TDbf manual

Micha Nelissen

11th September 2004

Contents

1 Introduction

TDBF is a freeware native data access component for all Borland Delphi language compatible environments. This includes Delphi, C++Builder, Kylix and FreePascal. It allows you to create very compact database programs which don't need any special installer programs. The DB engine code is compiled right into your executable. It has the following features:

- Works without the Borland Database Engine.
- Allows the use of all dBASE native type (character, numeric, logical, date, and memo). See property TableLevel.
- Memo files are supported, both text and binary so you can use fields with no size limit.
- File format 100% compatible with dBASE III+ or dBASE IV or dBASE for Windows.
- Support for Clipper and Visual FoxPro tables.
- Restructuring existing tables to drop, add or modify a current table structure while retaining table data.
- Multi-user access through BDE compatible locking. Only 1 user can lock a particular record for writing, but multiple users can read the record.
- Index support available for fast sorting, searching and ranging of big tables. Indexes supported include NDX and MDX index files.
- Expression parser for both indexes and filters.
- OS dependant multi-codepage and multi-locale support. This allows you to read and write tables in different codepages than your OS and to specify a sort order better suited for your language or locale. Currently, this is not yet supported on the Linux OS.

2 Class structure

2.1 TDbf

File: Dbf.pas

The TDbf class is the main class, and the main interface for most programmers. You open/-close dbase files, add/select/remove indexes with this class. It stores the current record number, the selected index, etc. TDbf is a TDataSet-descendant and therefore the usual delphi data-aware controls like TDBEdit, TDBNavigator, TDBGrid, etc. can be used.

2.2 TDbfFile

File: Dbf_DbfFile.pas

This class handles the dbf file itself, it remembers what index files are open, and updates them when needed and provides an interface to lock records. The split TDbf/TDbfFile was originally to share open dbf files, so more TDbf components can open the same file at the same time. However, this has multi-user problems, and now each TDbf component opens it's own TDbfFile.

2.3 TDbfFieldDef

File: Dbf_Fields.pas

TDbfFieldDef is an enhanced version of TFieldDef, handling native field types to VCL field types conversion and dbase fields specific features. One problem of VCL TFieldDefs is their inability to store the size and precision of float fields. To use this ability you need to create a set of TDbfFieldDefs instead.

3 File types

.dbf A dBase or FoxPro file. Contains the data of all fixed size fields. Records are fixed size.

.dbt A memo file. Memo contents are variable in size, and the memo field in the .dbf contains a pointer to a block number in the memo file which contains the actual memo data.

.ndx An index file. Contains one index, and is created and referenced in tdbf using it's filename and extension. For example: `Dbf1.AddIndex('index1.ndx', 'field1', []);` creates and opens an index file named "index1.ndx". This index file can be closed by calling `Dbf1.CloseIndexFile('index1.ndx');`. If you close the dbf file without closing the index files, they are automatically closed. Note that they are not automatically opened next time the dbf is opened again. For the index to be maintained, it must be opened after the tdbf is opened by calling `Dbf1.OpenIndexFile('index1.ndx');`.

.mdx A Multiple Index file. Essentially a collection of several NDX files in one. Each dbf file can have one accompanying .mdx file, if the dbf file is named file.dbf, the mdx file is named file.mdx. Indexes are created automatically in the accompanying mdx file if you name them without the extension .ndx; example: `Dbf1.AddIndex('index1', 'field1', []);`. Note that if an .mdx file is present, it's automatically opened and maintained, you don't need to call `OpenIndexFile` for it.

4 Expressions

Expressions are used in indexes and in filters. When adding an index with `AddIndex`, section 6.8, you need to supply an index expression. When you want to apply a filter, you need to supply a filter expression in the `Filter`, section 7.15, property. Functions and operators are case insensitive.

For index expressions, you must take care that if the result is a string, that then it's length is not longer than 100 characters. Use the `SUBSTR` function to get the first X characters, if it's too long. Example:

```
Dbf1.AddIndex('INDEX1', 'DTOS(DATEFIELD)+SUBSTR(LONGFIELD,1,10)+SUBSTR
   (LONGFIELD2,1,20)', []);
```

Supported operators:

`+`	Concatenate two strings, or add two numbers
`-`	Subtract two numbers
`*`	Multiply two numbers
`/`	Divide two numbers
`=`	Compare two strings, numbers or booleans for equality
`<>`	Compare two strings, numbers or booleans for inequality
`<`	Return iff first argument smaller than second argument
`<=`	Return iff first argument smaller than or equal to second argument
`>`	Return iff first argument greater than second argument
`>=`	Return iff first argument greater than or equal to second argument
`NOT`	Negate the boolean argument to the right
`AND`	Returns true iff first and second boolean argument are true
`OR`	Returns true iff first or second boolean argument is true

Supported functions:

`STR`(*num*[,*size*,*precision*])
: Converts a number *num* to a string with optional size and precision

num	number to convert to string
size	the total number of characters to output
precision	the number of digits to the right of the decimal point

`DTOS`(*date*)
: Converts *date* to a string with representation `YYYYMMDD`

`SUBSTR`(*str*,*index*,*count*)
: Extracts a substring from string *str* from position *index* and length *count*

`UPPER`(*str*)
: Returns the uppercase equivalent of string *str*

`LOWER`(*str*)
: Returns the lowercase equivalent of string *str*

5 FAQ

5.1 Does TDbf support images?

TDbf does support images. You need to make a table with TableLevel ≥ 4 to be able to do that. That's because the Memo needs to use level 4 (= binary) encoding. So set tablelevel to at least 4, make some fields, make a BLOB field (ftGraphic, whatever) and CreateTable. Attach a TDBImage to it, and it should work.

Another solution is attaching a TDBRichEdit to a memo field and pasting an image in it.

5.2 Why do deleted records not show up when using ixPrimary indexes?

When a record is deleted, it's key is deleted immediately from ixPrimary indexes, to prevent key violations when you want to insert another record with the same key. As the record is not in the index, it is not "visited", and therefore does not show up.

5.3 How can I make an index on a TDateTime field?

Use an expression index and the `DTOS` function:

```
Dbf1.AddIndex('INDEX1', 'DTOS(FIELD1)', []);
```

where FIELD1 is the ftDateTime field and INDEX1 the name of the index.

5.4 How can I sort the display of the records differently?

You need to add an index that sorts the records in the order you want. See for an example the previous question. Next, you need to "select" the index to be used to sort the records. This is done using:

```
Dbf1.IndexName := 'INDEX1';
```

where INDEX1 is the name of the index you created using `AddIndex`.

The index can be deselected by selecting the empty name index:

```
Dbf1.IndexName := '';
```

Note that you can create multiple indexes by calling `AddIndex` multiple times, but you can select only one index at a time.

5.5 Why does RecordCount return too many records?

When no index is active, the RecordCount property returns the total number of records in the table including deleted ones. When an index is active, it returns a number based upon internal index tree, usually very large. RecordCount is made for speed, and has the sequential property, so TDBGrid can approximately show using it's scrollbar the position of the current record in the range.

If you need exact number of active records, use the ExactRecordCount function, which is very slow (it reads all records from beginning to end).

5.6 How to know if a record is currently being edited by another user?

Just try to edit the record using `Dbf1.Edit;` and when a record "lock" is in place, meaning another user is editing the record, an exception will be fired.

5.7 How to handle different character sets and locale sorting?

When creating the table, you determine in what character set and sorting locale the data is stored and sorted. Make sure you are using tablelevel ≥ 4. In Dbf_Lang.pas you will find a list of all possible character set / locale combinations. Choose an apprioriate one and before creating the table set it, for example hex 22, hungarian locale, charset 852:

```
Dbf1.LanguageID := DbfLangId_HUN_852;
...
Dbf1.CreateTableEx(...);
```

The LanguageID and CodePage can also be queried of any opened table.

5.8 How to undelete records?

When records are `Delete`d, they are just marked as being deleted. To undelete them, set ShowDeleted to true, navigate to the record you want to undelete, and call `Undelete`.

5.9 How can I restructure a table?

To restructure a table, create a new TDbfFieldDefs collection with the new structure of the table, and pass that to `RestructureTable`. The `CopyFrom` property determines the index of the field to copy data from. Note that when using `TDbfFieldDefs.Assign` to copy the structure of the fields, the CopyFrom fields are assigned automatically, if you have Delphi 5 (or BCB 5) or higher. A `CopyFrom` property with value `-1` indicates a new field.

See also `RestructureTable`, section 6.35.

6 Methods

6.1 GetFieldData

```
function GetFieldData(Field: TField; Buffer: Pointer): Boolean;
   override;
```

{From Borland Help} Most applications do not need to call GetFieldData. TField objects call this method to implement their GetData method.

The Field or FieldNo parameter indicates the field whose data should be fetched. Field specifies the component itself, while FieldNo indicates its field number. The Buffer parameter is a memory buffer with sufficient space to accept the value of the field as it exists in the database (unformatted and untranslated). NativeFormat indicates whether the dataset fetches the field in C++Builders native format for the field type. When NativeFormat is false, the dataset must convert the field value to the native type. This allows the field to handle data from different types of datasets (ADO-based, BDE-based, and so on) in a uniform manner.

GetFieldData returns a value that indicates whether the data was successfully fetched.

GetFieldData returns true if the buffer is successfully filled with the fields data, and false if the data could not be fetched.

6.2 Resync

```
procedure Resync(Mode: TResyncMode); override;
```

TDbf supports disabling of resync calls. See property DisableResyncOnPost.

6.3 CreateBlobStream

```
function CreateBlobStream(Field: TField; Mode: TBlobStreamMode):
   TStream; override; {virtual}
```

{From Borland Help} Call CreateBlobStream to obtain a stream for reading data from or writing data to a binary large object (BLOB) field. The Field parameter must specify a TBlobField component from the Fields property array. The Mode parameter specifies whether the stream will be used for reading, writing, or updating the contents of the field.

Blob streams are created in a specific mode for a specific field on a specific record. Applications should create a new blob stream every time the record in the dataset changes rather than reusing an existing blob stream.

6.4 Translate

```
{$ifdef DELPHI_4}
function Translate(Src, Dest: PChar; ToOem: Boolean): Integer;
   override; {virtual}
{$else}
procedure Translate(Src, Dest: PChar; ToOem: Boolean); override; {
   virtual}
{$endif}
```

The data stored in a DBF file is written in a specific codepage, the "OEM" codepage. Windows uses the "ANSI" codepage to display data. This function translates between these

codepages.

Specifying true for ToOem translates from Windows to DBF. Specifying false for ToOem translates from DBF to Windows.

6.5 ClearCalcFields

```
procedure ClearCalcFields(Buffer: PChar); override;
```

An internal method.

6.6 CompareBookmarks

```
function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer;
   override;
```

{From Borland Help} Call CompareBookmarks to determine if two bookmarks are identical or not. Bookmark1 and Bookmark2 are the bookmarks to compare.

If the bookmarks differ, CompareBookmarks returns 1. If the Bookmarks are identical, or both bookmarks are NULL, CompareBookmarks returns 0.

6.7 CheckDbfFieldDefs

```
procedure CheckDbfFieldDefs(DbfFieldDefs: TDbfFieldDefs);
```

Checks if you used invalid field types in your TDbfFieldDef definitions taking into account the current TableLevel. When using TableLevel smaller than 7, not all types are possible.

6.8 AddIndex

```
{$ifdef DELPHI_5}
procedure AddIndex(const AIndexName, Fields: String; Options:
   TIndexOptions; const DescFields: String='');
{$else}
procedure AddIndex(const AIndexName, Fields: String; Options:
   TIndexOptions);
{$endif}
```

Name is the name of the new index. Name must contain an index name with length shorter than or equal to 10.

Fields is a AnsiString value containing the field or an expression on which the new index will be based.

Options is a set of attributes for the index. The Options parameter may contain any one, multiple, or none of the TIndexOptions constants: ixPrimary, ixUnique, ixDescending, ixCaseInsensitive, and ixExpression.

- ixPrimary specifies a distinct unique index. An exception will be thrown when you try to insert 2 equal keys.
- ixUnique specifies an unique index. Duplicate key entries will be ignored.
- ixDescending specifies reverse sorting order.
- ixCaseInsensitive is not used.
- ixExpression need not be given; it is autodetected when the Fields parameter is parsed.

6.9 RegenerateIndexes

```
procedure RegenerateIndexes;
```

Clears all attached indexes, then recreates them from scratch.

6.10 CancelRange

```
procedure CancelRange;
```

{From Borland Help} Call CancelRange to remove a range currently applied to a table. Canceling a range reenables access to all records in the dataset.

6.11 SearchKey

```
function  SearchKey(Key: Variant; SearchType: TSearchKeyType): Boolean
   ;
function  SearchKeyPChar(Key: PChar; SearchType: TSearchKeyType):
   Boolean;
```

This function assumes you selected a particular index, using the IndexName property.

Key specifies the value to search for in the active index. You can specify the key as a variant type, or pass a native buffer using the SearchKeyPChar function. In the native case, pass a buffer according to the following rules based on index and key type:

- String index: a pointer to the first character of a null-terminated character array.
- MDX, numeric: a pointer to a buffer containing a BCD, a binary coded decimal in dBase format.
- NDX, numeric: a pointer to a double.

SearchType is one of the following:

- stEqual searches exactly Key. Returns false if no key matches.

- stGreaterEqual searches exactly Key or, if not found, the record which key is greater. Returns false if end of file is found.
- stGreater searches the first record which key is greater than specified Key. Returns false if end of file is found.

When false is returned as result, the cursor is not moved.

6.12 SetRange

```
procedure SetRange(LowRange: Variant; HighRange: Variant);
procedure SetRangePChar(LowRange: PChar; HighRange: PChar);
```

This function assumes you selected a particular index, using the IndexName property. The function applies a range to the current dataset. LowRange specifies the lower bound and HighRange specifies the upper bound. For the parameter formatting of the SetRangePChar function, see SearchKeyPChar.

6.13 PrepareKey

```
function  PrepareKey(Buffer: Pointer; BufferType: TExpressionType):
   PChar;
```

This function converts a key from a given type into the internal type. This is useful for numeric indexes: a pointer to an integer or int64, with BufferType etInteger or etInt64, will return the same key in BCD format. The result can then be used in a call to SearchKeyPChar.

Note: the result is a temporary buffer, which is invalidated by the next call to a tdbf method!

6.14 ExtractKey

```
procedure ExtractKey(KeyBuffer: PChar);
```

KeyBuffer: you need to provide a buffer of length 100 characters minimal.

If an index is selected, the function will extract the key from the current record's buffer and copy it into KeyBuffer.

6.15 UpdateIndexDefs

```
procedure UpdateIndexDefs; override;
```

An internal method that calls update on the fielddefs, which in turn causes both the field and index definitions to be read from the dbase and index files.

6.16 GetFileNames

```
procedure GetFileNames(Strings: TStrings; Files: TDbfFileNames); {
   $ifdef SUPPORT_DEFAULT_PARAMS} overload; {$endif}
{$ifdef SUPPORT_DEFAULT_PARAMS}
function  GetFileNames(Files: TDbfFileNames  = [dfDbf]  ): string;
   overload;
{$else}
function  GetFileNamesString(Files: TDbfFileNames (* = [dfDbf] *) ):
   string;
{$endif}
```

Provide a TStrings instance, and the function will fill it with the currently opened filenames.

TDbfFileNames specifies what files to report:	dfDbf	The main dbf filename
	dfMemo	The memo filename if any
	dfIndex	Any open index files

6.17 GetIndexNames

```
procedure GetIndexNames(Strings: TStrings);
```

Strings specifies a list which upon return contains a list of all indexes. Entries in this list can be used to set the property IndexName.

6.18 GetAllIndexFiles

```
procedure GetAllIndexFiles(Strings: TStrings);
```

Returns all *.ndx in the directory of the dbf file, ie. which are possible candidates to open with `OpenIndexFile`.

6.19 TryExclusive

```
procedure TryExclusive;
```

Requires Active to be true. Call TryExclusive to try to get exclusive access to the open file, without having to call Close, set Exclusive to true and reopening. The property Exclusive will be properly updated to reflect the new status. Investigate the Exclusive property if this attempt was successful.

6.20 EndExclusive

```
procedure EndExclusive;
```

If you are done operating in exclusive mode, call EndExclusive to return to the previous mode.

6.21 LockTable

```
function  LockTable(const Wait: Boolean): Boolean;
```

Call LockTable to lock the whole table. Wait specifies whether the component should wait to actually get to lock, or fail if the lock cannot be applied.

The difference between LockTable and Exclusive mode is that in Exclusive mode, others cannot open the file anymore, except when ReadOnly is true, but when LockTable is called, they can still open the file in read write mode. When LockTable has succeeded others cannot alter records, because all attempts to lock an individual record will fail.

6.22 UnlockTable

```
procedure UnlockTable;
```

When the table was locked with LockTable, call UnlockTable to unlock the table.

6.23 OpenIndexFile

```
procedure OpenIndexFile(IndexFile: string);
```

Call OpenIndexFile to attach IndexFile, a secondary, non-maintained index file, for example an NDX file, to the DBF file. While the index file is attached, it will be maintained.

6.24 DeleteIndex

```
procedure DeleteIndex(const AIndexName: string);
```

AIndexName specifies an index to remove.

- If this index is contained in the accompanying MDX file, it will be removed there.
- In the case of an NDX file, it will be closed, detached, then removed from disk.

6.25 CloseIndexFile

```
procedure CloseIndexFile(const AIndexName: string);
```

An opened indexfile by OpenIndexFile, or by settings Indexes property, can be closed by calling CloseIndexFile. The particular index will not be maintained any longer.

6.26 RepageIndexFile

```
procedure RepageIndexFile(const AIndexFile: string);
```

When the size of an index file is watched, it can be noticed that the size will not decrease when an index is removed. This functions will “repage” the given index file to attempt to reduce it’s size. Enter an empty string to repage the accompanying MDX file of the table. The effect of RepageIndexFile and recreating all indexes in the index file is about the same, however,

RepageIndexFile will be much quicker. RepageIndexFile can be thought of as a "PackTable" for a given index file. NOTE: you need enough memory for this operation because a temporary index file will be created in memory and then written to disk.

6.27 CompactIndexFile

```
procedure CompactIndexFile(const AIndexFile: string);
```

CompactIndexFile is alike RepageIndexFile, the argument is compatible, and it also tries to reduce the index file size. However it will do this better, but slower, than RepageIndexFile. It compacts the index file so that it uses the minimum amount needed for the index, while usual AddIndex/RepageIndexFile leaves gaps in the index tree. Note that, when you start inserting or modifying keys after calling this function, the index file will slowly start to grow again to usual size.

6.28 Locate

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

Call Locate to search a dataset for a specific record and position the cursor on it.

KeyFields is a string containing a semicolon-delimited list of field names on which to search.

KeyValues is a variant array containing the values to match in the key fields. If KeyFields lists a single field, KeyValues specifies the value for that field on the desired record. To specify multiple search values, pass a variant array as KeyValues.

Options is a set that optionally specifies additional search latitude when searching on string fields. If Options contains the loCaseInsensitive setting, then Locate ignores case when matching fields. If Options contains the loPartialKey setting, then Locate allows partial-string matching on strings in KeyValues. If Options is an empty set, or if the KeyFields property does not include any string fields, Options is ignored.

Locate returns true if it finds a matching record, and makes that record the current one. Otherwise Locate returns false.

If you enter one keyfield to search on, and it matches an open index, then it will use that index to do the search. In that case, Options will be ignored, as if you specified loPartialKey.

6.29 LocateRecord

```
function  LocateRecord(const KeyFields: string; const KeyValues:
   Variant; Options: TLocateOptions; bSyncCursor: Boolean): Boolean;
```

This is an internal method that does the actual work for the Locate functions.

6.30 IsDeleted

```
function  IsDeleted: Boolean;
```

Call IsDeleted to check if the current record is marked as deleted. This can only be true if the property ShowDeleted is true.

6.31 Undelete

```
procedure Undelete;
```

Call Undelete to unmark the current record as deleted.

6.32 CreateTable

```
procedure CreateTable;
```

Call CreateTable at runtime to create a table using this datasets current definitions. If the table already exists, CreateTable overwrites the tables structure and data.

If the FieldDefs property contains values, these values are used to create field definitions. Otherwise the Fields property is used. One or both of these properties must contain values in order to create a database table.

If the Indexes property contain values, these values are used to create indexes on the table.

See also CreateTableEx.

6.33 CreateTableEx

```
procedure CreateTableEx(DbfFieldDefs: TDbfFieldDefs);
```

Call CreateTableEx to create a table using given field definitions. These dbf field definitions give more power, as you can specify precision for numeric fields, for example.

6.34 CopyFrom

```
procedure CopyFrom(DataSet: TDataSet; FileName: string;
   DateTimeAsString: Boolean; Level: Integer);
```

Use this procedure to copy the contents of a given DataSet into a new TDbf table. DataSet is the TDataSet you want to copy from, FileName is the complete (including path and extension) filename of the new table. DateTimeAsString determines whether datetime fields should be converted to string fields in the target table. This is especially useful if you want to use TDbf to create mailing sources for a text processor for example. If this parameter is set True an event OnCopyDateTimeAsString is triggered where you can override the default datetime-to-string conversion which is based on your current local settings. Level determines the TableLevel of the target table.

In order to convert prior TDbf version 6.0 datetime values into a BDE compatible format use this procedure as follows: drop two instances of TDbf on a form, set DateTimeHandling of TDbf1 to dtDateTime and connect it with the existing table. Make sure TDbf2 is set to dtBDETimeStamp and call CopyFrom with DataSet = TDbf1 and DateTimeAsString = False. You can then replace the old table with the new one and use TDbf in dtBDETimeStamp mode in your application.

6.35 RestructureTable

```
procedure RestructureTable(DbfFieldDefs: TDbfFieldDefs; Pack: Boolean)
    ;
```

Call RestructureTable to change the field structure of the current table.

DbfFieldDefs allows you to specify the new structure. Each fielddef contains a property CopyFrom, which is the index of field from which to copy information. Fields with index that are not mentioned in any DbfFieldDef's CopyFrom, will be dropped. If you Assign to copy a table's DbfFieldDefs to a new list for modification, then the CopyFrom property of the new list's fielddefs will be automatically assigned, except for Delphi 3 users. So Delphi 3 users beware, specify CopyFrom property correctly to prevent fields from getting dropped without you wanting it!

Pack specifies whether to pack the table, that is, to remove records marked for deletion.

6.35.1 Example

```
NewFieldDefs: TDbfFieldDefs;
NewFieldDef: TDbfFieldDef;
Dbf1: TDbf;
...
// create new field list
NewFieldDefs := TDbfFieldDefs.Create(Self);
// assign current list
NewFieldDefs.Assign(Dbf1.DbfFieldDefs);
// assume first field is string, 20 wide, make larger to 40
NewFieldDefs.Items[0].Size := 40;
// rename second field to 'RENAMED'
NewFieldDefs.Items[1].FieldName := 'RENAMED';
// add a float field
NewFieldDef := NewFieldDefs.AddFieldDef;
NewFieldDef.FieldName := 'NEW_FLOAT';
NewFieldDef.FieldType := ftFloat;
NewFieldDef.Size := 10;
NewFieldDef.Precision := 3;
// restructure table and pack
Dbf1.Restructure(NewFieldDefs, true);
// restructure table and not pack
//Dbf1.Restructure(NewFieldDefs, false);
// free mem
NewFieldDefs.Free;
```

6.36 PackTable

```
procedure PackTable;
```

Call PackTable to actually remove records marked for deletion. When records are deleted, they are just marked. When PackTable is called, these records will be physically removed. As is, it will call RestructureTable with a nil pointer for DbfFieldDefs, and true for Pack.

6.37 EmptyTable

```
procedure EmptyTable;
```

The EmptyTable method deletes all records from the table. It retains the current field and index structure.

6.38 Zap

```
procedure Zap;
```

An alias for EmptyTable.

6.39 InitFieldDefsFromFields

```
{$ifndef DELPHI_5}
procedure InitFieldDefsFromFields;
{$endif}
```

InitFieldDefsFromFields is an internal method used in various functions, CreateTable for example. It creates field definitions for a given set of Fields. This function is only needed for Delphi 4 and older, because in Delphi 5 and later, this function is implemented in TDataSet.

7 Properties

7.1 AbsolutePath

```
property AbsolutePath: string read FAbsolutePath;
```

The absolute path for the current table. See FilePathFull.

7.2 DbfFieldDefs

```
property DbfFieldDefs: TDbfFieldDefs read GetDbfFieldDefs;
```

DbfFieldDefs lists the field definitions for a dataset, alike the TDataSet.FieldDefs, except that TDbfFieldDefs are better suitable for dbase tables. It includes information for native field types, and precision for numeric fields for example. See TDbfFieldDefs.

To access fields and field values in a dataset, use the Fields and FieldValues properties, and the FieldByName method.

7.3 PhysicalRecNo

```
property PhysicalRecNo: Integer read GetPhysicalRecNo write
   SetPhysicalRecNo;
```

Examine PhysicalRecNo to determine the physical record number for the current record. It can be set to position the cursor on that record. The difference to RecNo is that reading RecNo returns the sequential record number which is the same if no indexes are active, but could be different if there is an index active.

7.4 LanguageID

```
property LanguageID: Integer read GetLanguageID write SetLanguageID;
```

Examine LanguageID to determine the codepage, locale combination the table is using. See Dbf_Lang.pas to decipher the information. Set it before calling CreateTable to specify a codepage/locale combination for a table.

7.5 LanguageStr

```
property LanguageStr: String read GetLanguageStr;
```

Examine LanguageStr to read codepage, locale information for level 7 dbase tables.

7.6 CodePage

```
property CodePage: Cardinal read GetCodePage;
```

Examine CodePage to determine the codepage the dbase data is stored in.

7.7 ExactRecordCount

```
property ExactRecordCount: Integer read GetExactRecordCount;
```

Examine ExactRecordCount to determine the exact number of records in the current dataset. This takes into account deleted, filtered and indexed records. This in contrary to RecordCount, which will always give a rough upper bound estimate. Note that this property needs to scan the complete dataset to find the number of records that are active, while RecordCount is just a simple calculation.

7.8 DbfFile

```
property DbfFile: TDbfFile read FDbfFile;
```

An internally used property to retrieve access to the more lower level access functions. Application users should have no need to use this property.

7.9 UserStream

```
property UserStream: TStream read FUserStream write FUserStream;
```

Use UserStream to specify a memory stream to be used for opening/creating a table. See also the Storage property.

7.10 DisableResyncOnPost

```
property DisableResyncOnPost: Boolean read FDisableResyncOnPost write
   FDisableResyncOnPost;
```

When a record is posted, TDataSet fetches all records in the "neighbourhood" of the current record. The property DisableResyncOnPost controls this behaviour. It can possibly increase speed if you're adding a block of records. See also TDataSet::DisableControls.

7.11 DateTimeHandling

```
property DateTimeHandling: TDateTimeHandling read FDateTimeHandling
   write FDateTimeHandling default dtBDETimeStamp;
```

Prior to version 6.0 TDbf used to store values in '@' (ftDateTime) fields as Delphi type TDateTime. To be compatible with the BDE, however, datetimes need to be stored as BDE type TimeStamp (which is milliseconds elapsed since 01/01/0001 plus one day). To provide backward compatibility you can use this property to determine whether TDbf will read and write datetime values as TDateTime or as BDE TimeStamp. Default now is dtBDETimeStamp but in order to read values in existing TDbf tables you need to choose dtDateTime. If you want to convert your data to be BDE compatible have a look at the new procedure CopyFrom.

7.12 Exclusive

```
property Exclusive: Boolean read FExclusive write FExclusive default
   false;
```

Use `Exclusive` to prevent other applications from accessing a table while it is open in this application. Before opening the table, set Exclusive to true. A table must be closed before changing the Exclusive property.

When `Exclusive` is true, then when the application successfully opens the table, no other application can access it. If the table for which the application has requested exclusive access is already in use by another application, an exception is thrown. To handle such exceptions, wrap the code that opens the table in a try..catch block. See also TryExclusive.

Do not set `Exclusive` to true at design time if you also set the Active property to true at design time. In this case an exception is thrown because the table is already in use by the IDE.

7.13 FilePath

```
property FilePath: string read FRelativePath write SetFilePath;
```

Examine `FilePath` to determine the user set file path of the current table. It can be relative to the current directory or an absolute path. See FilePathFull.

7.14 FilePathFull

```
property FilePathFull: string read FAbsolutePath write SetFilePath
   stored false;
```

Examine `FilePathFull` to determine the absolute path for the current table. It will always read the absolute path, whether a relative or absolute path was given in `FilePath`. Mostly used in the design time IDE, where you can set `FilePath` a relative path, then examine FilePathFull where to file is going to be opened or created.

7.15 Filter

```
property Filter: string read FFilter write SetFilterText;
```

The `Filter` property specifies a condition for a record to be displayed. Records not matching the condition are not displayed, they are skipped. Set the `Filtered` property to true to activate the filter. Note that the `OnFilterRecord` event, see section 7.33, also excludes records from view, namely those for which you set `Accept` to false within the event handler.

7.16 Indexes

```
property Indexes: TDbfIndexDefs read FIndexDefs write SetDbfIndexDefs
   stored false;
```

This property is deprecated in favour of the IndexDefs property, and may be removed in the future.

7.17 IndexDefs

```
property IndexDefs: TDbfIndexDefs read FIndexDefs write
   SetDbfIndexDefs;
```

IndexDefs is a collection of index definitions, each of which describes an available index for the table. Define the index definitions of a table before calling CreateTable or creating a table at design time.

Ordinarily, an application accesses or specifies indexes at runtime through the IndexName and IndexFieldNames properties.

If IndexDefs is updated or manually edited, the StoreDefs property becomes true.

The index definitions in IndexDefs may not always reflect the current indexes available for a table. Before examining IndexDefs, call its Update method to refresh the list.

7.18 IndexFieldNames

```
property IndexFieldNames: string read GetIndexFieldNames write
   SetIndexFieldNames;
```

Use IndexFieldNames as an alternative method of specifying the index to use for a table. In IndexFieldNames, specify the name of each column to use as an index for a table. You can also specify an expression of an existing index. The column name specified in IndexFieldNames must already be indexed.

The IndexFieldNames and IndexName properties are mutually exclusive. Setting one clears the other.

7.19 IndexName

```
property IndexName: string read GetIndexName write SetIndexName;
```

Use IndexName to specify an alternative index for a table. If IndexName is empty, a tables sort order is based on its physical record order.

If IndexName contains a valid index name, then that index determines the sort order of records. The index name supplied to the IndexName property must either reside in the table's master index file, or in another index file already specified in the Indexes property or opened with OpenIndexFile.

IndexFieldNames and IndexName are mutually exclusive. Setting one clears the other.

7.20 MasterFields

```
property MasterFields: string read GetMasterFields write
   SetMasterFields;
```

Use MasterFields after setting the MasterSource property to specify the names of one or more fields to use in establishing a detail-master relationship between this table and the one specified in MasterSource.

MasterFields is a string containing one or more field names in the master table. Separate field names with semicolons.

Each time the current record in the master table changes, the new values in those fields are

used to select corresponding records in this table for display.

7.21 MasterSource

```
property MasterSource: TDataSource read GetDataSource write
   SetDataSource;
```

Use MasterSource to specify the name of the data source component whose DataSet property identifies a dataset to use as a master table in establishing a detail-master relationship between this table and another one. The specified DataSource's DataSet must be another TDbf table.

At design time choose an available data source from the MasterSource propertys drop-down list in the Object Inspector.

After setting the MasterSource property, specify which fields to use in the master table by setting the MasterFields property. At runtime each time the current record in the master table changes, the new values in those fields are used to select corresponding records in this table for display.

7.22 OpenMode

```
property OpenMode: TDbfOpenMode read FOpenMode write FOpenMode default
    omNormal;
```

OpenMode specifies what to do if a table does not exist by this name and Active is set to true, or Open is called.

- omNormal fails the open action if the file does not exist.
- omAutoCreate creates a new table as if CreateTable is called, and opens it.
- omTemporary is not used.

7.23 ReadOnly

```
property ReadOnly: Boolean read FReadOnly write FReadonly default
   false;
```

ReadOnly specifies to open the file in read only mode. If it is true, the table's data can not be altered. You can open a table in read only mode although it is already open in exclusive mode.

7.24 ShowDeleted

```
property ShowDeleted: Boolean read FShowDeleted write SetShowDeleted
   default false;
```

ShowDeleted specifies whether or not to show the records marked for deletion. Use the IsDeleted function to determine if the current record is marked for deletion.

7.25 Storage

```
property Storage: TDbfStorage read FStorage write FStorage default
    stoFile;
```

This property specifies what kind of storage type is used: file or memory. If it is stoFile, you need to specify a FilePath and TableName where to open or create the file. When it is stoMemory, you need to specify a UserStream, which is then used as backend storage.

7.26 StoreDefs

```
property StoreDefs: Boolean read FStoreDefs write FStoreDefs default
    False;
```

If StoreDefs is true, the table's index and field definitions are stored with the data module or form. Setting StoreDefs to true makes the CreateTable method into a one-step procedure that creates fields, indexes, and validity checks at runtime.

StoreDefs is false by default. It becomes true whenever FieldDefs or Indexes is updated or edited manually; to prevent edited (or imported) definitions from being stored, reset StoreDefs to false.

7.27 TableName

```
property TableName: string read FTableName write SetTableName;
```

Use TableName to specify the file name of the database table this component encapsulates. You can specify a full filepath with filename, in which case the filepath part split and stored in the FilePath property.

To set TableName, the Active property must be false.

7.28 TableLevel

```
property TableLevel: Integer read FTableLevel write SetTableLevel;
```

Examine TableLevel to find the current table level. Set TableLevel to specify the table level for to be created tables. The Active property must be false to be able to set TableLevel. These are the possible levels:

- 3: dBase III+ compatible.
- 4: dBase IV compatible. The only difference to dBase III+ is the current codepage, locale. dBase III+ uses no translation for the codepage and a binary sort order for the indexes.

- 7: Visual dBase VII. Not all features are supported, but these give an summary:
  - More field types: datetime, 32 bit integers, 64 bit doubles.
  - Default values for fields. This info can be found via the DbfFieldDef.HasDefault and DefaultBuf properties.
  - Min and Max values for fields are NOT supported, but can be read.
  - Referential integrity is NOT supported.
- 25: FoxPro compatible. The native field types are a bit different, but very comparable to dBase IV. CDX indexes are not supported.

7.29 UseFloatFields

```
property UseFloatFields: Boolean read FUseFloatFields write
    FUseFloatFields default true;
```

When UseFloatFields is enabled, it forces the use of float fields, even though numeric fields have zero precision. When disabled, 32 or 64 bit integer fields will be used, depending on the size of the field.

7.30 Version

```
property Version: string read GetVersion write SetVersion stored false
    ;
```

Examine Version to find the version of the TDbf component.

7.31 BeforeAutoCreate

```
property BeforeAutoCreate: TBeforeAutoCreateEvent read
    FBeforeAutoCreate write FBeforeAutoCreate;
```

When a table does not exist, OpenMode is omAutoCreate and Open is called, this event will be fired. Implement BeforeAutoCreate to prevent the automatic creation of the table.

7.32 OnCompareRecord

```
property OnCompareRecord: TNotifyEvent read FOnCompareRecord write
    FOnCompareRecord;
```

This event is not used.

7.33 OnFilterRecord

```
TFilterRecordEvent = procedure(DataSet: TDataSet; var Accept: Boolean)
    of object;
property OnFilterRecord: TFilterRecordEvent read FOnFilterRecord write
    SetOnFilterRecord;
```

If the `Filtered` property is true, this event is called for each record to determine whether it should be viewed, or should be skipped. If you set the `Accept` parameter to `true`, the record is viewed, if you set it to `false`, the record is skipped. See also the `Filter` property, section 7.15.

7.34 OnLanguageWarning

```
property OnLanguageWarning: TLanguageWarningEvent read
   FOnLanguageWarning write FOnLanguageWarning;
```

Write an OnLanguageWarning event to inhibit the action taken when a table's data is stored in a specific codepage, but this OS can not translate the data for viewing into it's ANSI codepage. You can specify a readonly mode, or edit nonetheless.

7.35 OnLocaleError

```
property OnLocaleError: TDbfLocaleErrorEvent read FOnLocaleError write
    FOnLocaleError;
```

Write an OnLocaleError event to inhibit the action taken when index data is stored in a specific order, but this OS does not have the capability to sort records according to this sort order. You can try to read or even alter the index nonetheless, but the index can be easily corrupted if you do not know exactly what you are doing.

7.36 OnIndexMissing

```
property OnIndexMissing: TDbfIndexMissingEvent read FOnIndexMissing
   write FOnIndexMissing;
```

Write an OnIndexMissing event to inhibit the action taken when a table specifies that it had an index attached, but it is gone. The default is to break the link, but you can refuse to open the table.

7.37 OnCopyDateTimeAsString

```
property OnCopyDateTimeAsString: TConvertFieldEvent read
   FOnCopyDateTimeAsString write FOnCopyDateTimeAsString;
```

Write an OnCopyDateTimeAsString event to provide a custom formatting of DateTime fields into string fields. See the CopyFrom procedure.

7.38 OnTranslate

```
property OnTranslate: TTranslateEvent read FOnTranslate write
   FOnTranslate;
```

Write an OnTranslate event to provide custom translating of the table's data into the OS "ANSI" codepage.