TTUtility - A Delphi Paradox Table Verify & Repair Component

1. Introduction

Now there's an answer to data corruption problems that does not include the words "all nighter". 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.

This component was developed to give those responsible for data integrity a break.

Whether you are a contracted consultant or an internal dB administrator, TTUtility is a **must** for your component pallet and your personal well being.

Features

*** Fully implements Borland's TUtility API in an easy to use Delphi Component.**
*** Contains separate Verify and Rebuild execution methods.**
*** Supports Borrowed Table structures for Rebuilds.**
*** Uses canned or customized status guages (all callbacks are supported).**
*** Provides all the analytical information necessary to choose the correct rebuild method.**
*** Includes complete and installable component help.**
*** Special bonus component provides for checking and regenerating out of date Indexes.**

You also get three demonstration projects, one for single table verify/rebuild, one for batch table verify/rebuild and one that shows how to check and regenerate out of date indexes. These projects have been specifically designed to drop into existing Delphi applications with little or no modification.

Use the TUtility component to add verify and rebuild capabilities to your Paradox/Delphi applications. You can even create safe, multi-table, "as you sleep", batch verify and rebuild jobs.

Note that not all table corruption is fixable using this component - sometimes a table will be in such a corrupted state that the only method of restoring that table is from a backup. We recommend making periodic backup copies of your data.

The TUtility component is actually a suite of components that includes the main TTUtility component, two canned status dialog components and the bonus index verification and regeneration component.

All this adds up to fast and easy. Really, quite RAD!

This Document Outline

2. Disclaimer and Legal Stuff - PLEASE READ

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.

Use of this product is at your own risk. Neither Out & About Productions or Borland International is responsible for any damage to your data as a result of using this component (TTUtility) or the underling TUtility.DLL The TUtility.DLL is a unsupported product from Borland International. Borland International has made the TUtility.DLL (included with this product) freely redistributable with an application developed with the Borland Database Engine.

Along with the TUtility.DLL you may distribute any application that includes the TTUtility component and supporting components with no additional royalties beyond your initial license registration fee.

If you use the TTUtility component to develop an application where you also deliver the application's source then this is considered an additional license and the receiving party must license a copy of TTUtility from Out & About Production.

The TUtility.DLL and its API are NOT official Borland products, and as such Borland does not support the published API. You have the right to use this technical information subject to the terms of the no-nonsense License Statement that you received with Delphi. Out & About products are licensed with exactly the same rules as documented in Borland's no-nonsense License Statement.

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.

The TTUtility component is copyright 1995 by Out & About Productions and is protected by international law. We reserve all rights.

3. Installation

Complete installation of this component requires that you
1. Copy the distribution files to your disk drive.
2. Install the component into the Delphi Component Pallet.
3. Merge the components help into the Delphi help system.

1. Copy the distributed to your disk drive.
   - Unzip the product ZIP archive to your hard drive.
   - Create a directory and unzip the files into the new directory.
   - Copy the TU_INT.HLP file to the \Delphi\Bin directory.
   - Copy the TUTILITY.DLL to your \windows\system directory
   - Copy the TU_INT.KWF file to the \Delphi\Help directory.

You may keep the other files together in one directory as they are now, or seperate the component files from the demo files. Many people like to have a single third party component directory to keep their component search path short in Delphi. You can go ahead and do this, just make sure that all the files listed below as component files stay together and the files listed as demo files together.

(Component Files)

TU. DCU	TTUtility Unit File
TUIDX .DCU	TIdxUtl Unit File
REBDLG.DCU	Rebuild Status Dialog Unit File
REBDLG.DFM	Rebuild Status Dialog Form File
VERDLG.DCU	Verify Status Dialog Unit File
VERDLG.DFM	Verify Status Dialog Form File
TUTILITY.DLL	TUtility DLL from Borland. Must go into a directory on DOS search path.
TU_INT.DCR	Component Set Resource File
TU_INT.PAS	Component Set Registration File
TU_INT.RES	Component Set Resource File
TUSTRS.RES	Component Set Resource File
TUTILITY.WRI	This Document
README.TXT	Last minute notes.

(Help related files}

TU_INT. HLP	Help file must be copied to \delphi\bin
TU_INT.KWF	Help key word file. Must be copied to \delphi\help

(Demo #1 Files - Single File Verify & Rebuild)
PROJTU   DPR
PROJTU   RES
TUMAIN   DFM
TUMAIN   PAS
VWERRDLG DFM
VWERRDLG PAS

(Demo #2 Files - Batch File Verify & Rebuild)
BTCHPROG DPR
BTCHPROG RES
BATCHDLG DFM
BATCHDLG PAS
ERRTBDLG DFM
ERRTBDLG PAS
GETDLG   DFM
GETDLG   PAS
STATDLG DFM
STATDLG PAS

(Demo #3 Files - Index Chech and Regenerate)
INDXMAIN DFM
INDXMAIN PAS
INDXPROJ DPR
INDXPROJ RES

2. Install the component into the Delphi Component Pallet.

- Start Delphi, choose Options|Install Components and then
- Click the Browse button and locate TU_INT.PAS in your new directory.
- Select it.
- Press OK in the Install Components dialog and wait for the Library to rebuild.

3. Merge the components help into the Delphi help system. OK this is the toughest part install so pay close attention.
Install the Keyword File
a. Exit Delphi if it is running
b. Make a backup of \delphi\bin\delphi.hdx
c. Run the HelpInst application from \delphi\help (should be in you Delphi Group)
d. Open \delphi\bin\delphi.hdx
e. If the existing KWF files report "not found" then add \delphi\help to the search path by selecting Options|Search Path.
f. Select the Keywords|Add File menu choice and select TU_INT.KWF from \delphi\help directory where you copied it in step 1 above.
g. Select File|Save
h. Exit HelpInst
The new components help has now been merged into Delphi's help system.

That's it the component should now be correctly installed. Run the demos and have fun. We have not included any sample corrupt tables so you will need to create your own. One way to do this is to simply delete a table's .PX

How to Move the components to a Different Pallet.

When the TTUtility component set installs it defaults to a new pallet page called "Paradox" if you would like to move the components to a different component page you may. All you need to do is open the TU_INT.PAS file and edit the registration statements at the end of the file. For example, to move the components to a pallet page called "New Components" change;

```
procedure Register;
begin
 RegisterComponents('Paradox', [TTUtility]);
 RegisterComponents('Paradox', [TVerifyDlg]);
 RegisterComponents('Paradox', [TRebuildDlg]);
 RegisterComponents('Paradox', [TIdxUtl]);
end;
```

to

```
procedure Register;
begin
 RegisterComponents('New Components', [TTUtility]);
 RegisterComponents('New Components', [TVerifyDlg]);
 RegisterComponents('New Components', [TRebuildDlg]);
 RegisterComponents('New Components', [TIdxUtl]);
end;
```

4. General Implementation Rules and Design Considerations

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.

5. Outline for a successful Verify & Rebuild

This section outlines the basic steps you should incorporate into your application to ensure the best results form the verify/rebuild process. This is intended to be a brief outline and does not cover the finer points which are covered later in the reference section. These steps outline the basic steps you should follow to do a verify and rebuild. These do not outline a complete application. Refer to the demonstration projects to see examples on how these guidelines are implemented.

Setup

First you need to set up the TTUtility component by assigning certain properties.

1. **(Required)** Select a table to verify by assigning a value to the TableName property. This assignment can happen either at design time or at run time. Assigning a value to TableName has two side effects. The TblInfo property is given value and the header of the table is verified. The TblInfo property is a record structure that contains valuable information about the table. Of particular interest is the bValidInfo and iRecords fields in this structure. If bValidInfo is FALSE or iRecords is zero (and you know there are records in the table) then it means that the header of the table is corrupt. This is useful information and will be used later. Also, the iErrorLevel property should be checked to see the results of the header verify. Chances are that if either bValidInfo or iRecords delivered bad news then iErrorLevel will also. (See TableName, TblInfo, and iErrorLevel in the reference section for more info). OK, so just by assigning the TableName property you've actually done a whole lot of the information gathering work.

2. **(Optional)** 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. It is not required that you assign values to these properties since defaults will be assigned if you do not. These properties can be assigned at either design or run time.

3. **(Optional but Recommended)** Assign a value to the AltStructName property. This is the name of a known good table that ExecuteRebuild can borrow the structure from. It is always safer to rebuild a table using a good table header than it is to rebuild using a corrupt table header. This property can be set at either design or run time.

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.

4. **(Required)** Assign a value to the Password property if the table has a master password. (See the warnings above and below). Rember that the TUtility.dll has no way of knowing if this password has been assigned incorrectly.If a table is rebuilt without the correct password the resulting table will have no records in it. This property can be set at either design or run time.

5. **(Optional)** Either drop the Verify and Rebuild Status Dialogs into your form or define the onInfoVerify and onInfoRebuild events to respond to the status messages from the Verify and Rebuild processes. This gives you all the nice visuals. Note that there is a performance penalty in activating these status callbacks but they are a lot better than a blank screen. (See the reference section for more details).

Execution
Once the minimum setup is completed you are ready to verify and possibly rebuild the table named in the TableName property.

1. From some event in your application run the ExecuteVerify public method. If errors are found, ExecuteVerify will create an error table with the name specified in the tErrTableName property. The structure of this table is documented in the ExecuteVerify description in the reference section. ExecuteVerify will also set the iErrorLevel public property to the highest error level found in the error table. (This is the worst error found).

It is probably always a good idea to give the user a way to review the error table after verify is complete. Check out the Demo #1 to see an easy way to do this.

2. Now it's decision time. In you code you must read the iErrorLevel, TblInfo.bValidInfo, and the TblInfo.iRecords properties to determine whether to run ExecuteRebuild using the table structure of the table being rebuilt, or to borrow the table structure from another table. See Appendix B for a short discussion on making this decision. To use the table structure of the table being rebuilt pass ExecuteRebuild the pCurrentTblDesc property. To borrow the structure from another good table of the same structure pass ExecuteRebuild the pAltTblDesc property. From some event in you application run the ExecuteRebuild with the table description you have decided on.

Clean up
Once your table is rebuilt it always a good idea to clean up any of the side effect files created by ExecuteVerify and ExecuteRebuild. In Paradox for Windows the table repair utility only deletes the error table created by the verify step. The other tables (backup table, key violation table and problems table) are left to the user to clean up. You should do what ever is right for your application.

6. Reference TTUtility

TTUtility Public Properties

constructor Create(AOwner: TComponent)
You will only need this if you create your instance at runtime or if you create a descendent component. Been there done that.

destructor Destroy;
Only of value if you are creating a new component with TTUtility as the descendent.

property iErrorLevel : Word; (Read Only)
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. Everything's 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. (use AltTblInfo)
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).

procedure ExecuteVerify;
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**	
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 table's file date
Time	T		The table's 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.

procedure ExecuteRebuild(pTableDesc : pCRTblDesc);

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.

procedure ExecuteVerifyRebuild;
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 the execution of this procedure. See the documentation on those events for further information.

property TblInfo: TTableInfo; (ReadOnly)
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

property AltTblInfo: TTableInfo; (ReadOnly)
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. See TblInfo above for a more complete description of the fields in TTableInfo.

property pCurrentTblDesc: pCRTblDesc (Read Write)
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 all the discussion in the section on ExecuteRebuild above for more information.

ALSO NOTE - The user should not destroy or modify the Table Desc structure that is passed back by GetCRTblDesc 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 NOTE 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.

property pAltTblDesc: pCRTblDesc; (ReadOnly)
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. See all the discussions above for more information.

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 GetCRTblDesc it is to be looked at and passed to Rebuild table only.

TTUtility Published Properties

property AltStructAlways: Boolean;
Set this value to true if you always want the table specified in AltStructName to determine the structure of the rebuilt table. This is used only by ExecuteVerifyRebuild.

property AltStructName: TFileName;
Assign this property the name of the table to use as the structure for the rebuild. This should be a completely qualified 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.

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

property CBActive : Boolean;
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. This would be done for reasons of performance.

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.

property CBRebuildDialog: TRebuildDlg;
Assign this the value of the TRebuildDlg component on your form. This is the easiest way to get status information during ExecuteRebuild.

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

property Name : String;
Enter the component name here.

property Options: TVerifyOptions;
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.

property Pack : Boolean;
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.

property Password : String;
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.

property Table : TTable;
This property was added as an optional way of specifying the TableName (see 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,

property TableName : TFileName;
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.

Property Tag : LongInt;
The Tag property is available to store an integer value as part of a component. While the Tag property has no meaning to Delphi, your application can use the property to store a value for its special needs.

property tBkUpTableName: TFileName;
Assign this property the name of the backup table created by ExecuteRebuild. The default is TableName + '_'. So 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 .

property tErrTableName : TFileName;
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).

property tKeyVTableName: TFileName;
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).

property tProbTableName: TFileName;
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).

property OnInfoRebuild: TInfoRebuildEvent;
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) of object;                   {The data to be acted on}
```

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.
Here's a quick example of how this event might look:

```
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;
```

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.

property OnInfoVerify: TInfoVerifyEvent;
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) of object;                  {The data to be acted on}
```

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. Here's a quick example of how this event might look:

```
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;
```

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.

property OnInfoVerReb: TInfoVerRebEvent;
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.

7. Reference TIdxUtl

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 not 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.

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 section 10 of this document for a simple demonstration on the use of the TIdxUtl component.

TIdxUtl Public Properties

constructor Create(AOwner: TComponent)
You will only need this if you create your instance at runtime or if you create a descendent component. Been there done that.

destructor Destroy; override;
Only of value if you are creating a new component with TTUtility as the descendent.

Function CheckIndexes : Boolean;
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.

procedure RegenIndex;
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.

TIdxUtl Published Properties

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

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

property Table: TTable
This property was added as an optional way of specifying the TableName (see 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,

property TableName : TFileName
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.

property onInfoIdxCheck :   TInfoIdxCheckEvent
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.

property onInfoIdxRegen : TInfoIdxRegenEvent
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.

8. Demo #1 - A Simple Verify & Rebuild Application

The first demo project is PROJTU.DPR. It was developed in parallel with the TTUtility component as a way to test the validity of the component and to insure that all the needed functionality and logic is there. As a result the final project mimics the table repair utility found in Paradox for Windows in many ways.

In fact, the major difference is that this demo does not allow the user to manually define the table structure of the table to be rebuilt. The complexity, potential for error, and further data corruption of allowing the user to define the rebuild structure is so great that we believe that it should be avoided at all costs.

The project consists of the main form (TUMAIN) and a form to display the records in the Error Table created by ExecuteVerify (VWERRDLG).

Take a look at the way the two forms are set up. Pay particular attention to the way the properties are set in the TTUtility component.

The code in the TUMAIN.PAS unit is full of comments. You should review this code in conjunction with this documentation. There is nothing very tricky here. Notice the logic in the ButtonRebuildClick method that decides how to rebuild the table.

9. Demo #2 A Batch Verify & Rebuild Application

The BtchDemo.dpr application is a good start at a general purpose application that does batch verify and rebuilds of groups of tables. The idea behind this application is to give a general tool that the user can use to set up a batch of tables that need to be verified and possibly rebuilt. The user defines a list of tables that make up the batch and then process the batch. What follows describes this process.

The BtchDemo project demonstrates a number of the properties, methods and events in TTUtility. Specifically it shows how to:
* Define your own verify and rebuild status dialogs using onInfoVerify and onInfoRebuild.
* How to set up and use ExecuteVerifyRebuild to process a number of tables.
* How to build a batch processing log using the information returned in OninfoVerReb.
* How to do a batch verify with no rebuild and collect information on a number of tables in one error table.
* Plus lots of other little things.

Load the BtchDemo.DPR into Delphi and run it.

Step #1 - Select Batch
The first thing to do in using this application is to define a batch of files to be Verified/Rebuilt. The batch is defined by creating a database table where each record in the table represents one table to be verified and possibly rebuilt. When you run this demo the first time you must define a batch table. To do this follow these instructions:
1. Click on the Select Batch button and then on the New Batch Button.
2. Select a name and directory for the new batch. We suggest putting the batch in the same directory where you keep the empty tables you use for borrowing file structures (AtlStructTables) and press OK.
3. Now define the batch. Start by clicking on the add new record (+) button on the navigator. For each table in the batch you must enter the table's name. Also, the "Get Structure Table" field is absolutely necessary to ensure that the batch processes to completion. The other fields need to be defined separately for each table in the batch so that a complete record of the batch process is kept and the batch results can be studied. Use the navigator buttons to insert and save new batch records. One record for each table in the batch. When you are done defining the batch press OK.

Step #2 - Confirm Batch
Click the Confirm Batch button to do a quick check of the records in the batch. This currently does minimal checks but is here for completeness (see the code in BtchMain.pas). It is always a good idea to place as many quick checks on long batch processing runs up front so that the actual run goes a smoothly as possible. Additional checks could be built into this button's onClick event.

Step #3 - Verify Only
Clicking on this button processes the complete batch but only verifies the tables. No rebuild is attempted. Under some circumstances it is useful to know what errors are in the tables prior to doing the rebuild. The Verify Only button creates an ErrorTable the includes the errors for all the tables in the batch.

Step #4 - View Error Table
Click this button to view the error table created in step #3.

Step #5 - Verify & Rebuild
Click on the Verify & Rebuild button to verify and rebuild the tables in the batch. Information about the status of each table before the rebuild and the results of the rebuild for each file are stored in a number of different list/tables. Here is a list of your sources of information:

On Screen Log - The log visible on the screen records information about the general batch process.
Key Violations - The tables specified in the batch definition.
Problem Records - The tables specified in the batch definition.

The data in these source should be studied for each table in the batch to determine the successfulness of the rebuild.

Step #6 - Save Batch Log
Click here to save the on screen log to a file. You can also clear the on screen log from here.

Note that example demonstrates only one possible way of setting up a batch verify/rebuild. Other possibilities were suggested by the beta testing team. Some of these suggestions included:

- Eliminate the batch table and have the user just specify a directory path or alias. The batch verify/rebuild application would then verify (and possibly) rebuild every table in that directory/alias.
- Expand the Batch.DB structure to include scheduling information so the files in the batch would be verified/rebuilt on a regular periodic basis.

I leave these enhancements up to you folks.

10. Demo #3 A Index Check and Regenerate

The IndxProj.DPR demonstrates the use of the TIdxUtl component. It is a very simple project. Take a look at the TUIDX.PAS file to see how it's done. The inplementation of the two information events, IdxUtl1InfoIdxCheck and IdxUtl1InfoIdxRegen are probably the most interesting.

Appendix A - TUtility API Notes

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.

Let me say that 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 TUtility's 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. Here's 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.

Appendix B - 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. I'm sure you've all seen the bumper sticker:

Data Corruption Happens!

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 let us say again: "*When ever 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. It has even been suggested that ExecuteVerify should be part of the normal backup routine. This seems like a good idea if time permits. The sooner you catch and correct a corrupt table problem the better off you are.

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.

Appendix C - TUtility Error Codes

Error messages associated with the error codes returned in the Error Code field of the error table created by ExecuteVerify.

Error	**Message**
1	Unable to open file %s
2	Invalid Header size %u
3	Invalid file size %ld: %s
4	Invalid block size %u: %s
5	Real header size %u is invalid
6	Invalid sort byte in header
7	Sum of field sizes does not equal record size
8	Not enough memory to complete operation
9	Header size is not a multiple of 2K
10	Unsupported field type in field %u
11	Field name greater than 25 char
12	Header real size %u is greater than start of data %u
13	File type %s does not match extension %s
14	Unknown file type
15	Keyed db missing px file.
16	Maintained index missing .y* file.
17	Invalid record size
18	Unable to read file %s
19	Table header reports damage
20	Total blocks (%u) greater than max blocks (%u)
21	Header inconsistent
22	Invariant Field ID %u for Field %u is greater than max field id %u
23	First block greater than total blocks
24	Last block greater than total blocks
25	Index version does not match table version
26	Index record size does not match table keysize
27	Block %u is in the chain twice
28	Number of records %i in block %u does not match index %li
29	Reported header size is not large enough to hold header information
30	Header size (%u) + data size (%ld) does not equal filesize (%ld)\nDifference: %ld bytes
31	Block pointers in block %u do not match pointers in block %u
32	First record of Block %u does not match index
33	Forward block pointer in block %u is out of range
34	Number of records in index file is greater than the number of blocks in data file
35	Record %lu is not in sort order
36	Empty block (%u) in used block chain
37	Number of blocks is greater than the number of records in index
38	Last block in chain (%u) does not match header (%u)
39	Number of records found (%lu) does not match number of header records (%lu)
40	Number of blocks in used block chain (%u) does not equal used blocks in header (%u)
41	Empty block %u reports records

42	Empty block pointer in block %u is out of range
43	Total Blocks reported by header (%u) does not equal blocks found in chain (%u)
44	Previous block pointer in block %u is out of range
45	Secondary Index %s does not match table
46	Used blocks greater than total blocks\n\tUsed blocks = %u\n\tTotal blocks = %u
47	First block = 0 and header reports used blocks
48	Data start (%u) is greater than file size (%lu)
49	Hanging .PX file
50	Hanging .Y** file
51	Table is full. Unable to allocate block
52	BTree record mismatch from block (%u) record(%i) to block (%u)
53	BTree record number mismatch from block (%u) record(%i) to block (%u)
54	Pointer to last record (%i) in Block %u is not consistent with record size (%u)
55	Alpha Field %s in record %lu contains low ascii chars
56	Field %s of record %lu does not match secondary index %s
57	Field number out of bounds in secondary index %s
58	Invalid number of fields.
59	Field size %u is invalid for field type %s in field number %u
60	Invalid table format
61	Block %u points to itself
62	Number of key fields is greater than the number of fields in the table
63	Write-protect flag
64	Sort name missing from header
65	The indices are out of date:\nA rebuild is recommended!
66	Modified flags are set:\nA rebuild is recommended!
67	Invalid memo field
68	Header not consistent with format
69	Header inconsistent with version
70	Invalid sequence number
71	Data in field %s of record %lu does not match .MB file
72	Blob pointer offset for field %u of record %lu points to a memo block header invalid type
73	Blob pointer index for field %u of record %lu is invalid
74	Blob pointer length for field %u of record %lu does not match .MB file
76	Memo file signature is not valid
75	Blob pointer sequence number for field %u of record %lu does not match .MB file
77	Unsupported memo version
78	Unsupported memo free block size
79	Unsupported memo increment size
80	Unsupported memo list block size
81	Unsupported memo small block index size
82	Unsupported memo big block size
83	Unsupported memo small block size
84	Missing .MB file
85	File size is too large to be a replica table
86	Replica table reports records
87	Float field %s in record %lu has denormal value
88	Float field %s in record %lu has exponent smaller than accepted minimum
89	Float field %s in record %lu has +/- infinity value
90	Float field %s in record %lu has quiet NAN value
91	Float field %s in record %lu has signaling NAN value
92	Float field %s in record %lu has exponent larger than accepted maximum
93	Used memo block in free list chain
94	Free memo size does not match size reported in free list

95	Language driver unsupported
96	Invalid formatted memo header
97	Secondary Index %s is out of date

Appendix D - TTUtility Component Interface and structures

```
Type
 { Exception classes }
 ETUtilityError = class(Exception);

 { Verify Options }
 TVerifyOption = (
   vTU_Append_Errors, {append errors to an existing error table}
   vTU_No_Secondary,  {Bypass secondary indexes}
   vTU_No_Warnings,   {Prevents warnings of secondary errors}
   vTU_Header_Only,   {Verify table header only}
   vTU_Dialog_Hide,   {Reserved for future expansion. Do not use}
   vTU_No_Lock);      {Do not lock table bering varified}
 TVerifyOptions = Set of TVerifyOption;

 { Verify Callback processes }
 TUVerifyProcess = (TUVerifyHeader, TUVerifyIndex, TUVerifyData, TUVerifySXHeader,
                   TUVerifySXIndex, TUVerifySXData, TUVerifySXIntegrity,
                   TUVerifyTableName);

 { Call back info for Verify Callback function }
 TVerifyCBData = record
   PercentDone: word;
   TableName: String[82];
   Process: TUVerifyProcess;
   CurrentIndex: word;
   TotalIndex: word;
 end;

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

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

 TInfoRebuildEvent = procedure(
     Sender: TObject;              {Where the message came from}
     RebuildCBRec: TRebuildCBData) of object; {The data to be acted on}

 { Type used by the FOnInfoVerReb event to tell app which process is executiong }
 TUVerRebProcess = (TUVerifying, TURebuilding);

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

 { TTableInfo Record }
 TTableInfo = Record
   sTableType      : String[32];    { Driver type }
   iFields         : Word;          { No of fields in Table }
   iRecSize        : Word;          { Record size (logical record) }
   iKeySize        : Word;          { Key size }
   iIndexes        : Word;          { Number of indexes }
   iValChecks      : Word;          { Number of val checks }
   iRefIntChecks   : Word;          { Number of Ref Integrity constraints }
```

```
    iRestrVersion   : Word;          { Restructure version number }
    iPasswords      : Word;          { Number of Aux passwords }
    bProtected      : Bool;          { Table is protected by password }
    sLangDriver     : String[32];    { Language driver name }
    iBlockSize      : Word;          { Physical file blocksize in K }
    iRecords        : Longint;       { Number of records in table }
    bValidInfo      : Bool;          { Info available for this table }
  end;

  { TTUtility }
  TTUtility = class(TComponent)
  private
    FTableName: TFileName;
    FTable : TTable;
    FErrTableName: TFileName;
    FBkUpTableName: TFileName;
    FKeyVTableName: TFileName;
    FProbTableName: TFileName;
    FAltStructName: TFileName;
    FAltStructAlways: Boolean;
    FAlwaysRebuild : Boolean;
    FphTUSession: phTUSes;   { pointer TUTility session handle. }
    FOptions : TVerifyOptions;
    FiErrorLevel: word;     { Error Level of the table. }
    FCBActive : Boolean;
    FCBVerifyDialog: TVerifyDlg;
    FCBRebuildDialog : TRebuildDlg;
    FTableInfo : TTableInfo;
    FAltTableInfo : TTableInfo;
    FpCrDesc : pCRTblDesc;       { Pointer to Table Description }
    FOnInfoVerify: TInfoVerifyEvent; {The callback fires this through OnInfoUpdate}
    FOnInfoRebuild: TInfoRebuildEvent; {The callback fires this through OnInfoUpdate}
    FOnInfoVerReb : TInfoVerRebEvent;
    function ConvertName(const Name: string; Buffer: PChar): PChar;
    function GetATblDesc(aTable : TFileName): pCRTblDesc;
    function GetpCRTblDesc : pCRTblDesc;
    function GetpAltCRTblDesc : pCRTblDesc;
    function GetErrorName : TFileName;
    function GetBackupName : TFileName;
    function GetKeyViolName : TFileName;
    function GetProblemName : TFileName;
    function GetiOptions : Integer;
    function DestroyTUSession : phTUses;
    function NewphTUSession : phTUSes;
    procedure SetpCRTblDesc(value : pCRTblDesc);
    procedure SetTableName(const Value: TFileName);
    procedure SetErrTableName(const Value : TFileName);
    procedure SetBkUpTableName(const Value : TFileName);
    procedure SetKeyVTableName(const Value : TFileName);
    procedure SetProbTableName(const Value : TFileName);
    procedure SetAltStructName(const Value : TFileName);
    Procedure SetTableInfo(const Value : TFileName; var aInfoRec : TTableInfo);
    Procedure SetCBVerifyDialog(Const Value : TVerifyDlg);
    Procedure SetCBRebuildDialog(Const Value : TRebuildDlg);
    procedure FillTableInfoFromCURProps(const Value : TFileName; var aInfoRec : TTableInfo);
    procedure FillTableInfoFromTBLExtDesc(const Value : TFileName; var aInfoRec : TTableInfo);
    procedure CleanUpCrDesc;
  protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
    procedure Loaded; override;
    property ErrorName: TFileName read GetErrorName;
    property BackupName: TFileName read GetBackupName;
    property KeyViolName: TFileName read GetKeyViolName;
    property ProblemNameName: TFileName read GetProblemName;
    property iOptions : Integer read GetiOptions;
    procedure DoInfoUpdateVerify(pInfo: Pointer);
    procedure DoInfoUpdateRebuild(pInfo: Pointer);
```

```
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  property iErrorLevel : Word read FiErrorLevel;
  procedure ExecuteVerify;
  procedure ExecuteRebuild(pTableDesc : pCRTblDesc);
  procedure ExecuteVerifyRebuild;
  property TblInfo: TTableInfo read FTableInfo;
  property AltTblInfo : TTableInfo read FAltTableInfo;
  property pCurrentTblDesc: pCRTblDesc read GetpCRTblDesc write SetpCRTblDesc;
  property pAltTblDesc: pCRTblDesc read GetpAltCRTblDesc;
```

```
 published
   property CBActive : Boolean read FCBActive write FCBActive default false;
   property Options: TVerifyOptions read FOptions write FOptions default [];
   property Table : TTable;
   property TableName     : TFileName read FTableName     write SetTableName;
   property tErrTableName : TFileName read FErrTableName  write SetErrTableName;
   property tBkUpTableName: TFileName read FBkUpTableName write SetBkUpTableName;
   property tKeyVTableName: TFileName read FKeyVTableName write SetKeyVTableName;
   property tProbTableName: TFileName read FProbTableName write SetProbTableName;
   property AltStructName:  TFileName read FAltStructName write SetAltStructName;
   property AltStructAlways: Boolean  read FAltStructAlways write FAltStructAlways;
   property CBVerifyDialog: TVerifyDlg read FCBVerifyDialog write FCBVerifyDialog;
   property CBRebuildDialog: TRebuildDlg read FCBRebuildDialog write FCBRebuildDialog;
   property OnInfoVerify: TInfoVerifyEvent read FOnInfoVerify write FOnInfoVerify;
   property OnInfoRebuild: TInfoRebuildEvent read FOnInfoRebuild write FOnInfoRebuild;
   property OnInfoVerReb: TInfoVerRebEvent read FOnInfoVerReb write FOnInfoVerReb;
 end;
```

Appendix E - TTUtility Component Interface and structures

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

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

 TIdxUtl = class(TComponent)
 private
   FTableName: TFileName;
   FTable : TTable;
   FPassword : String;
   FRegenAll : Boolean;
   FTblFullDesc: TBLFullDesc;
   FDatabaseName : TFileName;
   FDatabase: TDatabase;
   FhCur : hDBICur;
   FOnInfoIdxCheck : TInfoIdxCheckEvent;
   FOnInfoIdxRegen : TInfoIdxRegenEvent;
   function ConvertName(const Name: string; Buffer: PChar): PChar;
   procedure FillTableInfoFromTBLExtDesc(const Value : TFileName);
   procedure SetPassword(const PW : String);
   Procedure SetTableInfo(const Value : TFileName);
   procedure OpenIndexList(TblName : TFileName);
   procedure SetTableName(const Value: TFileName);
   procedure SetTable(const Value: TTable);
 protected
   procedure Notification(AComponent: TComponent;
     Operation: TOperation); override;
 public
   constructor Create(AOwner: TComponent); override;
   destructor Destroy; override;
   Function CheckIndexes : Boolean;
   procedure RegenIndex;
 published
   property Password : String read FPassword write SetPassword;
   property RegenAll : Boolean read FRegenAll write FRegenAll;
   property Table: TTable read FTable write SetTable;
   property TableName     : TFileName read FTableName  write SetTableName;
   property onInfoIdxCheck : TInfoIdxCheckEvent read FonInfoIdxCheck write FonInfoIdxCheck;
   property onInfoIdxRegen : TInfoIdxRegenEvent read FonInfoIdxRegen write FonInfoIdxRegen;
 end;
```

TTUtility Ordering Information

The demonstration/shareware version of this components is fully functional inside the Delphi environment. Attempt's to run an application that uses this component will fail as a stand-a-lone executable. If you find this control useful and would like to use it in your applications you must order the regular version.

For $49(U.S.)(Calf. residents + 7% SST), you will receive the full version plus any updates for the next year. MasterCard and Visa are excepted. The component will be sent to you by Compuserve or Internet e-mail. The source code is available for an additional $40 (such a deal).

Or you can go SWREG and put it on your Compuserve bill.

SWREG # 6951 - Component without source	$49.00
SWREG # 6500 - Component with source	$89.00

To order please send me an e-mail including your credit card number, expiration date, name as it appears on the card. I would also like your mailing address and voice phone number but it's not completely nessary (you become a pin on a map of the world). If you are uncomffortable with the e-mail idea you can fax the info to me a (619) 566-0210, or, worse yet, write me at the address below. Please also include you e-mail address so I have some place to ship the product. If you are ordering through Compuserve the product will come as a ZIP file. If you are ordering through the Internet the product will come as a UUE file which you will need to comvert to a ZIP file.

Any technical questions regarding this component should forwarded to the email address below or the FAX number.

The above price includes all updates and upgrades to the component for a year and technical support via Compuserve/Internet.

Alec Bergamini
CompuServe ID 75664,1224
Internet : 75664.1224@compuserve.com

Out & About Productions
8526 Lepus Road
San Diego, CA 92126