word;
  bProtected            : Bool;
  sLangDriver           : String[32];
  iBlockSize            : Word;
  iRecords              : Longint;
  bValidInfo            : Bool;
end;
```

Description

TTableInfo is the type of TTUtilities TblInfo and AltTblInfo properties.

sTableType	: String[32];	Driver type - Should always be "Paradox"
iFields	: Word;	Number of fields in Table
iRecSize	: Word;	Record size in bytes
iKeySize	: Word;	Key size (Primary key)
iIndexes	: Word;	Number of indexes on the table
iValChecks	: Word;	Number of val checks on the table
iRefIntChecks	: Word;	Number of Ref Integrity constraints on the table
iRestrVersion	: Word;	Restructure version number
iPasswords	: Word;	Number of Aux passwords on the table
bProtected	: Bool;	True if the table is protected by a password
sLangDriver	: String[32];	Language driver name
iBlockSize	: Word;	Physical file blocksize in K
iRecords	: Longint;	Number of records in table
bValidInfo	: Bool;	Was the header information reliable.

TUtility API

The TTUtility component implements the functionality of Borland's TUtility.DLL through the function made public by the published TUtility API. If you would like a copy of the API then download the TUTILITY.ZIP file found in the Borland Tools forum on CompuServe. The following are some observations about the Tutility API that we made during the development of this component.

BDE Sessions vs. TUtility Sessions

The TUtility API does not use the standard BDE session. In fact the API supports it's own specialized session. This session has no knowledge of the things that are associated with a typical BDE session, like aliases for example. This does not mean that you have no access to BDE session information while working with the TTUtility component. You can still access the global session variable as long as the DB unit is referenced in your uses clause. Just bear in mind that TTUtility itself knows nothing about session. You do not need to worry about the TUtility session since it has been completely encapsulated into the component.

The TUtility API also does all its table verification and rebuilding without ever opening a BDE cursor on the table. This is a requirement for rebuild since if the table is corrupted no cursor could be opened in any case.

Once again. The TUtility component never opens a cursor on the table being worked on. Any verify/rebuild functionality that you build into your application must insure that the table to be worked has its active property set to false and that this property remain false while you use the TUtilitys executeXXXX methods.

Passwords

Passwords and the TUtility API are rather tricky. Since the TUtility does not use a standard BDE session it has no knowledge of what passwords are available. For this reason you must specify the table's master password in order for the ExecuteRebuild procedure to work correctly.

WARNING & VERY IMPORTANT, there is no way to check if the password entered is valid. If a table has a password assigned to it, ExecuteRebuild will run with no errors even if the wrong password (or no password) is entered. When this happens the resulting rebuild table will have no records. Note that this is even true in the Rebuild performed by Paradox for Windows. PFW will ask for a master password but if you key in an incorrect one there is no error message. The table is rebuilt and the record count is zero.

The TUtility component offers some ways to insure that the correct password is entered. Heres the strategy. When the TableName property is assigned, as a side effect, it checks to see if a password is required for the table. If a password is required it checks TUtility's password property to see if it has been assigned. If no password has been assigned a message box asks the user to assign the password. Now with the password in hand, TUtility attempts to get an extended description of the table without opening a cursor on the table. If this is successful then TUtility's TableInfo record property will show a positive value in the iRecords field. It's a good bet that if iRecords is zero then the password is incorrect but it's still no guarantee since a corrupt table header can also return an iRecords count of zero even when the password is correct.

Of course you could always try to open the table but if the table is corrupt then this may not be possible. Remember that the TUtility component never opens a cursor on the table being worked on.

So the bottom line is, make sure the value assigned to the password property is correct.

Active Tables at Design Time

Another side effect of the TUtilities independent session is that it does not recognize a table being made inactive (Active := False) at run time. **Note, this limitation only applies when trying to run an application inside the Delphi environment.**

In fact, the TUtility component sees Delphi design time as being a separate application that has the table it wants to work on open. This is unfortunate because it means that you **must** insure that no table that could possibly be run against executeVerify or ExecuteRebuild be set to active at design time. For example, you cannot set a table to active at design time and then set it to inactive at run time prior to running executeVerify. The problem is that the TUtility API has no knowledge of the DBE session so it does not see the table go inactive at run time. It still believes the table is open and you will get a "Table Busy" error message. So.... you must insure that all tables in an application that use the TUtility API must have there active property set to FALSE at design time.

This requirement is only true when you run your application under Delphi. The problem does not exist when running the compiled executable application alone outside the Delphi environment. Also there is no problem making a table active at run time. In fact, as long as the table starts off as Active = FALSE, then active can be set to TRUE, have something done to it, set back to FALSE and have executeVerify run against it!

The Rule is only that it must start off inactive if you want to test the verify/rebuild functionality inside the Delphi environment.

Strategies for corrupt file recovery

Creating an Environment for the Painless recovery of Table Corruption

Over the years, TUtility, in its various forms, has probably created more forum and seminar discussion than probably any other single aspect of the Paradox database system. Certainly corruption and lost data tends to be a loud issue.

Data corruption **does** seem to just happen. Probably the biggest reason is desktops or servers being turned off or rebooted while some table activity is going on. Often there is no apparent reason and often the data corruption can go on for a long period of time before anyone notices. The one thing that you can be sure of is that data corruption **will** eventually happen.

Creating an environment that allows you to recover from data corruption quickly, easily and with minimum loss of data is really very straight forward. Just follow three rules:

Rule #1 - Backup your files regularly and keep a number of iterations of the backups.

While this rule is obvious it still needed to be said. Going to a non-corrupt backup is sometimes the only way you have of recovering from a badly damaged table. If ExecuteVerify reports a level 4 error in iErrorLevel then you are SOL unless you have a recent (and good) backup.

Rule #2 - For every table in your database keep an empty clone of it stored in a separate directory and on removable disk.

As mentioned earlier, the safest way to rebuild a table is to borrow the tables structure from another identical table which is known to be good. The official word is that the TUtility.DLL can rebuild a table using its own structure as long as the iErrorLevel is less than 3. While this is the official word there are a number of users out there who would disagree based on experience. So, whenever possible rebuild tables using a borrowed structure from a known good clone.

Rule #3 - Verify all your tables on a regular basis.

The Paradox world is littered with stories of corrupt tables that turned out to also be corrupt on the last backup, on the backup before that. In fact, we heard one story of a shop that went back to its year end backup (9 months old) and discovered that the table in question was corrupt even then. The solution is simple: verify all the tables in your database on a regular basis**.**

Deciding How to Rebuild a Table

The decision on how to rebuild a table can be a little complicated. By far the simplest (and safest) solution is to not make the decision at all and to always rebuild the table using an alternate or borrowed table structure.

Having said that, here are the decision making rules if you want to make a decision.

Check the iErrorLevel property after ExecuteVerify is run. If iErrorLevel is less than 3 then the table is a good candidate for a rebuild using its own structure. The possibly of a successful rebuild goes down if the TblInfo.bValidInfo value is FALSE and/or the TblInfo.iRecord shows a value of zero when you know the table has records in it. But even then you might be OK.

What to do when you have a Level 4 Error and no table to borrow the structure from If this happens then you are still probably OK it's just that you need to do a little more work. You need to create a table using either Paradox or the Database Desktop. This table must be exactly the same structure as the corrupt table. If you don't know the structure then you are out of luck. Once you have created this table you can use it as the alternate structure table and borrow its structure to rebuild the corrupt table.

Support Etc.

TTUtility was created by Out & About Productions. We provide support for this component by e-mail, FAX., and snail mail.

e-mail	: Compuserve	75664,1224
	: Internet	75664.1224@compuserve.com
FAX	: 619.259.0210	
snail mail	: Out & About Production	
	: 8526 Lepus Road	
	: San Diego, CA 92126	

Disclaimers and Legal Stuff.

We really hate disclaimers but in the case of this component we feel it is absolutely necessary. The TUtility API and this component on which it is based are very powerful and can make your file maintenance tasks much easier. However, along with this power comes great potential for disaster.

We have studied the TUtility API in depth and have learned about a number of its quirks. All that we have discovered is documented here. We realize that the documentation is lengthy, but we highly recommend that you read it from front to back. If you follow the guidelines documented here you should be able to easily and successfully build applications that incorporate verify and rebuild functionality. We have tested the component both in house and in beta test, but this

testing is still a far cry from understanding all the different things that can happen to a file in the real world. So... here's the disclaimer.

If you would like additional information on Borland's TUtility API you should download the file TUTILITY.ZIP from the Borland Tools forum on CompuServe.

TIdxUtl Component

Properties Methods Events Support Etc.

The TIdxUtl component was added to the TUtility component set to give the Delphi application developer a simple way to validate and update Paradox table indexes. With TIdxUtl you can check to see if the indexes on a Paradox table are up to date and then if they are out of date the component can regenerate the index(es). This component was added for completeness of the component set. The TUtility.Dll does not check for out of date indexes so this component was added to fill the gap.

To use TdxUtl to check and regen a tables index you need to assign a table to the TableName property and assign the Password property if the table has a master password. Then run CheckIndexes to determine if the indexes are up to date. If indexes are found to be out of date run RegenIndex to regenerate the indexes. You can use the onInfoIdxCheck and onInfoIdxRegen events to monitor the progress.

This component is especially useful in situations where file servers or work stations are rebooted while table access is in progress. Often this does not cause table corruption however it often causes maintained indexes to become out of date. You can also use this component with tables that have non-maintained indexes. Use TIdxUtl with the TTUtility component for a complete table maintenance solution.

The TIdxUtl component does not use the TUtility.DLL. All calls access only are made directly the Borland Database Engine only.

Since TIdxUtl does not use the TUtility.DLL you could develop an application without the table verify and repair functionality but with index checking and regenerating and not worry about including TUtility.DLL as a distribution file.

See Also TTUtility Component

Properties

▶ Run-time only
🔑 Key properties

- 🔑 Password
- 🔑 RegenAll
- Table
- 🔑 TableName

Methods

(New topic text goes here.)

- CheckIndexes
- RegenIndex

Events

▶ Run-time only
🔑 Key properties

🔑 onInfoIdxCheck
🔑 onInfoIdxRegen

Password Property

Applies To - TIdxUtl

Declaration

property Password : String;

Description

If the table has a master password it must be specified in this property otherwise CheckIndexes and RegenIndex will not run.

RegenAll Property

Applies To - TIdxUtl

Declaration

property RegenAll : Boolean

Description

Set this property to TRUE if you want RegenIndex to regenerate all indexes, including the indexes that are up to date.

Table Property

Applies To - TIdxUtl

Declaration

property Table : TTable;

Description

This property was added as an optional way of specifying the TableName. You can drop a TTable component into your form and then use this property to select that TTable component. Assigning this component automatically assigns a value to the TableName property. This may be easier than assigning the TableName property directly if the path name is very long.

Note that this property is completely optional and is meant only as a way of assisting in the assigning the TableName property. Also note that using this method to assign Tablename uses slightly more resources than assigning TableName directly,

TableName Property

Applies To - TIdxUtl

Declaration

property TableName : TFileName;

Description

Assign this property the name of the table to be checked and/or regenerated. This should be a completely qualified file name, including the path.

CheckIndexes Method

Example

Applies To - TIdxUtl

Declaration

function CheckIndexes : Boolean;

Description

Execute CheckIndexes from your application to perform the check of the table identified in the TableName property. CheckIndexes examines the information stored in the tables header to determine if the index is out of date. Define an onInfoIdxCheck event to receive status information as the CheckIndexes process runs.

If CheckIndexes finds that any of the Table's indexes are out of date then it returns FALSE. If all indexes are up to date then TRUE is returned.

If the table is password protected then the Password property must be assigned, otherwise you will receive a run time error.

RegenIndex Method

Example

Applies To - TIdxUtl

Declaration

procedure RegenIndex

Description

Execute RegenIndex from your application to regenerate out of date indexes.

If the RegenAll property is set to FALSE then only the indexes that are out of date are regenerated. If RegenAll is TRUE than all indexes are regenerated. You can also control which indexes are regenerated by defining the onInfoIdxRegen event.

Note : The Borland Database Engine User's Guide states, "The effect of regenerating a maintained index is that it becomes more efficient and compact. (Frequent updates can fragment an Index)". pp277.

So... it a good idea to periodically regenerate all your indexes even if CheckIndexes shows all indexes are up to date.

If the table is password protected then the Password property must be assigned, otherwise you will receive a run time error.

onInfoIdxCheck Event

Example

Applies To - TIdxUtl

Declaration

property onInfoIdxCheck : TInfoIdxCheckEvent

Description

The onInfoIdxCheck   event reports information about the CheckIndexes process as it runs. The TInfoIdxCheckEvent event is defined as:

```
TInfoIdxCheckEvent = procedure(
        Sender: TObject;
        IndexName : String;
        IsUptoDate : Boolean) of object;
```

Where IndexName reports the name of the index being checked and IsUptoDate reports TRUE if the index is up to date and FALSE if the index is out of date.

onInfoIdxRegen Event

Example

Applies To - TIdxUtl

Declaration

property onInfoIdxRegen : TInfoIdxRegenEvent

Description

The onInfoIdxRegen event reports information about the RegenIndex process as it runs. You can also use the onInfoIdxRegen event to selectively regenerate indexes in the selected table. The TInfoIdxRegenEvent is defined as:

```
TInfoIdxRegenEvent = procedure(
        Sender: TObject;
        IndexName : String;
        IsUptoDate : Boolean;
        var Skip : Boolean) of object;
```

Where IndexName is the name of the index to be regenerated. IsUptoDate reports TRUE if the index is up to date and FALSE if the index is out of date.

Use Skip to selectively regenerate indexes. If you set Skip to FALSE then the index will be regenerated even if it is already up to date. Set Skip to TRUE if you do not want the index regenerated. Skip has default values that will apply if you do not set its value. If the IsUptoDate parameter is TRUE then the default value for Skip is TRUE. If IsUptoDate is FALSE then the default value for Skip is FALSE.

Example

```
procedure TFormIndxProjMain.ButtonCheckIndexesClick(Sender: TObject);
begin
  if not IdxUtl1.CheckIndexes then
    MessageDlg('Index(es) are out of date and should be regenerated.',
                 mtWarning, [mbOk], 0);
end;
```

Example

```
procedure TFormIndxProjMain.ButtonRegenIndexesClick(Sender: TObject);
begin
  IdxUtl1.RegenIndex;
end;
```

Example

```
procedure TFormIndxProjMain.IdxUtl1InfoIdxCheck(Sender: TObject;
  IndexName: String; IsUptoDate: Boolean);
begin
  if IsUptoDate then
    SendToLog('Index ' + IndexName + ' is up to date.')
  else
    SendToLog('INDEX ' + Uppercase(IndexName) + ' IS OUT OF DATE.');
end;
```

Example

```
procedure TFormIndxProjMain.IdxUtl1InfoIdxRegen(Sender: TObject;
  IndexName: String; IsUptoDate: Boolean; var Skip: Boolean);
begin
  if IsUptoDate then
  begin
    if MessageDlg(IndexName + ' is not out of date do you want to regenerate
it anyway?',
                 mtInformation, [mbYes,mbNo], 0) = mrYes then
    begin
      Skip := False;
      SendToLog(IndexName + ' is being regenerated.');
    end
    else
    begin
      Skip := True;  {this line is not necessary cause Skip is true by
default}
      SendToLog(IndexName + ' not out of date and not being regenerated.');
    end;
  end
  else
  begin
    Skip := False; {this line is not necessary cause Skip is false by
default}
    SendToLog(IndexName + ' is being regenerated.');
  end;
end;
```