Copyrights and license agreement File:

kbmMemTable and all supportfiles are
copyrighted ©1999-2012 Components4Developers All rights reserved
unless specifically notified differently in the files.

This kbmMemTable help file and document is
copyright ©1999-2012 Components4Developers All rights reserved.

This document has been updated Feb. 28. 2012

You are **REQUIRED** to read this license agreement completely.
The moment you start using the component or documentation, you are agreeing to the complete license agreement.

LICENSE AGREEMENT

PLEASE NOTE THAT THE LICENSE AGREEMENT HAS CHANGED!!! 25. Jul. 2003

You are allowed to use this component in any application for free.
You are NOT allowed to claim that you have created this component or to
copy its code into your own component and claim that it was your idea/code.

IM OFFERING THIS FOR FREE FOR YOUR CONVINIENCE, BUT
YOU ARE REQUIRED TO SEND AN E-MAIL ABOUT WHAT PROJECT THIS COMPONENT
(OR DERIVED VERSIONS) IS USED FOR !

PLEASE NOTE THE FOLLOWING CHANGES TO THE LICENSE AGREEMENT:
1. Aug. 2003
Parts of the package which constitute kbmMemTable Pro is not open source nor freware.
These parts may not be disclosed to any 3rdparty without prior written agreement with
Components4Developers.
Discussion in public fora about the algorithms used in those parts is not permitted.
Each unit which is under that part of the license agreement have it specifically
written in the top of the unit file as part of its own specific license agreement.

25. Jul. 2003
You are allowed to use this component in any application for free.
You are not allowed to use the knowledge gained from this component or any code
from it for creating applications or developer tools directly competing with any
C4D tool unless specifically approved in writing by C4D.
If you choose to make developer tools containing this component you are required to
acknowledge that visually in the application/tool/library f.ex. in the About box,
or in the beginning of the central documentation for the application/tool/library.
The acknowledgement must contain a link or reference to www.components4developers.com
You dont need to state Components4Developers name in your end user application, unless
its covered under the previous phrase, although it would be appreciated if you do.

If you find bugs or alter the component, please DONT just send the corrected/new code out on the internet, but instead send it to me, so I can put it into the official version. You will be acredited if you do so.

DISCLAIMER
By using this component or parts theirof you are accepting the full responsibility of the use. You are understanding that the author cant be made responsible in any way for any problems occuring using this component. You also recognize the author as the creator of this component and agrees not to claim otherwize!

Please forward corrected versions (source code ONLY!), comments, and emails saying you are using it for this or that project to:
kbm@components4developers.com

Support:

- Look for new information at:
  http://www.components4developers.com

- Look for new versions at our portal at:
  https://portal.components4developers.com

- Check the newsgroups at:
  news://news.components4developers.com

EMemTableDupKey File: kbmMemTable

Exception thrown when a record that is being inserted contain field values violating a unique index constraint.

Delphi declaration

```
EMemTableDupKey=Class(EMemTableError);;
```

EMemTableError File: kbmMemTable

The class which will be created and raised (thrown) if an error occurs.
The errormessage itself is dependent on which language ressource have been compiled into kbmMemTable.
Make sure to uncomment only one of these definitions in the sourcecode to select the desired language:

```
{$define KBMMEM_RES_ENGLISH}
//{$define KBMMEM_RES_GERMAN}
//{$define KBMMEM_RES_FRENCH}
//{$define KBMMEM_RES_BRASIL}
//{$define KBMMEM_RES_RUSSIAN}
//{$define KBMMEM_RES_SPANISH}
//{$define KBMMEM_RES_ITALIAN}
//{$define KBMMEM_RES_ROMANIAN}
//{$define KBMMEM_RES_SLOVAKIAN}
//{$define KBMMEM_RES_CZECH}
//{$define KBMMEM_RES_DUTCH}
//{$define KBMMEM_RES_DANISH}
//{$define KBMMEM_RES_HUNGARIAN}
//{$define KBMMEM_RES_UKRAINIAN
```

Delphi declaration

```
EMemTableError=Class(EDataBaseError);;
```

EMemTableFatalError File: kbmMemTable

An exception that is raised if a really serious error has occured.
ne One example is the unique identifier counter. Each record gets a unique identifier, and that is a long value. When kbmMemTable runs out of value space, the exception is raised.

Delphi declaration

```
EMemTableFatalError=Class(EMemTableError);;
```

EMemTableFilterError File: kbmMemTable

This exception is raised if there is a problem with interpreting an expression filter.

Delphi declaration

```
EMemTableFilterError=Class(EMemTableError);;
```

EMemTableIndexError File: kbmMemTable

Currently not used.

Delphi declaration

```
EMemTableIndexError=Class(EMemTableError);;
```

EMemTableInvalidLocale File: kbmMemTable

This exception is raised if the chosen locale settings prevent the CompareString/CompareStringW function to operate.

Delphi declaration

```
EMemTableInvalidLocale=Class(EMemTableLocaleError);;
```

EMemTableInvalidRecord File: kbmMemTable

This exception is thrown when there is something really bad going on within kbmMemTable's record handling.
One example can be if data has been overwritten by other data due to buffer overflows.ne Another example can be if a record contains indirectly referenced string/blob fields, and the original record containing the original reference has gone missing. This will never happen, unless you manipulate kbmMemTables internal structures directly and incorrectly.

Delphi declaration

```
EMemTableInvalidRecord=Class(EMemTableFatalError);;
```

EMemTableLocaleError File: kbmMemTable

Not currently used.

Delphi declaration

```
EMemTableLocaleError=Class(EMemTableError);;
```

Field modifiers File:

When specifying which field or fields should be used in a search or an index, several extra field specific options can be given for each field as desired.

Creating an index on Field1 and Field2 could f.ex. be done like this:nene**AddIndex('iMyIndex','Field1;Field2',[ixDescending,ixCaseInsensitive])**

But this would create an index which is descending and case insensitive on the complete key (both Field1 and Field2).

If one would like to create an index where f.ex. Field1 is descending and Field2 ascending and both fields case insensitive, its possible to specify that using a field modifier.

AddIndex('iMyIndex','Field1:D;Field2', [ixCaseInsensitive])

Since no modifier is given to Field2, and no index option specify otherwise, Field2 will be ascending.

The following field modifiers are available:

C for Case insensitive - Comparisons on the field will not distinguish between upper and lower case.

D for Descending - The field will be sorted in descending order.

P for Partial - The nearest partial match in this field with the field in the search key will be enough for selecting the record during search/locate operation. Not very usefull for index definitions.

N for ignore Nulls - A null value in a search key field will match anything in the specified field.

L for ignore Locale - Comparisons on this field will be made using simple ERROR: INVALID LINK: !Alink(CompareText,1) CompareText /ERROR: INVALID LINK: !Alink(CompareStr,1) CompareStr instead of CompareString (Please see Microsoft documentation - Win32.hlp for further info).

Filter expressions File:

Use Filter to specify a memory table filter expression. When filtering is applied to a dataset, only those records that meet a filter's conditions are available to an application. Filter contains the string that describes the filter condition.
To enable a filter set the Filtered property to true after setting the Filter expression.
Notice that FindFirst, FindNext, FindPrior and FindLast methods respects any filter expression or filter event set, even if the Filtered property is False. Thus using those methods you can search easily between records using a filter expression, but at the same time show all the records.

For example, the following filter condition displays only those records where the State field is 'CA' or 'MA':

State = 'CA' or State = 'MA'

When a filter is set, Blank records do not appear unless explicitly included in the filter. For example:

State <> 'CA' or State = NULL

When a field name contains spaces, you must enclose the field name in brackets. For example:

[Home State] = 'CA' or [Home State] = 'MA'

The FilterOptions property controls case sensitivity and filtering on partial comparisons.

The Filter can be changed at runtime at wish.

The following field types can be part of a filter:

ftSmallInt, **ftWord**, **ftInteger**, **ftAutoInc**, **ftFloat**, **ftCurrency**, **ftString**, **ftFixedChar**, **ftDate**, **ftTime**, **ftDateTime**, **ftBoolean**, **ftBCD**, and any other field which can return a string or a numeric value.

The following operators are available:

Arithmetic:
+ Addition can be used on all numeric and string expressions.
- Subtraction can be used on all numeric expressions.
***** Multiplication can be used on all numeric expressions.
/ Division can be used on all numeric expressions.

Conditions
= Equal
> Greater than
< Less than
>= Greather than or equal
<= Less than or equal
<> Not equal
IS NULL True if expression is null
IS NOT NULL True if expression is not null
NOT Negates the boolean expression
OR True if one of the two boolean expressions are true
AND True if both the two boolean expressions are true
IN (..., ... **)** True if the result of the left side expression is found in the list of values.
LIKE '....**'** True if the left side string expression is matching the wildcards in the right side string.
The wildcards consists of:
***** Matches any number of unknown characters.
? Matches exactly one unknown character.
An example: Fld1 LIKE 'JOHN*'

Functions

UPPER(expression **)** Convert string expression to uppercase.

LOWER(expression **)** Convert string expression to lowercase.

SUBSTRING(expression **,** startpos **,** count **)** Extract a substring from the expression string starting at startpos

and with max length of count chars.

The count parameter

can be omitted, in which case is means

the rest of the string.

TRIM(expression **)** Trim string expression for all leading and trailing spaces.

TRIMLEFT(expression **)** Trim string expression for all leading spaces.

TRIMRIGHT(expression **)** Trim string expression for all trailing spaces.

GETDATE Returns a floating point date/time value for current time.

YEAR(expression **)** Return the year of a floating point date/time value.

MONTH(expression **)** Return the month of the year (1..12) of a floating point date/time value.

DAY(expression **)** Return the day of the month (1..28/29/30/31) of a floating point date/time value.

HOUR(expression **)** Return the hour (0..23) of a floating point date/time value.

MINUTE(expression **)** Return the minute (0..59) of a floating point date/time value.

SECOND(expression **)** Return the second (0..59) of a floating point date/time value.

DATE(datestring **,** formatstring **)** Convert the datestring to a floating point date/time value according to the ERROR: INVALID LINK: !Alink(SysUtils..FormatDateTime,1) format string .

DATE(expression **)** Return the date part of a floating point date/time value.

TIME(timestring **,** formatstring **)** Convert the timestring to a floating point date/time value according to the ERROR: INVALID LINK: !Alink(SysUtils..FormatDateTime,1) format string .

TIME(expression **)** Return the time part of a floating point date/time value.

Public Properties/Variables

COMPONENT_VERSION

Delphi declaration

```
COMPONENT_VERSION='5.00 beta';
```

A constant providing the current version string for the component.

FieldKindNames

Delphi declaration

```
FieldKindNames: [0..4]=('Data', 'Calculated',
'Lookup', 'InternalCalc', 'Aggregate');
```

An array of FieldKind names.

FieldTypeNames

Delphi declaration

```
FieldTypeNames: [TFieldType]=('Unknown', 'String',
'SmallInt', 'Integer', 'Word', 'Boolean', 'Float',
'Currency', 'BCD', 'Date', 'Time', 'DateTime', 'Bytes',
'VarBytes', 'AutoInc', 'Blob', 'Memo', 'Graphic',
'FmtMemo', 'ParadoxOle', 'dBaseOle', 'TypedBinary',
'Cursor');
```

Defines the field type names available in Delphi 3.
For later Delphi/BCB/Kylix editions, this global array is defined in DB.pas.

INFINITE

Delphi declaration

```
INFINITE=LongWord($FFFFFFFF);
```

A value indicating an infinite amount of time.

KBM_MAX_FIELDS

Delphi declaration

```
KBM_MAX_FIELDS=256;
```

This constant sets the maximum number of fields the memory table can handle.
If you need to handle more than 256 fields you need to alter this value.

kbmAutoIndex

Delphi declaration

```
kbmAutoIndex='__MT__AUTO_';
```

This is the prefix used for naming any autogenerated indexes.

kbmBinaryTypes

Delphi declaration

```
kbmBinaryTypes: TkbmFieldTypes=[DB.ftBlob, DB.ftMemo,
DB.ftGraphic, DB.ftFmtMemo, DB.ftParadoxOle,
DB.ftDBaseOle, DB.ftTypedBinary, DB.ftVarBytes,
DB.ftBytes, DB.ftOraBlob, DB.ftOraClob];
```

Lists all the fieldtypes which are considered binary by TkbmMemTable.

See also:
kbmBlobTypes kbmNonBlobTypes kbmStringTypes kbmSupportedFieldTypes kbmVarLengthNonBlobTypes

kbmBlobTypes

Delphi declaration

```
kbmBlobTypes: TkbmFieldTypes=[DB.ftBlob, DB.ftMemo,
DB.ftGraphic, DB.ftFmtMemo, DB.ftParadoxOle,
DB.ftDBaseOle, DB.ftTypedBinary, DB.ftOraBlob,
DB.ftOraClob, DB.ftWideMemo];
```

Lists all the field types which are considered blob types by TkbmMemTable.

See also:
kbmBinaryTypes kbmNonBlobTypes kbmStringTypes kbmSupportedFieldTypes kbmVarLengthNonBlobTypes

kbmBookmarkCurrent

Delphi declaration

```
kbmBookmarkCurrent=$00000000;
```

kbmBookmarkFirst

Delphi declaration

```
kbmBookmarkFirst=$00000001;
```

kbmBookmarkLast

Delphi declaration
```
kbmBookmarkLast=$00000002;
```

kbmDefaultIndex

Delphi declaration
```
kbmDefaultIndex='__MT__DEFAULT_';
```

kbmDefSortIndex

Delphi declaration
```
kbmDefSortIndex='__MT__DEFSORT_';
```

This is the name of the index automatically created when the Sort or SortOn method is called.

See also:
TkbmCustomMemTable.Sort TkbmCustomMemTable.SortOn

kbmDetailIndex

Delphi declaration
```
kbmDetailIndex='__MT__DETAIL_';
```

kbmffCompress

Delphi declaration
```
kbmffCompress=$02;
```

Field flag indicating if the field data should be compressed or not.

See also:

kbmffData

Delphi declaration
```
kbmffData=# 02;
```

Record buffer field flag

If the field contains data (is not null) the field flag must contain this value.

See also:

kbmffIndirect

Delphi declaration
```
kbmffIndirect=$01;
```

If the field data is not held within the record buffer, but is allocated seperately and only a pointer is stored in the record buffer, this flag must be set on the field.

See also:

kbmffModified

Delphi declaration
```
kbmffModified=$04;
```

Field flag that is set the moment a fields value is updated/modified.

See also:

kbmffNull

Delphi declaration
```
kbmffNull=# 00;
```

Record buffer field flag

If the field contains data (is not null) the field flag must contain this value.

See also:

kbmffUnknown

Delphi declaration
```
kbmffUnknown=# 01;
```

Record buffer field flag

If its unknown if the field is containing any data, the field flag must contain this value.

See also:

kbmGetRowsRest

Delphi declaration

```
kbmGetRowsRest=$FFFFFFFF;
```

kbmGroupIndex

Delphi declaration

```
kbmGroupIndex='__MT__GROUP_';
```

kbmkbKey

Delphi declaration

```
kbmkbKey=0;
```

kbmkbMasterDetail

Delphi declaration

```
kbmkbMasterDetail=3;
```

kbmkbMax

Delphi declaration

```
kbmkbMax=3;
```

kbmkbMin

Delphi declaration

```
kbmkbMin=0;
```

kbmkbRangeEnd

Delphi declaration

```
kbmkbRangeEnd=2;
```

kbmkbRangeStart

Delphi declaration

```
kbmkbRangeStart=1;
```

KBMMEMTABLE_EDITION

Delphi declaration

```
KBMMEMTABLE_EDITION='CodeGear';
```

KBMMEMTABLE_VERSION

Delphi declaration

```
KBMMEMTABLE_VERSION='7.14.00';
```

kbmNonBlobTypes

Delphi declaration

```
kbmNonBlobTypes: TkbmFieldTypes=[DB.ftString,
DB.ftSmallint, DB.ftInteger, DB.ftWord, DB.ftBoolean,
DB.ftFloat, DB.ftCurrency, DB.ftDate, DB.ftTime,
DB.ftDateTime, DB.ftAutoInc, DB.ftBCD, DB.ftFmtBCD,
DB.ftTimeStamp, DB.ftBytes, DB.ftVarBytes, DB.ftGUID,
DB.ftWideString, DB.ftFixedChar, DB.ftLargeInt,
DB.ftADT, DB.ftArray, DB.ftFixedWideChar,
DB.ftOraTimeStamp, DB.ftOraInterval, DB.ftLongWord,
DB.ftShortint, DB.ftByte, DB.ftExtended, DB.ftSingle];
```

Lists all the field types which are not considered to be blobs by TkbmMemTable.

See also:

kbmBinaryTypes kbmBlobTypes kbmStringTypes kbmSupportedFieldTypes kbmVarLengthNonBlobTypes

kbmRangeIndex

Delphi declaration

```
kbmRangeIndex='__MT__RANGE_';
```

kbmRecordIdent

Delphi declaration

```
kbmRecordIdent=$6A1B2C3E;
```

This value is used for checking that the basic record buffer has not been messed up because of a invalid memory write.
If kbmMemTable is compiled with the flag DO_CHECKRECORD defined, a check will be made on all record buffers before use.
This is mainly interesting as a debug tool during development of advanced record handling operations within kbmMemTable itself.
Default the overrun checking has been disabled, and no space allocated in the records for the check.

kbmrfDontCheckPoint

Delphi declaration

```
kbmrfDontCheckPoint=$02;
```

Flag indicating if a record should be checkpointed on next checkpoint operation.
If set, the record and its versions wont be checkpointed which means that all versions of the record will be retained.

kbmrfInTable

Delphi declaration

```
kbmrfInTable=$01;
```

Flag indicating if the record structure is part of the table or if its a work record. If set, its part of the table.

kbmRowOrderIndex

Delphi declaration

```
kbmRowOrderIndex='__MT__ROWORDER_';
```

This is the name of the automatically created row order index.
The row order index keeps track of where inserts/deletes are made relatively to eachother when no other index is in use.

kbmSaveAllData

Delphi declaration

kbmSaveAllData: TkbmMemTableSaveFlags=[mtfSaveNonVisible, mtfSaveBlobs, mtfSaveDef, mtfSaveFiltered, mtfSaveIgnoreRange, mtfSaveIgnoreMasterDetail, mtfSaveData];

Standard set of save flags for SaveToStream, SaveToBinaryStream, SaveToFile and SaveToBinaryFile which saves all data and table definitions, but no deltas (versions of records).

See also:
kbmSaveAllDataAndDelta kbmSaveAllDeltas ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryFile TkbmCustomMemTable.SaveToBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryStream TkbmCustomMemTable.SaveToBinaryStream TkbmCustomMemTable.SaveToFile TkbmCustomMemTable.SaveToStream TkbmMemTableSaveFlag TkbmMemTableSaveFlags

kbmSaveAllDataAndDelta

Delphi declaration
kbmSaveAllDataAndDelta: TkbmMemTableSaveFlags=[mtfSaveNonVisible, mtfSaveBlobs, mtfSaveDef, mtfSaveFiltered, mtfSaveIgnoreRange, mtfSaveIgnoreMasterDetail, mtfSaveData, mtfSaveDeltas];

Standard set of save flags for SaveToStream, SaveToBinaryStream, SaveToFile and SaveToBinaryFile which saves all data, table definitions and all deltas (versions of records).

See also:
kbmSaveAllData kbmSaveAllDeltas ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryFile TkbmCustomMemTable.SaveToBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryStream TkbmCustomMemTable.SaveToBinaryStream TkbmCustomMemTable.SaveToFile TkbmCustomMemTable.SaveToStream TkbmMemTableSaveFlag TkbmMemTableSaveFlags

kbmSaveAllDeltas

Delphi declaration
kbmSaveAllDeltas: TkbmMemTableSaveFlags=[mtfSaveNonVisible, mtfSaveBlobs, mtfSaveDef, mtfSaveFiltered, mtfSaveIgnoreRange, mtfSaveIgnoreMasterDetail, mtfSaveDeltas];

Standard set of save flags for SaveToStream, SaveToBinaryStream, SaveToFile and SaveToBinaryFile which saves all deltas (versions of records) and table definitions, but no unmodified data. F.ex. good for passing updates between a client and a server.

See also:
kbmSaveAllData kbmSaveAllDataAndDelta ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryFile TkbmCustomMemTable.SaveToBinaryFile ERROR: INVALID LINK:

kbmMemTable.TkbmCustomMemTable.SaveToBinaryStream TkbmCustomMemTable.SaveToBinaryStream TkbmCustomMemTable.SaveToFile TkbmCustomMemTable.SaveToStream TkbmMemTableSaveFlag TkbmMemTableSaveFlags

KBMSQL_ORDERBY_INDEXNAME

Delphi declaration

```
KBMSQL_ORDERBY_INDEXNAME='__KBMSQL_ORDERBY';
```

KBMSQL_UNIQUE_INDEXNAME

Delphi declaration

```
KBMSQL_UNIQUE_INDEXNAME='__KBMSQL_UNIQUE';
```

kbmSQLDBAPIRegistrations

Delphi declaration

```
kbmSQLDBAPIRegistrations: TkbmSQLDBAPIRegistrations;
```

kbmSQLFunctionRegistrations

Delphi declaration

```
kbmSQLFunctionRegistrations:
TkbmSQLFunctionRegistrations;
```

kbmStringTypes

Delphi declaration

```
kbmStringTypes: TkbmFieldTypes=[DB.ftString,
DB.ftMemo, DB.ftFmtMemo, DB.ftWideString,
DB.ftFixedChar, DB.ftOraClob, DB.ftGuid,
DB.ftFixedWideChar, DB.ftWideMemo, DB.ftOraInterval];
```

Lists the field types which TkbmMemTable consider as string types.

See also:
kbmBinaryTypes kbmBlobTypes kbmNonBlobTypes kbmSupportedFieldTypes kbmVarLengthNonBlobTypes

kbmSupportedFieldTypes

Delphi declaration

```
kbmSupportedFieldTypes: TkbmFieldTypes=[DB.ftString,
DB.ftSmallint, DB.ftInteger, DB.ftWord, DB.ftBoolean,
DB.ftFloat, DB.ftCurrency, DB.ftDate, DB.ftTime,
DB.ftDateTime, DB.ftAutoInc, DB.ftBCD, DB.ftFmtBCD,
DB.ftTimeStamp, DB.ftBlob, DB.ftMemo, DB.ftGraphic,
DB.ftFmtMemo, DB.ftParadoxOle, DB.ftDBaseOle,
DB.ftTypedBinary, DB.ftBytes, DB.ftVarBytes,
DB.ftFixedChar, DB.ftWideString, DB.ftLargeInt,
DB.ftADT, DB.ftArray, DB.ftOraBlob, DB.ftOraClob,
DB.ftGUID, DB.ftFixedWideChar, DB.ftWideMemo,
DB.ftOraTimeStamp, DB.ftOraInterval, DB.ftLongWord,
DB.ftShortint, DB.ftByte, DB.ftExtended];
```

Lists all the fieldtypes supported by TkbmMemTable.

See also:
kbmBinaryTypes kbmBlobTypes kbmNonBlobTypes kbmStringTypes kbmVarLengthNonBlobTypes

kbmVarLengthNonBlobTypes

Delphi declaration

```
kbmVarLengthNonBlobTypes:
TkbmFieldTypes=[DB.ftString, DB.ftBytes,
DB.ftVarBytes, DB.ftWideString, DB.ftFixedChar];
```

Lists the field types which can be compressed but which arent blob types (which all can be compressed).

See also:
kbmBinaryTypes kbmBlobTypes kbmNonBlobTypes kbmStringTypes kbmSupportedFieldTypes kbmVarLengthNonBlobTypes

mtifoCaseInsensitive

Delphi declaration

```
mtifoCaseInsensitive=$02;
```

Index field option for a field participating in an index.ne If specified, that field should be considered case insensitive in index comparisons.

See also:

mtifoDescending

Delphi declaration
mtifoDescending=$01;

Index field option for a field participating in an index.ne If specified, that field should be considered using descending order rather than ascending in index comparisons.

See also:

mtifoIgnoreLocale

Delphi declaration
mtifoIgnoreLocale=$10;

Index field option for a field participating in an index.ne If specified, that field should compare using raw ascii compares rather than using comparisons supporting locales in index comparisons.

See also:

mtifoIgnoreNull

Delphi declaration
mtifoIgnoreNull=$08;

Index field option for a field participating in an index.ne If specified, comparisons with null on that field will be skipped in index comparisons.

See also:

mtifoPartial

Delphi declaration
mtifoPartial=$04;

Index field option for a field participating in an index.ne If specified, that field should allow for partial comparisons in index comparisons.

See also:

NullVarLength

Delphi declaration

```
NullVarLength=PkbmVarLength(0);
```

Public Methods/Functions

AnsiBase64ToString

Delphi declaration

```
function AnsiBase64ToString(Source: AnsiString):
AnsiString;
```

AnsiStringToBase64

Delphi declaration

```
function AnsiStringToBase64(Source: AnsiString):
AnsiString;
```

Base64ToString

Delphi declaration

```
function Base64ToString(Source: ): ;
```

Decodes a Base 64 coded string to a binary string.
Its used internally for loading binary field contents from a text format storage.

See also:
ERROR: INVALID LINK: kbmMemTable.StringToBase64
kbmMemTable.StringToBase64
kbmMemTable.TkbmCustomMemTable.LoadFromFile
kbmMemTable.TkbmCustomMemTable.LoadFromStream
kbmMemTable.TkbmCustomMemTable.SaveToFile
kbmMemTable.TkbmCustomMemTable.SaveToStream

CodedStringToString

Delphi declaration

```
function CodedStringToString(Source: ; EscapeChar:
Char): ;
```

Convert a coded string to a normal text string.
All LF, CR, #0 and % are coded using StringToCodedString.

Its used internally for loading string field contents from a text format storage.

See also:

ERROR: INVALID LINK: kbmMemTable.StringToCodedString
kbmMemTable.StringToCodedString
kbmMemTable.TkbmCustomMemTable.LoadFromFile
kbmMemTable.TkbmCustomMemTable.LoadFromStream
kbmMemTable.TkbmCustomMemTable.SaveToFile
kbmMemTable.TkbmCustomMemTable.SaveToStream

CompareFields

Delphi declaration

```
function CompareFields(KeyField, AField: pointer;
FieldType: TFieldType; LocaleID: TkbmLocaleID;
IndexFieldOptions: TkbmifoOptions; FullCompare:
boolean): Integer;
```

Is used internally to compare two fieldbuffers with each other.
If you dont write your own OnCompareFields event handler, this function will be called by default.
If you choose to write an eventhandler, you can make your own special rules for how to sort a specific field (or all fields). For the fields you dont want to handle, you can simply call CompareFields with the parameters provided by the OnCompareFields handler.

Notice that KeyField and AField is a pointer to the internal representation of the field contents.

Returns: <0 if KeyField less than AField
=0 if KeyField equals AField
>0 if KeyField greather than AField

See also:

TkbmCustomMemTable.OnCompareFields

CompareOptions2IndexOptions

Delphi declaration

```
function CompareOptions2IndexOptions(AOptions:
TkbmMemTableCompareOptions): TIndexOptions;
```

Converts `TkbmMemTableCompareOptions to TIndexOptions`

CopyFieldDefs

Delphi declaration

```
procedure CopyFieldDefs(Source: TFieldDefs; Dest:
TFieldDefs);
```

IndexOptions2CompareOptions

Delphi declaration

```
function IndexOptions2CompareOptions(AOptions:
TIndexOptions): TkbmMemTableCompareOptions;
```

Converts `TIndexOptions to TkbmMemTableCompareOptions`

kbmMemMove

Delphi declaration

```
procedure kbmMemMove(Source; Dest; Count: Integer);
```

MergeDataTypes

Delphi declaration

```
function MergeDataTypes(ALeft: TFieldType; ARight:
TFieldType): TFieldType;
```

StreamToVariant

Delphi declaration

```
function StreamToVariant(stream: TStream): variant;
```

Convert the contents of a stream to a variant array.

See also:
VariantToStream

StringToBase64

Delphi declaration

```
function StringToBase64(Source: ): ;
```

Encodes a binary string to Base 64 format which means the binary data can be represented as normal text.
Its used internally for saving binary field contents to a text format storage.

See also:
ERROR: INVALID LINK: kbmMemTable.Base64ToString
kbmMemTable.Base64ToString
kbmMemTable.TkbmCustomMemTable.LoadFromFile
kbmMemTable.TkbmCustomMemTable.LoadFromStream
kbmMemTable.TkbmCustomMemTable.SaveToFile
kbmMemTable.TkbmCustomMemTable.SaveToStream

StringToCodedString

Delphi declaration

```
function StringToCodedString(Source: ; EscapeChar:
Char): ;
```

Convert a normal text string to a coded string.
All LF, CR, #0 and % are coded to a text representation.
Its used internally for saving string field contents to a text format storage.

See also:
ERROR: INVALID LINK: kbmMemTable.CodedStringToString
kbmMemTable.CodedStringToString
kbmMemTable.TkbmCustomMemTable.LoadFromFile
kbmMemTable.TkbmCustomMemTable.LoadFromStream
kbmMemTable.TkbmCustomMemTable.SaveToFile
kbmMemTable.TkbmCustomMemTable.SaveToStream

VariantToStream

Delphi declaration

```
procedure VariantToStream(AVariant: variant; stream:
TStream);
```

Convert a variant array to a stream.
The stream must allready exist, and its contents will be overwritten.

See also:
StreamToVariant

KBM_MAX_FIELDS File: kbmMemTable

This event is called before every data load operation including LoadFromStream, LoadFromFile, LoadFromBinaryStream, LoadFromBinaryFile and LoadFromDataSet.

In all cases except LoadFromDataSet, the *Stream* parameter is the stream from which the load is going to happen.
For LoadFromDataSet, the *Stream* parameter will be nil.
StorageType gives you a hint about which of the load methods invoked this event.

Delphi declaration
TkbmOnLoad=procedure(DataSet: TDataSet; StorageType: TkbmMemTableStorageType; Stream: TStream) of object;

See Also:
TkbmCustomMemTable.LoadFromBinaryFile TkbmCustomMemTable.LoadFromBinaryStream TkbmCustomMemTable.LoadFromDataSet TkbmCustomMemTable.LoadFromFile TkbmCustomMemTable.LoadFromStream TkbmOnSave

kbmDefSortIndex File: kbmMemTable

The name of the index automatically created by kbmMemTable when the Sort or SortOn method is used.

Delphi declaration
kbmDefSortIndex='__MT__DEFSORT';

See Also:
TkbmCustomMemTable.Sort TkbmCustomMemTable.SortOn

kbmRowOrderIndex File: kbmMemTable

The name of the standard and always present row order index.
The row order index is used to maintain the order of insertion of records if no other index is selected.

Delphi declaration
kbmRowOrderIndex='__MT__ROWORDER';

See Also:
TkbmCustomMemTable.IndexFieldNames TkbmCustomMemTable.IndexName

PBookmarkFlag File: kbmMemTypes

Pointer to a bookmark flag.
Since the base table of attached tables need to maintain more than one bookmark (one for each attached table), the pointer type is needed.

Delphi declaration

```
PBookmarkFlag=^TBookmarkFlag;
```

See Also:
kbmMemTable.PkbmBookmark

PDateTimeRec File: kbmMemTypes

Pointer to a TDateTimeRec.

Delphi declaration

```
PDateTimeRec=^TDateTimeRec;
```

Performance enhancements File:

Several methods exist for performance improvements using kbmMemTable.
Here I will list some of the most important ones.

Batch load of 100s, 1000s or millions of records.

Normally when you insert a record using Append, Set fields, Post, all indexes will be updated on the fly. This is a relatively high cost, since several search operations are performed to figure out where each record should be inserted into each of the defined indexes. A better aproach is to disable all indexes duing the batch load, reenable them after, and finally make sure all indexes are updated at the end. Further you also should remember to disable updates of all controls during the batch load. Eg.

```
mt.DisableControls;
mt.EnableIndexes:=false;
ds.Dataset:=nil;    // This also improves speed.
try
   for i:=0 to 10000 do
   begin
      ... // Append a record.
   end;
finally
   mt.EnableIndexes:=true;
   mt.UpdateIndexes;
   ds.Dataset:=mt;  // Otherwise nothing will be shown in dataaware controls.
   mt.EnableControls;
end;
```

Its very important to remember to call UpdateIndexes because until its called, no application or user will every see the newly appended records.

Estimated performance improvement, approx. 700%

Further small improvement can be gained by setting the Capacity property to the approx. number of records the table will hold just before the batch load operation.

Performance improvement using field objects instead of field names.

Whenever many field updates are performed, its, from a performance view, a better choise to use the Fields list instead of using the FieldByName method. An even better choise is to use field objects directly. Eg.

#1: Slowest:

```
mt.FieldByName('SomeFieldName').AsString := 'abc';
```

#2: Faster if you know that SomeFieldName is field number 2:

```
mt.Fields[2].AsString:='abc';
```

#3: Even faster if you have many updates of specific fields:

```
var
   fld:TStringField;
begin
   fld:=mt.FieldByName('SomeFieldName'); // or fld:=mt.Fields[2];
```

```
   for x:=1 to 10000 do
   begin
      ...
      fld.AsString:='abc';
      ...
   end;
end;
```

Estimated performance improvement compared to #1: #2 approx. 50%, #3 approx 100-200% maybe even more.

The method AppendRecord is another alternative. Its field access is not the fastest for many inserts, but it does skip a few steps in the actual append of the record.

Applying a filter to a kbmMemTable.

When kbmMemTable is used in Delphi 5 or BCB 5, its possible to write filters similar to the WHERE clause of a SQL SELECT statement. This is very smart in cases where a user wants to alter the filter criterias and the criterias are more complex than a simple start and end value. But it is also a relatively slow operation, since every record must be matched against the filter criterias at every fetch and count of the records. There are several options of how to apply filters.

#1: Filter using Filter property (D5/BCB5 only):

mt.Filtered:=true;

mt.Filter:='([AField]>5) and ([AField]<10)';

#2: Filter using OnFilterRecord (this works on D3, D4, BCB3 and BCB4 too). Write an OnFilterRecord eventhandler:

```
mt.OnFilterRecord:=kbmMemTable1FilterRecord;
mt.Filtered:=true;

procedure TForm1.kbmMemTable1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
var
   v:integer;
begin
   v:=DataSet.FieldByName('AField').AsInteger;
   Accept:=(v>5) and (v<10);
end;
```

#3: Filter using filtered indexes. Write an OnFilterIndex event handler:

Add an index to the table, f.ex. during designtime.

```
mt.OnFilterIndex:=kbmMemTable1FilterIndex;

procedure TForm1.kbmMemTable1FilterIndex(DataSet: TDataSet;  Index: TkbmIndex; var Accept:
Boolean);
var
   v:integer;
begin
   // Only filter on my special filtered index.
   if Index=myFilteredIndex then
   begin
      v:=DataSet.FieldByName('AField').AsInteger;
```

```
      Accept:=(v>5) and (v<10);
   end
   else Accept:=true;
end;
```

#4: Filter using filtered indexes. Create an index with a filter expression.

```
mt.AddFilteredIndex('IndexName','AField1:D;AField2',[],'([AField1]>5) and
([AField1]<10)',[]);
```

Creates an index on AField1 (descending) and AField2 but only includes the records where AField1>5 and AField1<10.

The purpose of making filtered indexes is that during searching, scrolling, browsing and record counting, theres no need to check if a record is valid according to the filter or not. Since it has been accepted during insertion/edit, it is considered valid later on during browse etc. F.ex. getting RecordCount on a filtered index is alot faster than getting RecordCount on a table where the filter is set the 'normal' way, since no recount is needed. Further grids vertical scrollbar will bahave nicely, since the memtable always can report the correct total number of records available for the grid.

But since the records are only validated during insertion and edit, changing the filter on the fly requires a bit more attention. If the filter criterias are changed, the index must be rebuild usingnemt.Indexes.Rebuild('myFilteredIndexName')neafter changing the filter rules, to make sure only the records matching the new filter is included in the index.

Compared to #1 the performance will be affected as follows:

#2: Browse performance is increased by approx 100%, Insert/edit performance is not affected. DBGrids will not be able to update the vertical scrollbar precisely.

#3: Browse performance is increased by approx 500%, Insert/edit performance is decreased by 75%. DBGrid vertical scrollbar is displayed precisely.

#4: Browse performance is increased by approx 500%, Insert/edit performance is decreased by 50%. DBGrid vertical scrollbar is displayed precisely.

Single thread operation:

If you are sure only one thread is accessing the table at the time, you can disable all thread protection code by setting the property Standalone:=true. Performance improvement is around 5-10%.

Datastorage and memory consumption:

Another important issue is memory consumption. Paging and swapping in Windows is a very costfull operation which will slow down your application to almost a halt for many seconds. Thus sometimes its better to sacrefice a little bit speed for a more compact memory handling. kbmMemTable gives you several options:

#1: Set Perfomance to mtpfFast

This makes sure no compression or memory optimizations are done on the records when they are stored in memory. On relatively small tables it gives a performance increase compared to the other performance flags. Thus a record is always taking up maximum record space. Eg. if a record contains a ftString field of size 30, the record will always take up the full 30 bytes + overhead, regardless if the data in the field is only 5 chars long, or perhaps the field is even NULL.

#2: Set Performance to mtpfBalanced

This instructs kbmMemTable to optimize record allocation in such way that NULL string fields only takes up 4 bytes regardless of the defined size of the field. Compared to the previous example, the memory usage will actually be a bit higher instead of smaller, since string fields with data in them will take up the defined size of the field + 4 bytes. Thus this performance setting should primarely be used when there are few string fields containing data, and many containing NULL. Further there is a small CPU wise performance penalty because the string field data has to be set and fetched using another indirection.

#3: Set Performance to mtpfSmall

This instructs kbmMemTable to work like with #2, but also introduces field compression. Default it will only store the data actually in the string fields plus 5 byte extra per field. Thus in the previous example only 11 bytes are allocated instead of 30. You can write your own rules for in memory compression using the OnCompressField and OnDecompressField.

#4: Compression of blobs

If you are using blobs, compression and decompression of blobs on the fly can save large amounts of memory, but with a CPU wise performance penalty depending on the method of compression. Use the OnCompressBlob and OnDecompressBlob events if you want to implement blob compression/decompression.

Searching:

If you have relatively large amounts of records (>100) you might want to benefit from adding an index containing the fields on which you want to search/locate. If no index is defined which match the fields you are using during a search operation, the search will be performed sequentially, which means that all accessible records will be checked, one by one.

Searching on an index will make the search many times faster. If you f.ex. want to search for a specific record in a memtable containing 10000 records using Locate or Lookup, and no index is matching the fields you are searching on, the search will in worst case walk 10000 records. Using an index, only 10 records will have to be checked in worst case.

Other improvements:

#1: The RecalcOnFetch property is default set to true to make kbmMemTable similar to how TTable and many other TDataset descendants operates. The standard set by Borland actually requires that all calculated fields are recalculated every time a record is fetched from the memory table. This is in most cases really not needed. Instead set it to false, to make sure the recalculation is only performed when fields in the record are altered, and thus there is a potential for the calculated field to change. Performance improvement during browsing approx 10%.

#2: RecalcOnIndex determines if the calculated fields should be updated before all comparison operations during search etc. If the data in the table is known not to change, you can safely set RecalcOnIndex to false (default true). This will during sequential seaches gain another 10-15% performance.

#3:If you dont need versioning, dont enable it. Changing record data will, if you have versioning enabled, create another copy of the record (depending on the VersioningMode property), and store that version together with the new one in memory. There is no reason to do that if you dont need versioning. Performance gain on updates/deletes approx. 15%.

#4: Its not enough to use DisableControls/EnableControls to avoid data aware controls to use some CPU. To disconnect them fully, make sure that any TDataSource's Dataset pointing to the memory table are set to nil before lengthy batch operations, and then reset afterwards.

See Also:

Filter expressions kbmMemTable.TkbmCustomMemTable.Capacity kbmMemTable.TkbmCustomMemTable.EnableIndexes kbmMemTable.TkbmCustomMemTable.OnCompressField kbmMemTable.TkbmCustomMemTable.OnDecompressField kbmMemTable.TkbmCustomMemTable.OnFilterIndex kbmMemTable.TkbmCustomMemTable.Performance kbmMemTable.TkbmCustomMemTable.Standalone

PkbmAllocRecordBuffer File: kbmMemTypes

Delphi declaration

```
PkbmAllocRecordBuffer=PChar;
```

PkbmBookmark File: kbmMemTable

Pointer to a bookmark.
Since the base table of attached tables need to maintain more than one bookmark (one for each attached table), the pointer type is needed.

Delphi declaration
```
PkbmBookmark=^TkbmBookmark;
```

See Also:
PBookmarkFlag

PkbmCalculateFields File: kbmMemTypes

Delphi declaration

```
PkbmCalculateFields=PChar;
```

PkbmClearCalcFields File: kbmMemTypes

Delphi declaration

```
PkbmClearCalcFields=PChar;
```

PkbmFreeRecordBuffer File: kbmMemTypes

Delphi declaration

```
PkbmFreeRecordBuffer=PChar;
```

PkbmGetBookmarkData File: kbmMemTypes

Delphi declaration

```
PkbmGetBookmarkData=PChar;
```

PkbmGetBookmarkFlag File: kbmMemTypes

Delphi declaration

```
PkbmGetBookmarkFlag=PChar;
```

PkbmGetCalcFields File: kbmMemTypes

Delphi declaration

```
PkbmGetCalcFields=PChar;
```

PkbmGetFieldData File: kbmMemTypes

Delphi declaration

```
PkbmGetFieldData=Pointer;
```

PkbmGetRecord File: kbmMemTypes

Delphi declaration

```
PkbmGetRecord=PChar;
```

PkbmInternalAddRecord File: kbmMemTypes

Delphi declaration

```
PkbmInternalAddRecord=Pointer;
```

PkbmInternalBookmarkValid File: kbmMemTypes

Delphi declaration

```
PkbmInternalBookmarkValid=Pointer;
```

PkbmInternalGotoBookmark File: kbmMemTypes

Delphi declaration

```
PkbmInternalGotoBookmark=Pointer;
```

PkbmInternalInitRecord File: kbmMemTypes

Delphi declaration

```
PkbmInternalInitRecord=PChar;
```

PkbmInternalSetToRecord File: kbmMemTypes

Delphi declaration

```
PkbmInternalSetToRecord=PChar;
```

PkbmRecord File: kbmMemTypes

Pointer to a record structure.
It is used many places for casting TDataset pointers to a actual record structure.

Delphi declaration

```
PkbmRecord=^TkbmRecord;
```

PkbmRecordBuffer File: kbmMemTypes

Delphi declaration

```
PkbmRecordBuffer=PChar;
```

PkbmSetBookmarkData File: kbmMemTypes

Delphi declaration

```
PkbmSetBookmarkData=PChar;
```

PkbmSetBookmarkFlag File: kbmMemTypes

Delphi declaration

```
PkbmSetBookmarkFlag=PChar;
```

PkbmSetFieldData File: kbmMemTypes

Delphi declaration

```
PkbmSetFieldData=Pointer;
```

PkbmUserBookmark File: kbmMemTable

Pointer to a `TkbmUserBookmark`

Delphi declaration

```
PkbmUserBookmark=^TkbmUserBookmark;
```

PkbmVarLength File: kbmMemTable

Pointer to a variable length field.
Its used extensively by blobs, long strings, binary fields and all other fields which has the **kbmffIndirect** field flag set.

Delphi declaration

```
PkbmVarLength=PAnsiChar;
```

See Also:
PPkbmVarLength

PPkbmRecord File: kbmMemTable

Pointer to a pointer to a record structure.
Mainly used for bookmark casting since a bookmark internally is pointer to a pointer of a real record.

Delphi declaration
PPkbmRecord=^PkbmRecord;

PPkbmVarLength File: kbmMemTable

Pointer to a pointer to a variable length record. Its used for internal arithmetic calculations to get to the right variable length field.

Delphi declaration

```
PPkbmVarLength=^PkbmVarLength;
```

See Also:

PkbmVarLength

PPWideChar File: kbmMemTable

Pointer to a pointer to a WideChar

Delphi declaration

```
PPWideChar=^PWideChar;
```

PSQLTimeStamp File: kbmMemTable

Delphi declaration

```
PSQLTimeStamp=^TSQLTimeStamp;
```

PWordBool File: kbmMemTypes

Pointer to a word size (16 bit) boolean value.
Actually Delphi/BCB has own definitions of pointers to WordBool, but they are not made public.
Boolean fields are allways stored as a WordBool type in the record.

Delphi declaration

```
PWordBool=^WordBool;
```

RecordCount File:

Return the number of records in the table taking into account any filters, filtered indexes, ranges, master/detail relations etc.
Notice that if any filter, range or master/detail relation is setup, RecordCount will have to scan all records in the current index to check if they are filtered out or not.
If RecordCount will be used extensively on large tables, replace ordinary filters with filtered indexes .

See Also:
TDataset.RecordCount

TkbmBinaryStreamFormat File: kbmMemBinaryStreamFormat

TkbmBinaryStreamFormat is one of the record streaming classes bundled with kbmMemTable. Its used for storing/loading kbmMemTable data in a stream in a proprietary binary format.

Delphi declaration

```
TkbmBinaryStreamFormat=Class(TkbmCustomBinaryStreamFormat);
```

Public Events

TkbmBlobStream File: kbmMemTable

Class which aids ERROR: INVALID LINK: !Alink(TDataset,1) TDataset in interacting with blobs.
Do not use directly.
Instead use the LoadFromStream/File and SaveToStream/File methods of the blob field.

Delphi declaration

```
TkbmBlobStream=Class(TMemoryStream);
```

Public Methods/Functions

Create

Delphi declaration

```
Create(Field: TBlobField; Mode: TBlobStreamMode);
```

Create a stream access to the blob field given the specified access mode.
Do not use directly.

Destroy

Delphi declaration

```
Destroy; ;
```

Updates the blob field if the access mode was originally for write or read/write and the Write method has been called.
Then the blob field access object will be deallocated.
Do not use directly.

Truncate

Delphi declaration

```
Truncate;
```

Deallocate any existing blob field value and set the blob field to null.
Do not use directly.

Write

Delphi declaration

```
(Buffer; Count: Longint): Longint; ;
```

Write the specified number of bytes from the buffer into a blob field.
Do not use directly.

TkbmBookmark File: kbmMemTable

Record which contains a bookmark representation.

Delphi declaration
`TkbmBookmark=`

Public Properties/Variables

Bookmark

Delphi declaration
```
Bookmark: PkbmRecord;
```

A bookmark actually is a pointer to a physical record allocation from which the actual record number can be deducted through some clever algorithms.

Flag

Delphi declaration
```
Flag: TBookmarkFlag;
```

The bookmark flag actually has not much to do with bookmarks, but is a very volatile and temporary storage for some ERROR: INVALID LINK: !Alink(TDataset,1) TDataset flags which is used for internal use only. Do not mess with it.

RecordID

Delphi declaration
```
RecordID: TkbmNativeInt;
```

TkbmCommon File: kbmMemTable

Class which handles all common operations of multiple TkbmCustomMemTable 's.
TkbmCommon makes sure to synchronize all operations when attached tables are used, and its also the sole holder of all record and transaction information. TkbmCustomMemTable doesnt hold record information, but only table and index definition.
You would normally not need to access any of the methods or properties of this class directly.

Delphi declaration
```
TkbmCommon=Class
```

See Also:
TkbmCustomMemTable

Public Properties/Variables

AttachCount

Delphi declaration
```
AttachCount: integer GetAttachCount;
```

Return the number of tables attached to this TkbmCommon.

See also:
AttachTable DeAttachTable

AttachMaxCount

Delphi declaration
```
AttachMaxCount: integer GetAttachMaxCount
SetAttachMaxCount;
```

Specifies the maximum number of tables that can be attached to this TkbmCommon without the need of closing TkbmCommon and thus all attached tables.
Beware that each record will take up approx. 8 bytes pr. each extra attached table.
Thus if you allocate room for f.ex. 10 attached records, 80 bytes/record will be allocated + the space needed for fieldcontents and other structures.

See also:
AttachCount AttachTable DeAttachTable

AutoIncMax

Delphi declaration
```
AutoIncMax: TkbmNativeInt GetAutoIncMax SetAutoIncMax;
```

The largest value of the auto inc field in the table.

See also:
AutoIncMin

AutoIncMin

Delphi declaration
```
AutoIncMin: TkbmNativeInt GetAutoIncMin SetAutoIncMin;
```

The minimum value allowed for the autoinc field in the table. It can be set to start the autoinc value at a specific point.

See also:
AutoIncMax

Capacity

Delphi declaration
```
Capacity: longint GetCapacity SetCapacity;
```

Returns or sets the current allocated number of entries in the record list. It will typically be larger than RecordCount, since allocations are made in chunks of performance reasons.

DataRecordSize

Delphi declaration
```
DataRecordSize: longint FDataRecordSize;
```

Returns the calculated size of the raw record, only counting the size of a pointer for varlength fields .

DeletedCount

Delphi declaration
```
DeletedCount: LongInt FDeletedCount SetDeletedCount;
```

Returns the number of deleted records in the deleted records list.

EnableVersioning

Delphi declaration
```
EnableVersioning: boolean GetEnableVersioning
SetEnableVersioning;
```

Return true if versioning is enabled, else false.
Set to true to remember old versions of a record when the record is modified or deleted.
Transactions and deltahandlers need EnableVersions set to true.

Check also VersioningMode.

See also:
VersioningMode TkbmCustomDeltaHandler

FieldModified

Delphi declaration
```
FieldModified[i: integer]: boolean GetModifiedFlag
SetModifiedFlag;
```

Returns true if the specific field of the current record has been modified.

FLock

Delphi declaration
```
FLock: TkbmCS;
```

FRecordID

Delphi declaration
```
FRecordID: TkbmNativeInt;
```

FUniqueRecordID

Delphi declaration
```
FUniqueRecordID: TkbmNativeInt;
```

IsDataModified

Delphi declaration
```
IsDataModified: boolean GetIsDataModified
SetIsDataModified;
```

Returns true if any record has been modified.

LanguageID

Delphi declaration
```
LanguageID: integer GetLanguageID SetLanguageID;
```

Specifies the LanguageID which together with SubLanguageID and SortID will result in a LocaleID that will be used during comparison of string fields. This affects f.ex. sorting etc.

See also:
LocaleID SortID SubLanguageID ERROR: INVALID LINK: TLanguages TLanguages

LocaleID

Delphi declaration
```
LocaleID: TkbmLocaleID GetLocaleID SetLocaleID;
```

Specifies the LocaleID which will be used during comparison of string fields. This affects f.ex. sorting etc.
Changing LocaleID also changes LanguageID, SortID and SubLanguageID.

See also:
LanguageID SortID SubLanguageID ERROR: INVALID LINK: TLanguages TLanguages

Performance

Delphi declaration
```
Performance: TkbmPerformance GetPerformance
SetPerformance;
```

This property controls how records should be stored in memory.
Chosing the right Performance setting can result in drastic decrease of memory usage or a performance improvement.

RecordID

Delphi declaration
```
RecordID: TkbmNativeInt FRecordID SetRecordID;
```

Records

Delphi declaration
```
Records: TkbmList FRecords;
```

Returns the internal record list.
All indexes reference records in this list only.

SortID

Delphi declaration
```
SortID: integer GetSortID SetSortID;
```

Specifies the SortID which together with SubLanguageID and LanguageID will result in a LocaleID that will be used during comparison of string fields. This affects f.ex. sorting etc.

See also:
LanguageID LocaleID SubLanguageID ERROR: INVALID LINK: TLanguages TLanguages

Standalone

Delphi declaration
```
Standalone: boolean GetStandalone SetStandalone;
```

Setting Standalone to true will disable internal locking mechanisms. Set it to true only if you are sure no other tables will attach to this (AttachedTo) and you want maximum performance.
Default False.

See also:
TkbmCustomMemTable.AttachedTo

SubLanguageID

Delphi declaration
```
SubLanguageID: integer GetSubLanguageID
SetSubLanguageID;
```

Specifies the SubLanguageID which together with LanguageID and SortID will result in a LocaleID that will be used during comparison of string fields. This affects f.ex. sorting etc.

See also:
LanguageID LocaleID SortID ERROR: INVALID LINK: TLanguages TLanguages

TransactionLevel

Delphi declaration
```
TransactionLevel: integer GetTransactionLevel;
```

Return the current transaction level.
Every TkbmCustomMemTable.StartTransaction will raise the transaction level by one, and every TkbmCustomMemTable.Commit or TkbmCustomMemTable.Rollback will lower it by one.

See also:
DecTransactionLevel IncTransactionLevel TkbmCustomMemTable.Commit TkbmCustomMemTable.Rollback TkbmCustomMemTable.StartTransaction

UniqueRecordID

Delphi declaration

```
UniqueRecordID: TkbmNativeInt FUniqueRecordID SetUniqueRecordID;
```

Provides access to the next unique record id that will be used when inserting/adding records.

VersioningMode

Delphi declaration

```
VersioningMode: TkbmVersioningMode GetVersioningMode SetVersioningMode;
```

Specifies how record versioning should work.
Its used together with EnableVersioning to backup records upon modification or deletion.
If you want to use Transactions, you should set VersioningMode to mtvmAllSinceCheckPoint.
If you want the table to produce information for use in a deltahandler, you can choose either mtvmAllSinceCheckPoint or mtvm1SinceCheckPoint.

See also:
EnableVersioning TkbmCustomMemTable.CheckPoint TkbmCustomMemTable.Commit TkbmCustomMemTable.Rollback TkbmCustomMemTable.StartTransaction

Public Methods/Functions

_InternalAllocRecord

Delphi declaration

```
_InternalAllocRecord: PkbmRecord;
```

Internal function which allocates a record buffer.
The record buffer is provided as result.
All allocated record buffers must be deallocated using _InternalFreeRecord.

See also:
TkbmCommon._InternalFreeRecord

_InternalFreeRecord

Delphi declaration

```
_InternalFreeRecord(ARecord: PkbmRecord;
FreeVarLengths, FreeVersions: boolean);
```

Free a record buffer previously allocated by either _InternalAllocRecord or _InternalCopyRecord.
FreeVarLengths control if all varlength fields (like blobs and optionally strings) should be free'ed.
FreeVersions control if all record versions linked to this record should also be free'ed.
The contents of ARecord is unknown after this call.

See also:
TkbmCommon._InternalAllocRecord

AppendRecord

Delphi declaration

```
AppendRecord(ARecord: PkbmRecord);
```

Append one recordbuffer to the list of records.

AttachTable

Delphi declaration

```
AttachTable(ATable: TkbmCustomMemTable);
```

Attach another table to this TkbmCommon.

See also:
AttachCount DeAttachTable

ClearIndexes

Delphi declaration

```
ClearIndexes;
```

Clear all indexes of all attached tables.
The indexes are not removed, but only cleared.

CloseTables

Delphi declaration

```
CloseTables(Caller: TkbmCustomMemTable);
```

Close all tables attached to this TkbmCommon.

Commit

Delphi declaration

```
Commit;
```

Commits all changes made since the matching start of transaction. Transactions can be nested.

CompressFieldBuffer

Delphi declaration

```
CompressFieldBuffer(Field: TField; Buffer: pointer;
Size: longint): pointer;
```

Compresses the field buffer according to some simple rules. ftString and ftFixedChar fields are truncated to the actual length of the string ie. all field padding is removed.
You can choose to write your own rules for compression of fields, utilizing f.ex. RLE or LZH or some other compression of choise by writing an OnCompressField and OnDecompressField event handler for TkbmCustomMemTable.

See also:

TkbmCustomMemTable.OnCompressField TkbmCustomMemTable.OnDecompressField

Create

Delphi declaration

```
Create(AOwner: TkbmCustomMemTable);
```

Constructor of the TkbmCommon class.

DeAttachAllTables

Delphi declaration

```
DeAttachAllTables(AExceptTable: TkbmCustomMemTable);
```

DeAttachTable

Delphi declaration

```
DeAttachTable(ATable: TkbmCustomMemTable);
```

De attach a table from this TkbmCommon.

See also:
AttachCount AttachTable

DecompressFieldBuffer

Delphi declaration

```
DecompressFieldBuffer(Field: TField; Buffer: pointer;
Size: longint): pointer;
```

Decompress the field buffer according to some simple rules.
You can choose to write your own rules for decompression of fields, utilizing f.ex. RLE or LZH or some other decompression of choise by writing an OnCompressField and OnDecompressField event handler for TkbmCustomMemTable.

See also:
CompressFieldBuffer TkbmCustomMemTable.OnCompressField TkbmCustomMemTable.OnDecompressField

DecTransactionLevel

Delphi declaration

```
DecTransactionLevel;
```

Lower the transaction level by one.
Used by TkbmCustomMemTable.Commit and TkbmCustomMemTable.Rollback.

See also:
IncTransactionLevel TkbmCustomMemTable.Commit TkbmCustomMemTable.Rollback TkbmCustomMemTable.StartTransaction TkbmCustomMemTable.TransactionLevel

DeletedRecordCount

Delphi declaration

```
DeletedRecordCount: TkbmNativeInt;
```

Return the number of deleted records in the deleted records list.

See also:
PackRecords TkbmCustomMemTable.EmptyTable
TkbmCustomMemTable.PackTable

DeleteRecord

Delphi declaration
```
DeleteRecord(ARecord: PkbmRecord);
```

Remove the specified record from the list of records.

See also:
DeletedRecordCount

Destroy

Delphi declaration
```
Destroy; ;
```

Destructor of TkbmCommon. Do not call directly.

EmptyTables

Delphi declaration
```
EmptyTables;
```

Empty all attached tables.
Will clear all attached tables indexes and the internal record list.

See also:
TkbmCustomMemTable.EmptyTable

GetDeletedRecordsCount

Delphi declaration
```
GetDeletedRecordsCount: TkbmNativeInt;
```

Returns the number of entries in the deleted record list.

GetFieldDataOffset

Delphi declaration
```
GetFieldDataOffset(Field: TField): integer;
```

Return the offset in the record buffer where the provided field can be found.

GetFieldIsVarLength

Delphi declaration

```
GetFieldIsVarLength(FieldType: TFieldType; Size:
longint): boolean;
```

Returns true if the specified field type should be considered a variable length field.
The outcome depends on the field type, and the size of the field combined with the selected TkbmCustomMemTable.Performance settings.

See also:
TkbmCustomMemTable.Performance

GetFieldPointer

Delphi declaration

```
GetFieldPointer(ARecord: PkbmRecord; Field: TField):
PByte;
```

Return a pointer to the place where the specified field is in the specified record buffer.

GetFieldSize

Delphi declaration

```
GetFieldSize(FieldType: TFieldType; Size: longint):
longint;
```

Returns the number of bytes taken up by a field.
All varlength type fields (like blob fields) will return 0.

IncTransactionLevel

Delphi declaration

```
IncTransactionLevel;
```

Increase the transaction level by one.
Used by TkbmCustomMemTable.StartTransaction.

See also:
DecTransactionLevel TkbmCustomMemTable.StartTransaction TkbmCustomMemTable.TransactionLevel

IsAnyTableActive

Delphi declaration
```
IsAnyTableActive: boolean;
```

Returns true if any attached tables is open, otherwise false.

See also:
AttachTable DeAttachTable

LayoutRecord

Delphi declaration
```
LayoutRecord(AFieldCount: integer);
```

Defines the physical record layout from the table definitions.
Do not call directly.

Lock

Delphi declaration
```
Lock;
```

Locks the TkbmCommon to make sure other attached TkbmCustomMemTable's dont conflict during a critical operation.

See also:
Unlock

MarkIndexesDirty

Delphi declaration
```
MarkIndexesDirty;
```

Mark all attached table's indexes as dirty and in a need for a rebuild.

See also:
RebuildIndexes TkbmIndex.Rebuild TkbmIndexes.MarkAllDirty TkbmIndexes.ReBuildAll

PackRecords

Delphi declaration
```
PackRecords;
```

When a record is deleted, the record is freed, but the entry in the record list is not removed of performance reasons.
Instead a deleted record list is updated with the open spots, which will then be reused on a new record insertion.
Use TkbmCustomMemTable.PackTable to rebuild the recordlist without any open spots.
Usually not needed for normal operation where deletion and insertion happens relatively balanced. TkbmCustomMemTable.EmptyTable will also clear the deleted records list and remove all open spots.

See also:
DeletedRecordCount TkbmCustomMemTable.EmptyTable ERROR: INVALID LINK: TkbmCommon.DeletedRecord TkbmCommon.DeletedRecord

RebuildIndexes

Delphi declaration
```
RebuildIndexes;
```

Rebuild all indexes of all attached tables.

See also:
MarkIndexesDirty TkbmIndex.ReBuild TkbmIndexes.ReBuild TkbmIndexes.ReBuildAll

RecordCount

Delphi declaration
```
RecordCount: TkbmNativeInt;
```

Return the number of recordentries currently allocated. It includes open spots after deleted records.

ReflectToIndexes

Delphi declaration
```
ReflectToIndexes(Caller: TkbmCustomMemTable; How:
TkbmIndexUpdateHow; OldRecord, NewRecord: PkbmRecord;
RecordPos: integer; DontVersion: boolean);
```

Updates all indexes of all attached tables with the contents of the new record (NewRecord).
Its called by TkbmCustomMemTable whenever a record is inserted, deleted or modifed.
Do not call directly.

RefreshTables

Delphi declaration
```
RefreshTables(Caller: TkbmCustomMemTable);
```

Force a visual refresh of all attached tables.

See also:
ResyncTables ERROR: INVALID LINK: TDataset.Refresh TDataset.Refresh

ResyncTables

Delphi declaration
```
ResyncTables;
```

Force a visual resync of all attached tables.

See also:
RefreshTables ERROR: INVALID LINK: TDataset.Resync TDataset.Resync

RollBack

Delphi declaration
```
Rollback;
```

Rollback one transaction. Will effectively remove all changes since the start of the transaction.

See also:
Commit DecTransactionLevel IncTransactionLevel

Undo

Delphi declaration
```
Undo(ARecord: PkbmRecord);
```

Undo all changes made to the record identified by the record buffer pointer.
Essentially it will remove all newer versions of the record, and let the oldest version become the current version.

Unlock

Delphi declaration
```
Unlock;
```

Unlocks the TkbmCommon so other attached TkbmCustomMemTable's again can access the inner structures of TkbmCommon.

See also:
Lock

UpdateIndexes

Delphi declaration
```
UpdateIndexes;
```

Update both indexdefinitions and contents of all indexes for all tables.

See also:
TkbmCustomMemTable.UpdateIndexes

Protected Methods/Functions

_InternalAppendRecord

Delphi declaration
```
_InternalAppendRecord(ARecord: PkbmRecord);
```

Appends an allready allocated record buffer structure to the record list maintained by TkbmCommon.
If there are entries in the internal list of vacant slots in the record list, their spots are reused before extending the list.
The record will automatically be given a new unique record identifier upon append.

_InternalCheckRecord

Delphi declaration
```
_InternalCheckRecord(ARecord: PkbmRecord);
```

Optional method which checks the boundaries of a record buffer.
The code is only compiled in if the following definition has been set early in the source code:
{$DEFINE DO_CHECKRECORD}
The purpose of the method is to validate that record buffer overruns has not occured, and if they do occur, a EMemTableInvalidRecord exception is thrown as soon as possible.

_InternalClearRecord

Delphi declaration
```
_InternalClearRecord(ARecord: PkbmRecord);
```

Wipes out any contents of a record buffer and supporting data like unique record id etc.
It doesnt autmatically deallocate any indirect references to blobs or string fields.

_InternalCompareRecords

Delphi declaration

```
_InternalCompareRecords(AFieldList: TkbmFieldList;
AMaxFields: integer; AKeyRecord, ARecord: PkbmRecord;
AIgnoreNull, APartial, AUniqueConstraint: boolean;
AHow: TkbmCompareHow): Integer;
```

Compares two record buffers on the fields provided by **FieldList**.
If the **KeyRecord** has a lower value than the **ARecord** on the specified fields, it will return -1
If the **KeyRecord** has a higher value than the **ARecord** on the specified fields, it will return 1
If the **KeyRecord** and **ARecord** are equal on the specified fields, it will return 0
(These comparisons may be swapped if the index field options dictates a descending comparison instead of the default ascending).

FieldList provides a list over the fields (and their index options) that should be compared.
MaxFields specifies how many of the fields in the FieldList should be used for comparison. If -1, all fields will be used.
KeyRecord is one of the records to compare
ARecord is the record to compare to
IgnoreNull is a boolean value that controls if the comparison should ignore to compare fields which have a value of Null in the KeyRecord.
Partial is a boolean value that controls if all fields compared should automatically be compared as a partial compare.
How controls when the comparison ends. It can be:

chBreakNE - The comparison ends when the value of a field isnt the same in the two record buffers.

chBreakGTE - The comparison ends when the value of the field in KeyRecord is having the same or bigger value than the same field in ARecord.

chBreakLTE - The comparison ends when the value of the field in KeyRecord is having the same or lesser value than the same field in ARecord.

chBreakGT - The comparison ends when the value of the field in KeyRecord is having a bigger value than the same field in ARecord.

chBreakLT - The comparison ends when the value of the field in KeyRecord is having a lesser value than the same field in ARecord.

See also:

Field modifiers ERROR: INVALID LINK: Globals.CompareFields Globals.CompareFields

_InternalCopyRecord

Delphi declaration

```
_InternalCopyRecord(SourceRecord: PkbmRecord;
CopyVarLengths: boolean): PkbmRecord;
```

Makes an exact copy of a record buffer and returns the newly allocated record buffer.
CopyVarLengths controls if also indirect field contents (like blob and optionally strings) should also be duplicated during the copy.
If set to **false**, the new record will reference the exact same indirect field values as the original record.
If set to **true**, the new record will reference a copy of the indirect field values.

_InternalCopyVarLength

Delphi declaration

```
_InternalCopyVarLength(SourceRecord, DestRecord:
PkbmRecord; Field: TField);
```

Copy a specific varlength field (indirect field like blobs and optionally strings) from one record buffer to another.
If the destination record buffer already reference varlength data for the specified field, it will be free'ed first.
After the copy, the destination record will contain a copy of the varlength referenced in the source record for the specified field.

_InternalCopyVarLengths

Delphi declaration

```
_InternalCopyVarLengths(SourceRec, DestRec:
PkbmRecord);
```

Makes a copy of all varlength fields (blobs and optionall strings) in the source record and registeres the copy in the destination record.
If the destination record already references varlength data, these data will be free'ed before the copy.

_InternalDeleteRecord

Delphi declaration

```
_InternalDeleteRecord(ARecord: PkbmRecord);
```

Deletes a specific record, given by the record buffer pointer, from the internal record list.
The now open spot in the record list will be added to the deleted record list for later reuse.
The actual record buffer will be automatically free'ed along with all linked versions of it and all varlengths.

_InternalEmpty

Delphi declaration

```
_InternalEmpty;
```

Empties the internal record list completely in the fastest possible way.
All record buffers are free'ed along with their linked versions and varlengths.
The deleted records list is emptied and the unique record ID counter is reset.

_InternalFreeRecordVarLengths

Delphi declaration

```
_InternalFreeRecordVarLengths(ARecord: PkbmRecord);
```

Free's all varlength fields (blobs and optionally strings) in the provided record buffer.

_InternalMoveRecord

Delphi declaration

```
_InternalMoveRecord(SourceRecord, DestRecord:
PkbmRecord);
```

Moves the contents of the source record buffer into he destination record buffer.
If there are varlengths defined, the destination record buffer will reference the exact same varlengths as the source record buffer.

_InternalPackRecords

Delphi declaration

```
_InternalPackRecords;
```

Packs the internal record list.
If records has been deleted from the record list, there will be unfilled holes in it.
These holes have been registered in the deleted record list for later reuse.

However if one wants to compress the record list as much as possible, one can call this method.
It will clear the deleted records list, and renumber the record id of all records (not to confuse with the unique record id which will never be changed).

_InternalTransferRecord

Delphi declaration

```
_InternalTransferRecord(SourceRecord, DestRecord:
PkbmRecord);
```

A bit similar to _InternalMoveRecord except this is used for transfering a buffer record (one held by dataaware controls) to a physical record buffer.
The two buffers are of the same structure, but varlengths/indirect fields are handled differently to avoid the need of duplicating large blobs into the dataawares buffer space.
kbmMemTable contains logic to automatically find the actual physical record from a data aware controls buffer record, and thus provide the blob on demand.

CalcLocaleID

Delphi declaration

```
CalcLocaleID;
```

Calculates a locale ID based on current LanguageID, SublanguageID and SortID values

ClearModifiedFlags

Delphi declaration

```
ClearModifiedFlags;
```

Clear out all the flags indicating that a field has been modified.

EnsureFields

Delphi declaration

```
EnsureFields(ACount: integer);
```

GetAttachCount

Delphi declaration

```
GetAttachCount: integer;
```

Getter method that returns the number of currently attached tables. The number may very well be lower than the defined max number of attached tables.

See also:
TkbmCommon.GetAttachMaxCount TkbmCommon.SetAttachMaxCount

GetAttachMaxCount

Delphi declaration
```
GetAttachMaxCount: integer;
```

Getter method that returns the define max number of attached tables.

See also:
TkbmCommon.GetAttachCount TkbmCommon.SetAttachMaxCount

GetAutoIncMax

Delphi declaration
```
GetAutoIncMax: TkbmNativeInt;
```

Getter method that returns the highest value in use for the autoinc field.

See also:
TkbmCommon.SetAutoIncMax

GetAutoIncMin

Delphi declaration
```
GetAutoIncMin: TkbmNativeInt;
```

Getter method that returns the defined smallest allowed autoinc value.

See also:
TkbmCommon.SetAutoIncMin

GetCapacity

Delphi declaration
```
GetCapacity: longint;
```

Getter method that returns the define record list capacity.
The capacity will automatically be increased when needed. However predefining a correct starting capacity can provide a slight performance improvement in some cases.

See also:
TkbmCommon.SetCapacity

GetEnableVersioning

Delphi declaration
```
GetEnableVersioning: boolean;
```

Getter method that returns a boolean value indicating if record versioning has been enabled.

See also:
TkbmCommon.SetEnableVersioning

GetIsDataModified

Delphi declaration
```
GetIsDataModified: boolean;
```

Getter method that returns a boolean value indicating if data has been modified in the memory table since the DataModified flag was reset.

See also:
TkbmCommon.SetIsDataModified

GetLanguageID

Delphi declaration
```
GetLanguageID: integer;
```

Getter method returning the currently selected Language ID value.

See also:
TkbmCommon.GetLocaleID TkbmCommon.GetSortID TkbmCommon.GetSubLanguageID TkbmCommon.SetLanguageID

GetLocaleID

Delphi declaration
```
GetLocaleID: TkbmLocaleID;
```

Getter method returning the currently selected Locale ID.

See also:
TkbmCommon.GetLanguageID TkbmCommon.GetSortID
TkbmCommon.GetSubLanguageID TkbmCommon.SetLocaleID

GetModifiedFlag

Delphi declaration
```
GetModifiedFlag(i: integer): boolean;
```

Getter method returning a boolean value indicating if the current records field has been modified since navigating to the record.

See also:
TkbmCommon.SetModifiedFlag

GetPerformance

Delphi declaration
```
GetPerformance: TkbmPerformance;
```

Getter method returning the current performance setting.

See also:
TkbmCommon.SetPerformance

GetSortID

Delphi declaration
```
GetSortID: integer;
```

Getter method returning the currently selected Sort ID.

See also:
TkbmCommon.GetLanguageID TkbmCommon.GetLocaleID
TkbmCommon.GetSubLanguageID TkbmCommon.SetSortID

GetStandalone

Delphi declaration
```
GetStandalone: boolean;
```

Getter method returning if the memtable currently is setup as a standalone table.

See also:
TkbmCommon.SetStandalone

GetSubLanguageID

Delphi declaration
```
GetSubLanguageID: integer;
```

Getter method that returns the currently specified language ID.

See also:
TkbmCommon.GetLanguageID TkbmCommon.GetLocaleID TkbmCommon.GetSortID TkbmCommon.SetSubLanguageID

GetTransactionLevel

Delphi declaration
```
GetTransactionLevel: integer;
```

Function returning the current transaction level.

GetUniqueDataID

Delphi declaration
```
GetUniqueDataID: longint;
```

Generates a (mostly) unique long int value.

See also:
TkbmCommon.SetUniqueRecordID

GetVersioningMode

Delphi declaration
```
GetVersioningMode: TkbmVersioningMode;
```

Getter method returning the current versioning mode setting.

See also:
TkbmCommon.SetVersioningMode

SetAttachMaxCount

Delphi declaration
```
SetAttachMaxCount(Value: integer);
```

Setter method setting the max number of dynamically attached tables supported by this memtable.

See also:
TkbmCommon.GetAttachCount TkbmCommon.GetAttachMaxCount

SetAutoIncMax

Delphi declaration
```
SetAutoIncMax(Value: TkbmNativeInt);
```

Setter method setting the largest integer value the autoinc field has in the table.

See also:
TkbmCommon.GetAutoIncMax

SetAutoIncMin

Delphi declaration
```
SetAutoIncMin(Value: TkbmNativeInt);
```

Setter method setting the lowest integer value the autoinc field can have.

See also:
TkbmCommon.GetAutoIncMin

SetCapacity

Delphi declaration
```
SetCapacity(Value: longint);
```

Setter method that specifies the capacity of the internal record list.

See also:
TkbmCommon.GetCapacity

SetDeletedCount

Delphi declaration
```
SetDeletedCount(ACount: longint);
```

Setter method that specifies the number of deleted records.

See also:
TkbmCommon.GetDeletedRecordsCount

SetEnableVersioning

Delphi declaration
```
SetEnableVersioning(Value: boolean);
```

Setter method specifying if record versioning is currently enabled.

See also:
TkbmCommon.GetEnableVersioning

SetIsDataModified

Delphi declaration
```
SetIsDataModified(Value: boolean);
```

Setter method specifying if the data in the memtable has been modified.

See also:
TkbmCommon.GetIsDataModified

SetLanguageID

Delphi declaration
```
SetLanguageID(Value: integer);
```

Setter method specifying the current language ID

See also:
TkbmCommon.SetLocaleID

SetLocaleID

Delphi declaration
```
SetLocaleID(Value: TkbmLocaleID);
```

Setter method specifying the current locale ID

See also:
TkbmCommon.GetLocaleID TkbmCommon.SetLanguageID TkbmCommon.SetSortID TkbmCommon.SetSubLanguageID

SetModifiedFlag

Delphi declaration
```
SetModifiedFlag(i: integer; Value: boolean);
```

Setter method specifying that a specific field (given by an index from 0 to FieldCount-1) has been modified since navigation to the current record.

See also:
TkbmCommon.GetModifiedFlag

SetPerformance

Delphi declaration
```
SetPerformance(Value: TkbmPerformance);
```

Setter method specifying the current performance mode.

See also:
TkbmCommon.GetPerformance

SetRecordID

Delphi declaration
```
SetRecordID(ARecordID: TkbmNativeInt);
```

Setter method specifying the current record id (not to confuse with the unique record id).

See also:
TkbmCommon.GetLocaleID

SetSortID

Delphi declaration
```
SetSortID(Value: integer);
```

Setter method specifying the current sort ID

See also:
TkbmCommon.GetSortID TkbmCommon.SetLanguageID TkbmCommon.SetLocaleID TkbmCommon.SetSubLanguageID

SetStandalone

Delphi declaration
```
SetStandalone(Value: boolean);
```

Setter method specifying if the table is standalone or not (if yes, all threadlocking is disabled).

See also:
TkbmCommon.GetStandalone

SetSubLanguageID

Delphi declaration
```
SetSubLanguageID(Value: integer);
```

Setter method specifying the current sub language ID

See also:
TkbmCommon.GetSubLanguageID TkbmCommon.SetLanguageID TkbmCommon.SetLocaleID TkbmCommon.SetSortID

SetUniqueRecordID

Delphi declaration
```
SetUniqueRecordID(ARecordID: TkbmNativeInt);
```

Setter method specifying the next unique record id to be used when creating new records.

SetVersioningMode

Delphi declaration
```
SetVersioningMode(Value: TkbmVersioningMode);
```

Setter method specifying the chosen record versioning mode.

See also:
TkbmCommon.GetVersioningMode

TkbmCompareHow File: kbmMemTable

Value	Meaning
chBreakNE:	
chBreakLT:	
chBreakGT:	
chBreakLTE:	
chBreakGTE:	

Delphi declaration

```
TkbmCompareHow =(chBreakNE, chBreakLT, chBreakGT, chBreakLTE, chBreakGTE);
```

TkbmCS File: kbmMemTable

Delphi declaration
```
TkbmCS=Class(TCriticalSection);
```

Public Methods/Functions

Acquire

Delphi declaration
```
Acquire; ;
```

Create

Delphi declaration
```
Create(ACommon: TkbmCommon); ; ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

DumpAttachedTables

Delphi declaration
```
DumpAttachedTables(AText: );
```

DumpLockCount

Delphi declaration
```
DumpLockCount(AText: );
```

Release

Delphi declaration
```
Release; ;
```

TkbmCSVStreamFormat File: kbmMemCSVStreamFormat

TkbmCSVStreamFormat is one of the record streaming classes bundled with kbmMemTable.
Its used for storing/loading kbmMemTable data in a comma seperated format.
It default always stores data in the current index order.

Delphi declaration

```
TkbmCSVStreamFormat=Class(TkbmCustomCSVStreamFormat);
```

Public Events

TkbmCustomBinaryStreamFormat File: kbmMemBinaryStreamFormat

This can be the base for all binary stream formatters.
Binary stream formatters have the advantages of speed and compact data compared to CSV streamformatters.
In addition binary stream formatters support streaming deltas (record versions) and thus also information about deleted records.
CSV stream formatters have in turn the advantage of more flexibility, but generally doesnt support streaming deltas.

Delphi declaration
```
TkbmCustomBinaryStreamFormat=Class(TkbmCustomStreamFormat);
```

Protected Properties/Variables

BufferSize

Delphi declaration
```
BufferSize: LongInt FBuffSize SetBuffSize;
```

BufferSize controls the size of the read and write buffer.
Default size is 16kb.

sfDataTypeHeader

Delphi declaration
```
sfDataTypeHeader: TkbmStreamFlagDataTypeHeaders
FDataTypeHeader FDataTypeHeader;
```

See TkbmStreamFlagDataTypeHeader

sfUsingIndex

Delphi declaration
```
sfUsingIndex: TkbmStreamFlagUsingIndexs FUsingIndex
FUsingIndex;
```

See TkbmStreamFlagUsingIndex

Public Methods/Functions

Create

Delphi declaration
```
Create(AOwner: TComponent); ;
```

Protected Methods/Functions

AfterLoad

Delphi declaration
```
AfterLoad(ADataset: TkbmCustomMemTable); ;
```

AfterSave

Delphi declaration

```
AfterSave(ADataset: TkbmCustomMemTable); ;
```

BeforeLoad

Delphi declaration

```
BeforeLoad(ADataset: TkbmCustomMemTable); ;
```

BeforeSave

Delphi declaration

```
BeforeSave(ADataset: TkbmCustomMemTable); ;
```

DetermineLoadFieldIndex

Delphi declaration

```
DetermineLoadFieldIndex(ADataset: TkbmCustomMemTable;
ID: ; FieldCount: integer; OrigIndex: integer; NewIndex:
integer; Situation: TkbmDetermineLoadFieldsSituation);
;
```

GetVersion

Delphi declaration

```
GetVersion: ; ;
```

LoadData

Delphi declaration

```
LoadData(ADataset: TkbmCustomMemTable); ;
```

LoadDef

Delphi declaration

```
LoadDef(ADataset: TkbmCustomMemTable); ;
```

SaveData

Delphi declaration

```
SaveData(ADataset: TkbmCustomMemTable); ;
```

SaveDef

Delphi declaration

```
SaveDef(ADataset: TkbmCustomMemTable); ;
```

TkbmCustomCSVStreamFormat File: kbmMemCSVStreamFormat

This can be the base for all CSV (Comma Seperated Values) stream formatters.
CSV stream formatters have the advantage of more flexibility, but generally doesnt support streaming deltas.
Binary stream formatters have in turn the advantages of speed and compact data compared to CSV streamformatters.
In addition binary stream formatters support streaming deltas (record versions) and thus also information about deleted records.

Delphi declaration
```
TkbmCustomCSVStreamFormat=Class(TkbmCustomStreamFormat);
```

Public Properties/Variables

CommentChar

Delphi declaration
```
CommentChar: Char FCommentChar FCommentChar;
```

CSVFalseString

Delphi declaration
```
CSVFalseString: FCSVFalseString FCSVFalseString;
```

The textual representation of a False boolean value.
Default: False

CSVFieldDelimiter

Delphi declaration
```
CSVFieldDelimiter: Char FCSVFieldDelimiter
SetCSVFieldDelimiter;
```

The character to place between data of fields.
Default: , (comma)

CSVQuote

Delphi declaration
```
CSVQuote: Char FCSVQuote FCSVQuote;
```

The character to use to quote fields with.
Default: " (double quote)

CSVRecordDelimiter

Delphi declaration
```
CSVRecordDelimiter: Char FCSVRecordDelimiter
FCSVRecordDelimiter;
```

The character to seperate each record from each other.
Default: , (comma)

CSVTrueString

Delphi declaration

```
CSVTrueString: FCSVTrueString FCSVTrueString;
```

The textual representation of a True boolean value.
Default: True

DefaultStringFieldSize

Delphi declaration

```
DefaultStringFieldSize: integer
FDefaultStringFieldSize FDefaultStringFieldSize;
```

EscapeChar

Delphi declaration

```
EscapeChar: Char FEscapeChar FEscapeChar;
```

sfAutoInc

Delphi declaration

```
sfAutoInc: TkbmStreamFlagsAutoInc FsfAutoInc
FsfAutoInc;
```

sfLocalFormat

Delphi declaration

```
sfLocalFormat: TkbmStreamFlagsLocalFormat
FsfLocalFormat FsfLocalFormat;
```

Controls what date/time and floating point conversions that should happen.
If the flag is not specified, the following settings are automatically used:

```
DateSeparator:='/';
TimeSeparator:=':';
ThousandSeparator:=',';
DecimalSeparator:='.';
ShortDateFormat:='dd/mm/yyyy';
CurrencyString:='';
CurrencyFormat:=0;
```

NegCurrFormat:=1;
If specified, the current settings of the application is used.

sfNoHeader

Delphi declaration
```
sfNoHeader: TkbmStreamFlagsNoHeader FsfNoHeader
FsfNoHeader;
```

If specified, a header containing the DisplayName of the fields is not saved/loaded (expected).

sfPlaceHolders

Delphi declaration
```
sfPlaceHolders: TkbmStreamFlagsPlaceHolders
FsfPlaceHolders FsfPlaceHolders;
```

If specified, placeholders will be saved/expected during load for fields which are otherwise not saved due to other constraints.

sfQuoteOnlyStrings

Delphi declaration
```
sfQuoteOnlyStrings: TkbmStreamFlagsQuoteOnlyStrings
FsfQuoteOnlyStrings FsfQuoteOnlyStrings;
```

If specified, only string values will be quoted.
Default all values are quoted.

Public Methods/Functions

Assign

Delphi declaration
```
Assign(Source: TPersistent); ;
```

Create

Delphi declaration
```
Create(AOwner: TComponent); ;
```

Protected Methods/Functions

AfterLoad

Delphi declaration
```
AfterLoad(ADataset: TkbmCustomMemTable); ;
```

AfterSave

Delphi declaration

```
AfterSave(ADataset: TkbmCustomMemTable); ;
```

BeforeLoad

Delphi declaration

```
BeforeLoad(ADataset: TkbmCustomMemTable); ;
```

BeforeSave

Delphi declaration

```
BeforeSave(ADataset: TkbmCustomMemTable); ;
```

DetermineLoadFieldIDs

Delphi declaration

```
DetermineLoadFieldIDs(ADataset: TkbmCustomMemTable;
AList: TStringList; Situation:
TkbmDetermineLoadFieldsSituation); ;
```

DetermineLoadFieldIndex

Delphi declaration

```
DetermineLoadFieldIndex(ADataset: TkbmCustomMemTable;
ID: ; FieldCount: integer; OrigIndex: integer; NewIndex:
integer; Situation: TkbmDetermineLoadFieldsSituation);
;
```

GetChunk

Delphi declaration

```
GetChunk: boolean; ;
```

Loads a chunk of data into memory from the stream.
Also keeps track of how far the load process has progressed.

See also:
TkbmCustomCSVStreamFormat.GetLine
TkbmCustomCSVStreamFormat.GetWord

GetLine

Delphi declaration
```
GetLine: boolean; ;
```

Gets a line (ending with CR or LF or a combination) from the internal data buffer.

See also:
TkbmCustomCSVStreamFormat.GetChunk
TkbmCustomCSVStreamFormat.GetWord

GetVersion

Delphi declaration
```
GetVersion: ; ;
```

GetWord

Delphi declaration
```
GetWord(null: boolean): ; ;
```

Gets the next word from the currently loaded line.
null is set to true if the word indicates a null field value.
Otherwise returns the word as s string.

See also:
TkbmCustomCSVStreamFormat.GetChunk
TkbmCustomCSVStreamFormat.GetLine

LoadData

Delphi declaration
```
LoadData(ADataset: TkbmCustomMemTable); ;
```

LoadDef

Delphi declaration
```
LoadDef(ADataset: TkbmCustomMemTable); ;
```

SaveData

Delphi declaration

```
SaveData(ADataset: TkbmCustomMemTable); ;
```

SaveDef

Delphi declaration

```
SaveDef(ADataset: TkbmCustomMemTable); ;
```

WriteString

Delphi declaration

```
WriteString(AString: ); ;
```

Public Events

OnFormatLoadField

Delphi declaration

```
OnFormatLoadField: TkbmOnFormatLoadField
FOnFormatLoadField FOnFormatLoadField;
```

See TkbmOnFormatLoadField

OnFormatSaveField

Delphi declaration

```
OnFormatSaveField: TkbmOnFormatSaveField
FOnFormatSaveField FOnFormatSaveField;
```

See TkbmOnFormatSaveField

OnGetLine

Delphi declaration

```
OnGetLine: TkbmOnGetLine FOnGetLine FOnGetLine;
```

TkbmCustomDeltaHandler File: kbmMemTable

A delta handler is a class which is designed specifically to update other external storages (f.ex. a SQL table) using the record changes made in a TkbmMemTable.
For a deltahandler to work, it needs some data from the TkbmMemTable. Those data are only available if EnableVersioning is true.

To create a deltahandler you need to inherit from TkbmCustomDeltaHandler and override one or more of the methods DeleleRecord, InsertRecord, ModifyRecord, UnmodifiedRecord and optionally Resolve.

Each of DeleteRecord, InsertRecord, ModifyRecord and UnmodifiedRecord will be called as appropriate for each record in the TkbmMemTable you reference through the Dataset property when you call the Resolve method of the deltahandler.

The Resolve method contains all logic needed to figure out what has happened in the TkbmMemTable, and how to handle it. But if you are having special requirements, you can override Resolve too.

You should make sure that you are positive that either one of the fields of the TkbmMemTable is unique or a combination of fields are unique, and that they somehow match up with one unique record in the external storage.

Notice that if you make a Checkpoint in the table, all record versions before that checkpoint are removed, and all records will therefore be marked as unmodified at that time. Deleted records will be lost after a Checkpoint. Later updates to the TkbmMemTable will ofcourse again result in new versions and records will again be marked as modifed, inserted or deleted.

Delphi declaration

```
TkbmCustomDeltaHandler=Class(TComponent);
```

Delphi Example

```
// An example on how to create a deltahandler.
TDemoDeltaHandler = class(TkbmCustomDeltaHandler)
protected
  procedure InsertRecord; override;
  procedure DeleteRecord; override;
  procedure ModifyRecord; override;
end;

implementation

// An example on how to create a deltahandler.
procedure TDemoDeltaHandler.ModifyRecord;
var
   i:integer;
   s1,s2,sv:string;
   v:variant;
begin
     s1:='';
     s2:='';
     for i:=0 to FieldCount-1 do
     begin
          if (not (Fields[i].DataType in kbmBinaryTypes)) or
(Fields[i].DataType=ftMemo) then
          begin
               v:=Values[i];
               if (VarIsNull(v)) then sv:='<NULL>'
               else sv:=v;
               s1:=s1+sv+' ';

               v:=OrigValues[i];
               if (VarIsNull(v)) then sv:='<NULL>'
               else sv:=v;
               s2:=s2+sv+' ';
          end;
     end;
     ShowMessage(Format('Modified record (%s) to (%s)',[s2,s1]));
end;

procedure TDemoDeltaHandler.InsertRecord;
var
   i:integer;
   s,sv:string;
   v:variant;
begin
     s:='';
     for i:=0 to FieldCount-1 do
     begin
          v:=Values[i];
          if not (Fields[i].DataType in kbmBinaryTypes) then
          begin
               if (VarIsNull(v)) then sv:='<NULL>'
               else sv:=v;
               s:=s+sv+' ';
          end;
     end;
     ShowMessage(Format('Inserted record (%s)',[s]));
end;

procedure TDemoDeltaHandler.DeleteRecord;
var
   i:integer;
```

```
   s,sv:string;
   v:variant;
begin
     s:='';
     for i:=0 to FieldCount-1 do
     begin
          v:=Values[i];
          if not (Fields[i].DataType in kbmBinaryTypes) then
          begin
               if (VarIsNull(v)) then sv:='<NULL>'
               else sv:=v;
               s:=s+sv+' ';
          end;
     end;
     ShowMessage(Format('Deleted record (%s)',[s]));
end;
```

See Also:
TkbmCustomMemTable.CheckPoint

Public Properties/Variables

DataSet

Delphi declaration
```
DataSet: TkbmCustomMemTable FDataSet FDataSet;
```

Dataset references the TkbmCustomMemTable which should be resolved.
It will be automatically set when you set the DeltaHandler property of the TkbmCustomMemTable to point at this deltahandler.

See also:
TkbmCustomMemTable.DeltaHandler

FieldCount

Delphi declaration
```
FieldCount: integer GetFieldCount;
```

Returns the number of fields available in the TkbmCustomMemTable.

FieldNames

Delphi declaration
```
FieldNames[i: integer]: GetFieldNames;
```

Returns the fieldname for a specific field index as defined in the TkbmCustomMemTable.

Fields

Delphi declaration
```
Fields[i: integer]: TField GetFields;
```

Return a field reference using the field index as defined in the TkbmCustomMemTable.
You can f.ex. use this to figure out the type of a field.

See also:
FieldCount FieldNames

OrigValues

Delphi declaration
```
OrigValues[i: integer]: Variant GetOrigValues;
```

Return the original values of a field, specified via an index, for the current record.
This will return same values as the Values property except for modified records, in which case it will return the original value of the record before modification.

See also:
FieldCount OrigValuesByName Values ValuesByName

OrigValuesByName

Delphi declaration
```
OrigValuesByName[: ]: Variant GetOrigValuesByName;
```

Return the original values of a field, specified via a field name, for the current record.
This will return same values as the Values property except for modified records, in which case it will return the original value of the record before modification.

See also:
OrigValues Values ValuesByName

RecNo

Delphi declaration
```
RecNo: TkbmNativeInt GetRecordNo;
```

Returns the record number (1..RecordCount) of the current record based on the current index of the memory table.

UniqueRecID

Delphi declaration
```
UniqueRecID: TkbmNativeInt GetUniqueRecordID;
```

Returns a unique record identification number for the current record. All records added/inserted to the memory table are getting a unique number which will never be repeated during until the table is closed.

Values

Delphi declaration

```
Values[i: integer]: Variant GetValues;
```

Return the current values of a field, specified via an index, for the current record.

See also:
FieldCount OrigValues OrigValuesByName ValuesByName

ValuesByName

Delphi declaration

```
ValuesByName[: ]: Variant GetValuesByName;
```

Return the current values of a field, specified via an index, for the current record.

See also:
OrigValues OrigValuesByName Values

Public Methods/Functions

Resolve

Delphi declaration

```
Resolve; ;
```

Call this method to start resolving the deltas.
Resolving means to check each record if something has happened with it and then act accordingly.
Resolve will call DeleteRecord whenever a record has been deleted, InsertRecord whenever a new record has been inserted, ModifyRecord whenever a record has been modified and UnmodifiedRecord whenever nothing has happened to the record.
Resolve also automatically figures out which was the original value of the record in the case of a deleted or modified record.

If you have special resolving needs, you can override this method.

See also:
DeleteRecord InsertRecord ModifyRecord UnmodifiedRecord

Protected Methods/Functions

AfterRecord

Delphi declaration
```
AfterRecord; ;
```

BeforeRecord

Delphi declaration
```
BeforeRecord; ;
```

DeleteRecord

Delphi declaration
```
DeleteRecord(Retry: boolean; State: TUpdateStatus); ;
```

This method is called every time a record has been deleted from the TkbmMemTable.
You can determine the value of the fields of the deleted record using the Values or the ValuesByName properties.
In an SQL deltahandler you should then generate the correct SQL statement eg. 'DELETE FROM xxx WHERE xxx1=yyy ...'
where xxx1 is a name of a field and yyy is the Values or ValuesByName value.

See also:
InsertRecord ModifyRecord UnmodifiedRecord

InsertRecord

Delphi declaration
```
InsertRecord(Retry: boolean; State: TUpdateStatus); ;
```

This method is called every time a record has been inserted into the TkbmMemTable.
You can determine the value of the fields of the inserted record using the Values or the ValuesByName properties.
In an SQL deltahandler you should then generate the correct SQL statement eg. 'INSERT INTO xxx (xxx1,....) VALUES (yyy,....)'
where xxx1 is a name of a field and yyy is the Values or ValuesByName value.

ModifyRecord

Delphi declaration

```
ModifyRecord(Retry: boolean; State: TUpdateStatus); ;
```

This method is called every time a record has been modifed.
You can determine the current value of the fields of the modified record using the Values or the ValuesByName properties.
You can use OrigValues and OrigValuesByName to get the original values from before the modification.
In an SQL deltahandler you should then generate the correct SQL statement
eg. UPDATE xxx SET xxx1=yyy1,.... WHERE xxx2=yyy2 ....'
where xxx1 and xxx2 are a names of fields and yyy1 is the Values or ValuesByName value, and yyy2 the OrigValues or OrigValuesByName.

See also:
DeleteRecord InsertRecord UnmodifiedRecord

Notification

Delphi declaration

```
Notification(AComponent: TComponent; Operation:
TOperation); ;
```

UnmodifiedRecord

Delphi declaration

```
UnmodifiedRecord(Retry: boolean; State:
TUpdateStatus); ;
```

This method is called every time a record has not been modifed, inserted or deleted from the TkbmMemTable.
You can determine the value of the fields of the unmodified record using the Values or the ValuesByName properties.
Generally this is normally not used in an SQL delta handler.

See also:

DeleteRecord InsertRecord ModifyRecord

Public Events

OnGetValue

Delphi declaration

```
OnGetValue: TkbmDeltaHandlerGetValue FOnGetValue
FOnGetValue;
```

Event which is called each time a field value is required.
Its possible to alter the value returned to the deltahandler at runtime, without affecting the original memtable by using this event.

TkbmCustomMemTable File: kbmMemTable

This class is the base for memory based datasets, and contains all required functionality. You will however usually never instantiate this class, but rather the descendant class TkbmMemTable.

Delphi declaration
```
TkbmCustomMemTable=Class(TDataSet);
```

Public Properties/Variables

AllData

Delphi declaration
```
AllData: variant GetAllData SetAllData;
```

Gets or sets the contents of a memory table using a variant.
This can be used to easily move the contents of one memory table optionally incl. definitions etc. to another memory table.
Internally the data is stored in the variant in the format specified by AllDataFormat.

Delphi Example
Example 1 - Fetch the contents of a memory table into a variant, then close the table and later put the variant back into the memory table.

```
var
 v:Variant;
begin
 mt.AllDataOptions:=kbmSaveAllData;
 v:=mt.AllData;
 mt.Close;
 mt.Open;
 mt.AllData:=v;
end;
```

See also:
ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.InternalLoadFromBinaryStream InternalLoadFromBinaryStream ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.InternalSaveToBinaryStream InternalSaveToBinaryStream

AllDataFormat

Delphi declaration
```
AllDataFormat: TkbmCustomStreamFormat FAllDataFormat
FAllDataFormat;
```

Set to the streamformatter used for generating AllData data.

AllDataOptions

Delphi declaration
property AllDataOptions: TkbmMemTableSaveFlags read FAllDataOptions write FAllDataOptions;

These options specify how the memory table should be stored and which data should be part of the save when AllData is fetched.

See also:
AllData

AttachCount

Delphi declaration
```
AttachCount: integer GetAttachCount;
```

Return the current number of tables attached to the same base table.

AttachedAutoRefresh

Delphi declaration
```
AttachedAutoRefresh: boolean FAttachedAutoRefresh
FAttachedAutoRefresh;
```

Set to true or false depending on if all attached tables should be auto refreshed on each update of the physical records.
If you set it to false, you have to refresh the tables yourself, and you can risc having data shown which is not valid anymore because the physical record has changed or been deleted.

AttachedTo

Delphi declaration
```
AttachedTo: TkbmCustomMemTable FAttachedTo
SetAttachedTo;
```

Set this to point to another memory table if you want to use that table as the holder of the physical records.
This way you can have different virtual tables which all looks at the same data, but with different filters, indexes, ranges, master/detail setup etc. Essentially its much like having more cursors into the same data.
If AttachedAutoRefresh is true, all changes made to the physical records will be automatically reflected to all the attached tables.
ne You cannot attach to a table which itself is attached to another table.
Unless you specify a number bigger than the default value of 1 in AttachMaxCount, the base table has to be closed before any attachment can be made.

See also:
AttachCount AttachedAutoRefresh AttachMaxCount

AttachMaxCount

Delphi declaration

```
AttachMaxCount: integer GetAttachMaxCount
SetAttachMaxCount;
```

Specify the maximum number of tables you want to attach to the base table. Default is 1.
The reason for setting this value should be if you dont want the base table to be closed before you can attach a table more to it. This way attachments can be done on the fly without loosing the memory table data.
Dont set the count higher than absolutely needed, since each physical record will grow with *AttachMaxCount* * 8 bytes extra.

See also:
AttachCount AttachedAutoRefresh AttachedTo

AutoAddIndexes

Delphi declaration

```
AutoAddIndexes: boolean FAutoAddIndexes
FAutoAddIndexes false;
```

If set to true, will automatically create indexes to speed up searches when Locate/Lookup or other searching is performed.
Be aware that depending on the variation of the searches this can result in many indexes that must be maintained on updates.

AutoIncMinValue

Delphi declaration

```
AutoIncMinValue: longint GetAutoIncMin
SetAutoIncMinValue 0;
```

Set this to specify the next value for an auto increment field when a new record is inserted/appended.
There can only be one ftAutoInc field in a memory table. The default start value is 1.

AutoIncValue

Delphi declaration

```
AutoIncValue: longint GetAutoIncValue;
```

Returns the currently highest used auto increment value used for a ftAutoInc field.

AutoReposition

Delphi declaration

```
AutoReposition: boolean FAutoReposition
FAutoReposition false;
```

If you make a change in a record, and that results in the record being moved to another place in the current index, it could result in the record seemingly disappearing from f.ex a DBGrid.
If you want the grid or any other controls to 'follow' to the new place of the record, set *AutoReposition* to true.

AutoUpdateFieldVariables

Delphi declaration

```
AutoUpdateFieldVariables: boolean
FAutoUpdateFieldVariables
SetAutoUpdateFieldVariables;
```

Set to true to automatically have kbmMemTable update the global persistent field variables on the form when the table is opened.

BeforeInsert

Delphi declaration

```
BeforeInsert: TDatasetNotifyEvent FBeforeInsert
FBeforeInsert;
```

Capacity

Delphi declaration

```
Capacity: longint GetCapacity SetCapacity;
```

This property can be used to preallocate space in the physical record table for minimum Capacity number of records.
Normally there is no reason to set this property, except if you want to squeeze the last few msecs of performance out of the application during large inserts of records.

CommaText

Delphi declaration

```
CommaText: GetCommaText SetCommaText;
```

Actually misleading name as this property returns the contents of the table as a string.
If it is a CSV formatted string is determined by the setting of CommaTextFormat.

CommaTextFormat

Delphi declaration

```
CommaTextFormat: TkbmCustomStreamFormat
FCommaTextFormat FCommaTextFormat;
```

The streamformatter used when using the CommaText property.

Common

Delphi declaration

```
Common: TkbmCommon FCommon;
```

This is the direct gateway to the internal physical record structures. Generally dont fool around with this unless you know what you are doing.

CSVFalseString

Delphi declaration

property CSVFalseString: string read FCSVFalseString write FCSVFalseString;

Set this to the string you want to represent a False value in a boolean field during CSV text save/load operations.
Default is 'False'

See also:

CSVFieldDelimiter CSVQuote CSVRecordDelimiter CSVTrueString LoadFromFile LoadFromStream SaveToFile SaveToStream

CSVFieldDelimiter

Delphi declaration

property CSVFieldDelimiter: char read FCSVFieldDelimiter write FCSVFieldDelimiter;

Set this to the character which should seperate fields during CSV text load and save operations.
For save operations this can be set to any character, but notice that to be able to load the data, the delimiter should be unique.
Default is ',' (comma)

See also:
CSVFalseString CSVQuote CSVRecordDelimiter CSVTrueString LoadFromFile LoadFromStream SaveToFile SaveToStream

CSVQuote

Delphi declaration
property CSVQuote: char read FCSVQuote write FCSVQuote;

Set this to the character which should surround fields during CSV text load and save operations.
For save operations this can be set to any character, but notice that to be able to load the data, the quote should be unique.
Default is " (double quote)

See also:
CSVFalseString CSVFieldDelimiter CSVRecordDelimiter CSVTrueString LoadFromFile LoadFromStream SaveToFile SaveToStream

CSVRecordDelimiter

Delphi declaration
property CSVRecordDelimiter: char read FCSVRecordDelimiter write FCSVRecordDelimiter;

Set this to the character which should seperate records during CSV text load and save operations.
For save operations this can be set to any character, but notice that to be able to load the data, the delimiter should be unique.
You can disable saving a record seperator by setting it to #0.
Notice that all records will anyway be delimited with CR/LF pairs.
Default is ',' (comma)

See also:
CSVFalseString CSVFieldDelimiter CSVQuote CSVTrueString LoadFromFile LoadFromStream SaveToFile SaveToStream

CSVTrueString

Delphi declaration
property CSVTrueString: string read FCSVTrueString write FCSVTrueString;

Set this to the string you want to represent a True value in a boolean field during CSV text save/load operations.
Default is 'True'

See also:
CSVFalseString CSVFieldDelimiter CSVQuote CSVRecordDelimiter LoadFromFile LoadFromStream SaveToFile SaveToStream

CurIndex

Delphi declaration
```
CurIndex: TkbmIndex FCurIndex;
```

Returns a reference to the current index.

DefaultFormat

Delphi declaration
```
DefaultFormat: TkbmCustomStreamFormat FDefaultFormat
FDefaultFormat;
```

This is the streamformatter used when doing LoadFromFile/Stream and SaveToFile/Stream.

DeletedRecordsCount

Delphi declaration
```
DeletedRecordsCount: TkbmNativeInt
GetDeletedRecordsCount;
```

Return the number of open spots in the internal physical table record list.
It will increase on each delete of a record (unless the delete is versioned), and decrease on each insert/append.
You can zero this value by using PackTable .

DeltaHandler

Delphi declaration
```
DeltaHandler: TkbmCustomDeltaHandler FDeltaHandler
SetDeltaHandler;
```

Set this to point to the deltahandler you want to handle the resolving operations.
To activate the resolving, call the *Resolve* method of the deltahandler.
The memory table does never itself automatically issue a resolve.

DesignActivation

Delphi declaration

```
DesignActivation: boolean FDesignActivation
FDesignActivation;
```

Set to true (default) if the table should be activated at runtime if its active property was set to true at designtime.

DetailFields

Delphi declaration

```
DetailFields: FDetailFieldNames SetDetailFields;
```

In normal TTable master/detail relations, the detail fields are the same as the current index.
Thus it is not easily possible to have one set of fields for ordering and another set for M/D relations.
That structure is ofcourse supported by the memory table, but additionally, you can specify the detail fields directly, and setup any index you would like to.
If you specify DetailFields, those fields will be used for M/D instead of the IndexFieldNames.
If not, IndexFieldNames will be used like aa normal TTable.

Delphi Example

mt.DetailFields:='MasterRefField1;MasterRefField2';

EnableIndexes

Delphi declaration

```
EnableIndexes: boolean FEnableIndexes FEnableIndexes
true;
```

Set this property to enable or disable updating indexes.
Notice that while the indexes are disabled, you wont be able to see records being inserted, and if you delete records and then try to reach them through an index unpredictable errors can occur.
Thus after disabling the indexes, doing your operation, remember to reenable them and call UpdateIndexes to get them back up to date before using them.
Its mainly supported for performance reasons.

EnableVersioning

Delphi declaration

```
EnableVersioning: boolean GetEnableVersioning
SetEnableVersioning false;
```

Set this to true to enable versioning. The purpose of versioning is to maintain backup versions of records whenever the record is modified or deleted.
This is used by transactions and deltahandlers .

FilterOptions

Delphi declaration
```
FilterOptions: TFilterOptions FFilterOptions
FFilterOptions;
```

Filter options for the current filter.

FormFormat

Delphi declaration
```
FormFormat: TkbmCustomStreamFormat FFormFormat
FFormFormat;
```

This is the streamformat used when storing the memtable contents in a form ressource file.

IgnoreAutoIncPopulation

Delphi declaration
```
IgnoreAutoIncPopulation: boolean
FIgnoreAutoIncPopulation FIgnoreAutoIncPopulation;
```

If set to true, ftAutoInc fields will not be given any new unique values on record insert/append.

IgnoreReadOnly

Delphi declaration
```
IgnoreReadOnly: boolean FIgnoreReadOnly
FIgnoreReadOnly false;
```

Set this property to true, if you temporarely want to update a memory table which is actually marked as read only.

See also:
ReadOnly

IndexByName

Delphi declaration
```
IndexByName[IndexName: ]: TkbmIndex GetIndexByName;
```

Return an index structure or nil depending if any indexes could be found with the given name.
You can f.ex. use this to disable or enable a specific index.

See also:
AddFilteredIndex AddIndex

IndexDefs

Delphi declaration
```
IndexDefs: TIndexDefs FIndexDefs SetIndexDefs;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.IndexDefs,1) TTable.IndexDefs .

Indexes

Delphi declaration
```
Indexes: TkbmIndexes GetIndexes;
```

Return a TkbmIndexes instance which can be used for accessing all index instances on the memory table.
Generally do not fool around with this structure unless you know what you are doing.

IndexFieldNames

Delphi declaration
```
IndexFieldNames: FIndexFieldNames SetIndexFieldNames;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.IndexFieldNames,1) TTable.IndexFieldNames
Setting the property to '' will select the internal row order index.

IndexFields

Delphi declaration
```
IndexFields[: Integer]: TField GetIndexField
SetIndexField;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.IndexFields,1) TTable.IndexFields

IndexName

Delphi declaration
```
IndexName: FIndexName SetIndexName;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.IndexName,1) TTable.IndexName
Setting this to nil will select the internal row order index.

IsDataModified

Delphi declaration
```
IsDataModified: boolean GetIsDataModified
SetIsDataModified;
```

Return true if data in the table has been modified.

See also:
IsFieldModified

IsFieldModified

Delphi declaration
```
IsFieldModified[i: integer]: boolean GetModifiedFlags;
```

Return true if the field has been modified.
Notice that when the *Post* or *Cancel* method is invoked, will the field modification flags be reset.
Thus the flags are only valid between an *Edit*/*Insert*/*Append* and *Post*/*Cancel*/*Prior*/*Next*/*First*/*Last* operation.
The index must be in the range 0 to *FieldCount*-1.

See also:
IsDataModified

IsVersioning

Delphi declaration
```
IsVersioning: boolean GetIsVersioning;
```

Return true if versioning is enabled.

See also:
EnableVersioning

LanguageID

Delphi declaration
```
LanguageID: integer GetLanguageID SetLanguageID;
```

Set in combination with *SortID* and *SubLanguageID* to specify the Locale that the table should work under.

Setting any of *LanguageID*, *SubLanguageID* or *SortID* will recalculate the *LocaleID* property.
Setting the *LocaleID* property will recalculate the other ID's.
This way, indexes and searches can perform as one would expect to according to different countries and languages sort rules. See

LoadCount

Delphi declaration
```
LoadCount: integer FLoadCount FLoadCount;
```

Return the number of records loaded during last load operation. LoadFromStream, LoadFromFile, LoadFromBinaryStream, LoadFromBinaryFile, AllData, CommaText and LoadFromDataset all sets this value during load.

See also:
AllData CommaText LoadedCompletely ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryFile LoadFromBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryStream LoadFromBinaryStream LoadFromDataSet LoadFromFile LoadFromStream LoadLimit

LoadedCompletely

Delphi declaration
```
LoadedCompletely: boolean FLoadedCompletely
FLoadedCompletely;
```

Return true if the last load operation loaded all available records. Return false if there was still records left to load, and the LoadLimit was reached.

See also:
LoadCount

LoadLimit

Delphi declaration
```
LoadLimit: integer FLoadLimit FLoadLimit -1;
```

Set this to the maximum number of records to load during a load operation.
Set to -1 if to load all.

See also:
LoadCount LoadedCompletely

LocaleID

Delphi declaration
```
LocaleID: TkbmLocaleID GetLocaleID SetLocaleID;
```

Set in combination with *SortID* and *SubLanguageID* to specify the Locale that the table should work under.
Setting any of *LanguageID*, *SubLanguageID* or *SortID* will recalculate the *LocaleID* property.
Setting the *LocaleID* property will recalculate the other ID's.
This way, indexes and searches can perform as one would expect to according to different countries and languages sort rules. See

MasterFields

Delphi declaration
```
MasterFields: GetMasterFields SetMasterFields;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.MasterFields,1) TTable.MasterFields

MasterSource

Delphi declaration
```
MasterSource: TDataSource GetDataSource SetDataSource;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.MasterSource,1) TTable.MasterSource .

OnDecompressBlobStream

Delphi declaration
property OnDecompressBlobStream: TkbmOnDecompress read FOnDecompressBlobStream write FOnDecompressBlobStream;

Write an eventhandler here if you want to de compress blobs in memory. Use it in combination with OnCompressBlobStream.

OnDecompressLoad

Delphi declaration
property OnDecompressLoad: TkbmOnDecompress read FOnDecompressLoad write FOnDecompressLoad;

Write an event handler here if you want to de compress the data before its loaded from an external storage.
Use the event in combination with the OnCompressSave event.

OverrideActiveRecordBuffer

Delphi declaration

```
OverrideActiveRecordBuffer: PkbmRecord
FOverrideActiveRecordBuffer
FOverrideActiveRecordBuffer;
```

Internal property which allows for specifying a specific record structure as the current record.
Generally dont use this directly.

Performance

Delphi declaration

```
Performance: TkbmPerformance GetPerformance
SetPerformance mtpfFast;
```

Set this to change the way physical records are stored in memory. You can only change this while the table (and all other attached tables) is closed.

Persistent

Delphi declaration

```
Persistent: boolean FPersistent FPersistent false;
```

Set this to true to automatically load and save the contents of the memory table during table open/close, in the file specified in the property PersistentFile using the save format specified in PersistentSaveFormat and with the save options as specified in ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.PersistentSaveOptions PersistentSaveOptions .

See also:
LoadPersistent PersistentBackup PersistentBackupExt PersistentFile PersistentFormat PersistentSaved PersistentSaveFormat SavePersistent

PersistentBackup

Delphi declaration

```
PersistentBackup: boolean FPersistentBackup
FPersistentBackup;
```

Set to true if a backup file should be made before a persistent save is made.

See also:
Persistent PersistentBackupExt PersistentFile PersistentFormat PersistentSaved PersistentSaveFormat ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.PersistentSaveOptions PersistentSaveOptions

PersistentBackupExt

Delphi declaration
```
PersistentBackupExt: FPersistentBackupExt
FPersistentBackupExt;
```

Set to the file extension you want a backup file to have. Remember to put a dot infront.
Eg: PersistentBackupExt := '.BAK';

See also:
Persistent PersistentBackup PersistentFile PersistentFormat PersistentSaved PersistentSaveFormat ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.PersistentSaveOptions PersistentSaveOptions

PersistentFile

Delphi declaration
```
PersistentFile: TFileName FPersistentFile
FPersistentFile;
```

Set to the name of the file (incl. extenstion and path if needed) in which the memory table should be saved to and loaded from if persistency is enabled.

See also:
Persistent PersistentBackup PersistentBackupExt PersistentFormat PersistentSaved PersistentSaveFormat ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.PersistentSaveOptions PersistentSaveOptions

PersistentFormat

Delphi declaration
```
PersistentFormat: TkbmCustomStreamFormat
FPersistentFormat FPersistentFormat;
```

The streamformat to use when persistencing the memtable.

See also:
Persistent PersistentBackup PersistentBackupExt PersistentFile PersistentSaved PersistentSaveFormat

PersistentSaved

Delphi declaration
```
PersistentSaved: boolean FPersistentSaved
FPersistentSaved false;
```

Returns true if the memtable havnt been saved using the persistency methods since it was loaded.

See also:
Persistent PersistentBackup PersistentBackupExt PersistentFile PersistentFormat PersistentSaveFormat

PersistentSaveFormat

Delphi declaration
property PersistentSaveFormat: TkbmMemTablePersistentSaveFormat read FPersistentSaveFormat write FPersistentSaveFormat default mtsfBinary;

Set to the format you want to save/load the persistent file to/from.

See also:
Persistent PersistentBackup PersistentBackupExt PersistentFile PersistentFormat PersistentSaved

ProgressFlags

Delphi declaration
```
ProgressFlags: TkbmProgressCodes FProgressFlags
FProgressFlags;
```

Set to the flags you want to trigger the OnProgress event.

Delphi Example
Example 1 - OnProgress should only trigger during a save or a load.
mt.ProgressFlags:=[mtpcLoad,mtpcSave];

RangeActive

Delphi declaration
```
RangeActive: boolean GetRangeActive;
```

Return true if a range is active.

See also:
ApplyRange SetRange

RangeIgnoreNullKeyValues

Delphi declaration

```
RangeIgnoreNullKeyValues: boolean
FRangeIgnoreNullKeyValues FRangeIgnoreNullKeyValues
true;
```

A boolean property which controls if key values that are null should be ignored in range comparisons.
Default true.

ReadOnly

Delphi declaration

```
ReadOnly: boolean FReadOnly FReadOnly false;
```

Set to true if you want the memory table to be read only.
You can override the read only status for brief periods using the IgnoreReadOnly property.
Default is false.

RecalcOnFetch

Delphi declaration

```
RecalcOnFetch: boolean FRecalcOnFetch FRecalcOnFetch
true;
```

Standard ERROR: INVALID LINK: !Alink(TDataset,1) TDataset behaviour is to calculate all calculated fields on every fetch, that means that every time a record is fetched, a complete recalculation of all calculated fields in that record is made.
Often there is no reason to make a recalculation unless some changes has been made in any other of the fields in the record.
Default is true.

See also:

OnCalcFields RecalcOnIndex

RecalcOnIndex

Delphi declaration

```
RecalcOnIndex: boolean FRecalcOnIndex FRecalcOnIndex
false;
```

RecalcOnIndex determines if any recalculations should be made on a record if its searched on. If you never search directly on values of calculated fields, its safe to set this to false.

Default is true.

See also:

OnCalcFields RecalcOnFetch

RecordTag

Delphi declaration

```
RecordTag: longint GetRecordTag SetRecordTag;
```

Set/get this to put a 64 bit value into the current record. The value can be used for your own purpose, as the memory table will not try to change it or use it for anything.

SaveCount

Delphi declaration

```
SaveCount: integer FSaveCount FSaveCount;
```

Returns the number of records saved successfully.

SavedCompletely

Delphi declaration

```
SavedCompletely: boolean FSavedCompletely
FSavedCompletely;
```

Returns true if the save was complete. I.e. all records in the current recordset was saved.

SaveLimit

Delphi declaration

```
SaveLimit: integer FSaveLimit FSaveLimit -1;
```

Controls how many records to save.
Default -1 which means all.

SortFields

Delphi declaration
```
SortFields: FSortFieldNames FSortFieldNames;
```

Set to specify the fields which should be used for sorting on using either Sort or SortDefault .
See also SortOn .

SortID

Delphi declaration
```
SortID: integer GetSortID SetSortID;
```

Set in combination with *SortID* and *SubLanguageID* to specify the Locale that the table should work under.
Setting any of *LanguageID*, *SubLanguageID* or *SortID* will recalculate the *LocaleID* property.
Setting the *LocaleID* property will recalculate the other ID's.
This way, indexes and searches can perform as one would expect to according to different countries and languages sort rules. See

SortOptions

Delphi declaration
```
SortOptions: TkbmMemTableCompareOptions FSortOptions
FSortOptions;
```

Set to specify the options which control the sort procedure for SortDefault .
See also SortOn .

Standalone

Delphi declaration
```
Standalone: boolean GetStandalone SetStandalone false;
```

Set to true if you are absolutely sure no other tables should attach to this.
It will gain you a few percents of speed, since all internal thread locking mechanisms will be disabled.
Default false.

See also:

AttachedTo

StatusFilter

Delphi declaration

```
StatusFilter: TUpdateStatusSet FStatusFilter
SetStatusFilter;
```

Set this if you only want to see records having a certain status.

Delphi Example

Example 1 - Show only inserted or modified records.

mt.StatusFilter:=[usInserted,usModified];

StoreDataOnForm

Delphi declaration

```
StoreDataOnForm: boolean FStoreDataOnForm
FStoreDataOnForm false;
```

Set to true during designtime only if you want to store the current data in the table on the form file itself.
To use this, you have to use the memory table designer which you can get to by right clicking the memory table component on your form at designtime, and create a structure in the memory table.
Then make sure to load some data into the table, either from a file or from another *TDataset*.
Then your memory table will be *Active*. Before setting it inactive, set *StoreDataOnForm* to true and save your project.
Next time you open your project, or run the program and open the memory table, it will automatically be populated with the data from the form. No external files needed.

This can f.ex. be used to ship an application with default data build right into the app. When the application is then run, you check if the data file is existing. If its not, you simply open the memory table and voila, you have your data there. Then save the data to the datafile.
If the data file was allready existing, you set *StoreDataOnForm* to false before opening the memory table, and load the data from the data file.

See also:
DefineProperties ReadData WriteData

SubLanguageID

Delphi declaration
```
SubLanguageID: integer GetSubLanguageID
SetSubLanguageID;
```

Set in combination with *SortID* and *SubLanguageID* to specify the Locale that the table should work under.
Setting any of *LanguageID*, *SubLanguageID* or *SortID* will recalculate the *LocaleID* property.
Setting the *LocaleID* property will recalculate the other ID's.
This way, indexes and searches can perform as one would expect to according to different countries and languages sort rules. See

TableState

Delphi declaration
```
TableState: TkbmState FState FState;
```

Check this to see what is happening in the table right now.
This can be nice to know information during event handlers or for other threads.

See also:
Progress ProgressFlags

TransactionLevel

Delphi declaration
```
TransactionLevel: integer GetTransactionLevel;
```

Return the current transaction level.
If no transactions has been started, the value will be 0.

See also:
Commit Rollback StartTransaction

Version

Delphi declaration
```
Version: GetVersion FDummyStr;
```

Return a string stating the version of the memory table.

VersioningMode

Delphi declaration

```
VersioningMode: TkbmVersioningMode GetVersioningMode
SetVersioningMode mtvm1SinceCheckPoint;
```

Set to change how many backup records will be retained during versioning.

Protected Properties/Variables

FormatSettings

Delphi declaration

```
FormatSettings: TFormatSettings FFormatSettings
FFormatSettings;
```

IsFiltered

Delphi declaration

```
IsFiltered: boolean FIsFiltered;
```

Returns true if the table is currently filtered.

Public Methods/Functions

__CalcFieldsSize

Delphi declaration

```
__CalcFieldsSize: integer;
```

Not for general use.

__ClearBuffers

Delphi declaration

```
__ClearBuffers;
```

Not for general use.

__ClearCalcFields

Delphi declaration

```
__ClearCalcFields(Buffer: PkbmRecord);
```

Not for general use.

__GetCalcFields

Delphi declaration

```
__GetCalcFields(Buffer: PkbmRecord);
```

Not for general use.

__RestoreState

Delphi declaration

```
__RestoreState(Value: TDataSetState);
```

Not for general use.

__SetBlockReadSize

Delphi declaration

```
__SetBlockReadSize(Value: Integer);
```

Not for general use.

__SetTempState

Delphi declaration

```
__SetTempState(Value: TDataSetState): TDataSetState;
```

Not for general use.

AddFilteredIndex

Delphi declaration

```
AddFilteredIndex(, Fields: ; Options: TIndexOptions;
AUpdateStatus: TUpdateStatusSet; Filter: ;
FilterOptions: TFilterOptions; FilterFunc:
TkbmOnFilterIndex=): TkbmIndex; ;
AddFilteredIndex(, Fields: ; Options: TIndexOptions;
Filter: ; FilterOptions: TFilterOptions; FilterFunc:
TkbmOnFilterIndex=): TkbmIndex; ;

AddFilteredIndex(, Fields: ; Options: TIndexOptions;
AUpdateStatus: TUpdateStatusSet; Filter: ;
FilterOptions: TFilterOptions; FilterFunc:
TkbmOnFilterIndex=): TkbmIndex; ;
AddFilteredIndex(, Fields: ; Options: TIndexOptions;
Filter: ; FilterOptions: TFilterOptions; FilterFunc:
TkbmOnFilterIndex=): TkbmIndex; ;
```

Add a filtered index to the memory table.
A filtered index works exactly like a normal index, except that a filtered index gives you the opportunity to select which records should be part of the index and which shouldnt.
Like any other indexes, filtered indexes will be updated whenever you insert, modify or delete records if the index is enabled (which it is by default).
When you switch to the filtered index, you will only see the records which match the filter criterias specificed with the OnFilterIndex event handler and the optional Filter string (only D5/BCB5 or newer environments).
The index can have a sort order just like any other index.

Name should be the name you will use to access or select the index. Required.
Fields should contain one or more fieldnames seperated with semicolon (;) and with optional field modifiers . Required.
Options should contain an array of index wide options. Can be an empty array [].
UpdateStatus should contain a set of updatestatus flags. By setting this correctly, even deleted records can be seen (when using versioning).
Filter can contain a filter expression or simply an empty string if no expression filter is required.
FilterOptions specifies options for the filter.

Remember to specify either a filter expression (D5 or BCB5 or newer only) or an OnFilterIndex event handler.

See also:
Field modifiers Filter expressions AddIndex

AddFilteredIndex2

Delphi declaration

```
function AddFilteredIndex2(, Fields: ; Options:
TIndexOptions; AUpdateStatus: TUpdateStatusSet;
Filter: ; FilterOptions: TFilterOptions; FilterFunc:
TkbmOnFilterIndex): TkbmIndex;
```

Only on pre Delphi 5 compilers.
See AddFilteredIndex for more info.

See also:
TkbmCustomMemTable.AddFilteredIndex

AddIndex

Delphi declaration

```
AddIndex(, Fields: ; Options: TIndexOptions;
AUpdateStatus: TUpdateStatusSet): TkbmIndex; ;
AddIndex(, Fields: ; Options: TIndexOptions):
TkbmIndex; ;

AddIndex(, Fields: ; Options: TIndexOptions;
AUpdateStatus: TUpdateStatusSet): TkbmIndex; ;
AddIndex(, Fields: ; Options: TIndexOptions):
TkbmIndex; ;
```

Add an index to the memory table.
Indexes can be used to reorder the logical record order of a memorytable, and to speed up search/locate operations.
You can add as many indexes as you would like.
Indexes doesnt contain any data from the records, but only pointers to the physical records. Likewise, the physical records are never actually sorted or rearranged when you create an index. Only the pointers specific for the index are rearranged.
There is always existing one internal index, the row order index. The row order index is used to keep the insertion order of the records added to the memory table and it cannot be deleted. If no other index is selected, the row order index will be used.

Name should be the name you will use to access or select the index. Required.
Fields should contain one or more fieldnames seperated with semicolon (;) and with optional field modifiers_ ERROR: Unknown Link _0 . Required.
Options should contain an array of index wide options. Can be an empty array [].
UpdateStatus should contain a set of updatestatus flags. By setting this correctly, even deleted records can be seen (when using versioning).

See also:
Field modifiers AddFilteredIndex

AddIndex2

Delphi declaration
```
AddIndex2(, Fields: ; Options: TIndexOptions;
AUpdateStatus: TUpdateStatusSet): TkbmIndex;
```

This is an alternative for AddIndex for pre Delphi 5 compilers.
See Addindex for more info.

See also:
TkbmCustomMemTable.AddIndex

Aggregate

Delphi declaration
```
Aggregate(AAggregateFields: ): variant; ;
```

ApplyRange

Delphi declaration
```
ApplyRange;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.ApplyRange,1) TTable.ApplyRange .

See also:
CancelRange EditRangeEnd EditRangeStart SetRange SetRangeEnd SetRangeStart

AssignRecord

Delphi declaration
```
AssignRecord(Source, Destination: TDataSet);
```

Copies the contents of the current record in the Source dataset to the current record of the Destination dataset.
Only fields with same field names are copied. The order of the fields doesnt matter.

BookmarkValid

Delphi declaration
```
BookmarkValid(Bookmark: TBookmark): boolean; ;
```

Call BookmarkValid to determine if a specifed bookmark is currently assigned a value. Bookmark specifies the bookmark to test. BookmarkValid returns True if a bookmark is valid. Otherwise, it returns False.

BuildFieldList

Delphi declaration

```
BuildFieldList(Dataset: TDataset; List: TkbmFieldList;
FieldNames: ; AggregateFieldNaming: boolean=false);
```

Generates a list of TFields from a FieldNames string for the specified dataset.
Primarely for internal use.

CancelRange

Delphi declaration

```
CancelRange;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.CancelRange,1) TTable.CancelRange

See also:
ApplyRange EditRangeEnd EditRangeStart SetRange SetRangeEnd SetRangeStart

CheckPoint

Delphi declaration

```
CheckPoint(AForce: boolean=false; AMarkAsInserted:
boolean=false);
```

Calling CheckPoint will remove all existing old versions of records. It can f.ex. be used to determine from which point in history a delta handler should resolve records.

See also:
TkbmCustomDeltaHandler Commit Rollback StartTransaction

CheckPointRecord

Delphi declaration

```
CheckPointRecord(RecordIndex: integer; AForce:
boolean=false; AMarkAsInserted: boolean=false):
TUpdateStatus;
```

Performs a checkpoint on a specific record.

All old versions are removed and only the newest version will be retained.

ClearModified

Delphi declaration

```
ClearModified;
```

Clear out modification flags for all fields.
When a field is altered, the modification flag will be set.
By the FieldModified property its possible to inspect if a field has been modified.
Notice the flags are automatically cleared when the record has been posted.

Commit

Delphi declaration

```
Commit; ;
```

Commit a transaction.
ne Transactions are used for atomic updates of multiple seperate record operations.
F.ex. you can start a transaction (StartTransaction), modify a few records, delete some others and perhaps insert some too, then you figure out that something was not quite right, and you would like to undo all the modifications.
This is done using Rollback . If instead you decided to keep all the changes made since the start of the transaction call Commit to make the updates/changes stick.

See also:
EnableVersioning Rollback StartTransaction TransactionLevel VersioningMode

CompareBookmarks

Delphi declaration

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

Return <0 if the record Bookmark1 is pointing at is before the record pointed to by Bookmark2 using the current index.
Return >0 if the record Bookmark1 is pointing at is after the record pointed to by Bookmark2 using the current index.
And return 0 if the bookmarks are pointing to the same record.

See also:

ERROR: INVALID LINK: TDataset.CompareBookmarks TDataset.CompareBookmarks

CopyFieldProperties

Delphi declaration

```
CopyFieldProperties(Source, Destination: TField);
```

Copy all the following properties from the Source field to the Destination field:

EditMask, DisplayWidth, DisplayLabel, Required, ReadOnly, Visible, DefaultExpression, Alignment, ProviderFlags (D5/BCB5 or newer only)

Its used internally by LoadFromDataset and CreateTableAs .

CopyFieldsProperties

Delphi declaration

```
CopyFieldsProperties(Source, Destination: TDataSet);
```

Copy properties from all fields in the source dataset to the destination dataset.
Used internally by CreateTableAs and LoadFromDataset .

See also:

CopyFieldProperties

CopyRecord

Delphi declaration

```
CopyRecord(Source, Destination: Integer): Boolean;
```

CopyRecords

Delphi declaration

```
CopyRecords(Source, Destination: TDataSet; Count:
longint; IgnoreErrors: boolean; Mapping: ;
WideStringAsUTF8: boolean): longint; ;
```

Copy Count records from the Source dataset to the Destination dataset.

Filters, ranges etc. on the source dataset are respected.
Only fields with the same name in both datasets are copied.

You can interfere in the copying operation through the **OnLoadField**, **OnLoadRecord**, **OnSaveField** and **OnSaveRecord** events.
If the specified number of record was loaded successfully, the property **LoadedCompletely** will be **true**, otherwise **false**.

Returns the number of records copied.

See also:
LoadedCompletely OnLoadField OnLoadRecord OnSaveField OnSaveRecord

Create

Delphi declaration
```
Create(AOwner: TComponent); ;
```

The constructor ot TkbmCustomMemTable.
You can construct memory tables in runtime if you so wish.
Usually you should remember to specify nil as AOwner amd handle freeing the memory table yourself at the appropriate time. If you specify another component as owner of the memory table component instance, you will not be able to free it by yourself if you so would like.

Delphi Example

Example 1 - Create a memory table at run time.

```
// Create the memorytable object, and set some properties.
kbmMemTable1 := TkbmMemTable.Create(Self);                    // Owner is Self. It will be auto-destroyed when self
                                                              // is destroyed..
kbmMemTable1.SortOptions := [];                                                    // Optional.
kbmMemTable1.PersistentFile := 'testfil.txt';                                      // Optional.
kbmMemTable1.OnCompressSave := kbmMemTable1CompressSave;                           // Optional.
kbmMemTable1.OnDecompressLoad := kbmMemTable1DecompressLoad;
        // Optional.
kbmMemTable1.OnCompressBlobStream := kbmMemTable1CompressBlobStream;
                // Optional.
kbmMemTable1.OnDecompressBlobStream :=
kbmMemTable1DecompressBlobStream;     // Optional.
kbmMemTable1.OnCalcFields := MemTable1CalcFields;                                  // Optional.
kbmMemTable1.OnFilterRecord := kbmMemTable1FilterRecord;
        // Optional.
kbmMemTable1.MasterSource := Nil;                                                  // Optional.

// Define the fields.                          // You should have some fields define to make any sense.
kbmMemTable1.FieldDefs.Clear;                       // We dont need this line, but it does not hurt either.
kbmMemTable1.FieldDefs.Add('Period', ftInteger, 0, False);
kbmMemTable1.FieldDefs.Add('Value', ftInteger, 0, False);
kbmMemTable1.FieldDefs.Add('Color', ftInteger, 0, False);
kbmMemTable1.FieldDefs.Add('Calc', FtString, 20, False);
kbmMemTable1.FieldDefs.Add('Date', ftDate, 0, False);

// Define index fields                                                    // Optional.
kbmMemTable1.IndexDefs.Add('Index1','Value',[]);

// Finally create the table according to definitions.                     // Required.
```

kbmMemTable1.CreateTable;

// Since this is a run-time created one, we have to assign the following here.
DataSource1.DataSet := kbmMemTable1;

// Optionel. IndexFields and SortFields must be assigned AFTER CreateTable
kbmMemTable1.IndexFields := 'Value';
kbmMemTable1.SortFields := 'Value';

See also:
Destroy

CreateBlobStream

Delphi declaration

```
CreateBlobStream(Field: TField; Mode:
TBlobStreamMode): TStream; ;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.CreateBlobStream,1) TTable.CreateBlobStream

Notice:

An easier and better way to get and set blob contents in a field from a stream is to use the TBlobField.SaveToStream and TBlobField.LoadFromStream methods.

CreateFieldAs

Delphi declaration

```
CreateFieldAs(Field: TField; AForceAsData:
boolean=false): TField;
```

Creates a new TField in the memory table as an exact definition copy of another field from another dataset.
Its mainly used by LoadFromDataset and CreateTableAs .

CreateIndexes

Delphi declaration

```
CreateIndexes;
```

Internal routine which actually creates the indexes defined in the index definitions.

See also:
ERROR: INVALID LINK: TDataset.IndexDefs TDataset.IndexDefs

CreateTable

Delphi declaration
```
CreateTable;
```

Creates the fields defined in FieldDefs and the Indexes defined in IndexDefs.
Similar workings as ERROR: INVALID LINK:
!Alink(TTable.CreateTable,1) TTable.CreateTable

See also:
Active

CreateTableAs

Delphi declaration
```
CreateTableAs(Source: TDataSet; CopyOptions:
TkbmMemTableCopyTableOptions);
```

Copies the complete fields definitions of another dataset to the memorytable.
Specify what to copy and how using the CopyOptions.

DeleteIndex

Delphi declaration
```
DeleteIndex(: );
```

Delete the index specified by the name.

See also:
AddFilteredIndex AddIndex

DeleteRange

Delphi declaration
```
DeleteRange(AFields: ; StartValues, EndValues:
variant): integer;
```

DeleteRecords

Delphi declaration
```
DeleteRecords: integer; ;
```

DeleteTable

Delphi declaration
```
DeleteTable;
```

Removes all field from the memory table after its closed.
Primarely used for full TTable compability.

See also:
CreateTable

Destroy

Delphi declaration
```
Destroy; ;
```

The destructor called whenever the memory table is freed.
Dont call this directly. Instead use the Free method.
Notice that if you have created the table with an owner and not with nil, you should not yourself free the table, since
the owner will do that for you. See CreateTable .

If you anyway want to free the table even if it was created with an owner, you need to check the Delphi Examples.

Delphi Example
Example 1 - Freeing when table created with owner.

```
kbmMemTable1.Parent:=nil;          // Very important to avoid Application Violations.

kbmMemTable1.Free;
kbmMemTable1:=nil;                 // Nice to do to make sure you dont use an invalid pointer later on.
                                   // You can also use the new FreeAndNil method of D5 to combine the
                                   // free and nil setting.
```

See also:
Create

DestroyIndexes

Delphi declaration
```
DestroyIndexes;
```

Internal support method which will remove all indexes defined on the table, but leave the definitions alone.

EditKey

Delphi declaration
```
EditKey;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.EditKey,1) TTable.EditKey

See also:
GotoKey SetKey

EditRangeEnd

Delphi declaration
```
EditRangeEnd;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.EditRangeEnd,1) TTable.EditRangeEnd

See also:
ApplyRange EditRangeStart

EditRangeStart

Delphi declaration
```
EditRangeStart;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.EditRangeStart,1) TTable.EditRangeStart

See also:
ApplyRange EditRangeEnd

EmptyTable

Delphi declaration
```
EmptyTable;
```

Empties the contents of the table and all attached tables.

Exists

Delphi declaration
```
Exists: boolean;
```

Return true if the file given in the property PersistentFile exists, else false.

Extract

Delphi declaration

```
Extract(AExtractFields: ; AStringList: TStrings;
AFormat: =''): integer;
```

FilterRecord

Delphi declaration

```
FilterRecord(ARecord: PkbmRecord; ForceUseFilter:
boolean): Boolean;
```

A method that returns true if the record is visible according to all the filter/masterdetail/range settings defined.
Otherwise it returns false.

FindFieldInList

Delphi declaration

```
FindFieldInList(List: TkbmFieldList; FieldName: ):
TField;
```

Internal function which returns the field from a list of fields which matches the name given in FieldName.
If no field with the given field name is found, **nil** is returned.

FindKey

Delphi declaration

```
FindKey(KeyValues: ): Boolean;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.FindKey,1) TTable.FindKey

See also:
EditKey GotoKey SetKey

FindNearest

Delphi declaration

```
FindNearest(KeyValues: ): Boolean;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.FindNearest,1) TTable.FindNearest

GetAggregateFieldName

Delphi declaration

```
GetAggregateFieldName(AFieldName: ; AOptions:
TkbmifoOptions): ; ;
```

GetAggregateFieldOption

Delphi declaration

```
GetAggregateFieldOption(AAggregateFunction: ):
TkbmifoOptions; ;
```

GetExtractFieldOption

Delphi declaration

```
GetExtractFieldOption(AExtractOption: ):
TkbmifoOptions; ;
```

GetFieldData

Delphi declaration

```
GetFieldData(Field: TField; Buffer:
PGetFieldDataBuffer): Boolean; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support function which in Delphi 3/BCB 3 is protected and in Delphi 4+ / BCB 4+ is public available.
The purpose of the function is to return the raw data of a specific field from the current record.
Buffer should point to an array of bytes big enough to recieve the complete contents of the field.
If **Buffer** is **nil** then no data will be returned.
The function return *true* if the field contents is *not null*, else *false*.
Generally do not use this function directly. Instead use the normal TDataset.As..... functions.

See also:
SetFieldData ERROR: INVALID LINK: TDataset.GetFieldData
TDataset.GetFieldData

GetRows

Delphi declaration

```
GetRows(Rows: Integer; Start: Variant; Fields:
Variant): Variant;
```

Function to generate a variant array and populate it with the contents of the memory table starting with a record defined either with a ERROR: INVALID LINK: !Alink(TTable.GetBookmark,1) bookmark or one of the following constants in the *Start* parameter:
kbmBookMarkFirst Start with the first record using current index and filters etc.
kbmBookMarkLast Start with last record using current index and filters etc. Not very usefull at all!
kbmBookmarkCurrent Start with the current record.

Rows is a simple count for how many rows should be copied to the variant array.
Rows can be set to **kbmGetRowsRest** if all remaining records in the memory table should be copied.

Fields specifies which fields to copy. The fields can be specified in several ways:
varEmpty Copy all fields.
'Fieldname' Copy the field specified with fieldname.
['Fieldname1',..'FieldNameN'] Copy the fields specified.
1 Copy the first physical field in the table according to the FieldDefs. See ERROR: INVALID LINK: !Alink(TDataset.FieldNo,1) TDataset.FieldNo
[1,3,5] Copy field 1, 3 and 5.

The resulting variant array can be accessed as:

array[n,RowNo] where n is a number indicating a field (0..), and RowNo is a number of the row in the array (0..)

GetVersionCount

Delphi declaration

```
GetVersionCount: integer;
```

Function that returns the number of versions of the current record.
If versioning is not enabled , the result will be 1.

If versioning is enabled and versioning mode is **mtvm1SinceCheckPoint**, the result will be 1 or 2.
If versioning is enabled and versioning mode is **mtvmAllSinceCheckPoint**, the result will be 1 or more.

It can be used in cooporation with GetVersionFieldData and GetVersionStatus to obtain the field value and status of any version of a record.

See also:
TkbmCustomDeltaHandler

GetVersionFieldData

Delphi declaration

```
GetVersionFieldData(Field: TField; Version: integer):
variant;
```

Returns the data of the specified field and version of the current record as a variant.
Version should be 0 to GetVersionCount -1
Version=**0** refers to the newest version
Version=**1** refers to the 2nd. newest if VersioningMode =**mtvmAllSinceCheckPoint** or to the oldest version if VersioningMode =**mtvm1SinceCheckPoint**
Version=n referes to the nth version of the current record.

See also:
TkbmCustomDeltaHandler

GetVersionStatus

Delphi declaration

```
GetVersionStatus(Version: integer): TUpdateStatus;
```

Return the update status of the specified version of the current record. Using this function you can determine if a delete, insert or modification was the reason for that specific version.
Version=0 to GetVersionCount -1

See also:
TkbmCustomDeltaHandler

GotoCurrent

Delphi declaration

```
GotoCurrent(DataSet: TkbmCustomMemTable); ;
```

GotoKey

Delphi declaration
```
GotoKey: boolean;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.GotoKey,1) TTable.GotoKey

See also:
EditKey FindKey SetKey

GotoNearest

Delphi declaration
```
GotoNearest: boolean;
```

GroupBy

Delphi declaration
```
GroupBy(ADestDataset: TkbmCustomMemTable;
ASourceGroupFieldList: TkbmFieldList;
ASourceAggregateFieldList, ADestAggregateFieldList:
TkbmFieldList); ; ;
GroupBy(ADestDataset: TkbmCustomMemTable;
AGroupFields: ; AAggregateFields: ); ; ;

GroupBy(ADestDataset: TkbmCustomMemTable;
ASourceGroupFieldList: TkbmFieldList;
ASourceAggregateFieldList, ADestAggregateFieldList:
TkbmFieldList); ; ;
GroupBy(ADestDataset: TkbmCustomMemTable;
AGroupFields: ; AAggregateFields: ); ; ;
```

IndexFieldCount

Delphi declaration
```
IndexFieldCount: Integer;
```

Return the number of fields in the current index.

IsFieldListsBegin

Delphi declaration

```
IsFieldListsBegin(List1, List2: TkbmFieldList;
ASameCase: boolean): boolean;
```

Function that return true if the fields in list2 also represents the first fields of list1.

IsFieldListsEqual

Delphi declaration

```
IsFieldListsEqual(List1, List2: TkbmFieldList;
ASameCase: boolean): boolean;
```

Function that return true if the two list of fields are 100% identical, otherwise it will return false.

IsMasterDetailActive

Delphi declaration

```
IsMasterDetailActive: Boolean;
```

IsSequenced

Delphi declaration

```
IsSequenced: Boolean; ;
```

Will return true if no filtering (except for filtered indexes) is enabled.

LoadFromDataSet

Delphi declaration

```
LoadFromDataSet(Source: TDataSet; CopyOptions:
TkbmMemTableCopyTableOptions; Mapping: =''); ;
```

Load the contents of another dataset into the memory table.
CopyOptions specifies how the operation should happen.
Only fields with same field names and which datatypes are supported by kbmMemTable will be copied. All other fields will be ignored.
If no table structure should be copied, the data will be appended to the currently available records.

See also:
ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryFile LoadFromBinaryFile ERROR: INVALID LINK:

kbmMemTable.TkbmCustomMemTable.LoadFromBinaryStream LoadFromBinaryStream LoadFromFile LoadFromStream SaveToDataSet

LoadFromFile

Delphi declaration

```
LoadFromFile(FileName: );
```

Populate the memory table with the data from a file formatted according to the DefaultFormatter settings.ne See ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.InternalLoadFromStream InternalLoadFromStream

See also:

ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.InternalLoadFromBinaryStream InternalLoadFromBinaryStream ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.InternalLoadFromStream InternalLoadFromStream ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.InternalSaveToBinaryStream InternalSaveToBinaryStream ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.InternalSaveToStream InternalSaveToStream ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryFile LoadFromBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryStream LoadFromBinaryStream LoadFromDataSet LoadFromStream ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryFile SaveToBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryStream SaveToBinaryStream SaveToDataSet SaveToFile SaveToStream

LoadFromFileViaFormat

Delphi declaration

```
LoadFromFileViaFormat(FileName: ; AFormat:
TkbmCustomStreamFormat);
```

Load data from a file using a specified stream formatter instead of the DefaultFormatter.

LoadFromStream

Delphi declaration

```
LoadFromStream(Stream: TStream);
```

Populate the memory table with the data from a file formatted according to the DefaultFormatter settings.ne See InternalLoadFromStream_ ERROR: Unknown Link _0

See also:
ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.InternalLoadFromBinaryStream InternalLoadFromBinaryStream ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.InternalLoadFromStream InternalLoadFromStream ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.InternalSaveToBinaryStream InternalSaveToBinaryStream ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.InternalSaveToStream InternalSaveToStream ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryFile LoadFromBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryStream LoadFromBinaryStream LoadFromDataSet LoadFromFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryFile SaveToBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryStream SaveToBinaryStream SaveToDataSet SaveToFile SaveToStream

LoadFromStreamViaFormat

Delphi declaration

```
LoadFromStreamViaFormat(Stream: TStream; AFormat:
TkbmCustomStreamFormat);
```

Load data from a stream using a specified stream formatter instead of the DefaultFormatter.

LoadPersistent

Delphi declaration

```
LoadPersistent;
```

Internal support method which checks if a persistent file is setup and exists. If it does and the property Persistent is true, it will be loaded.

See also:
ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryFile LoadFromBinaryFile LoadFromFile Persistent PersistentFile

Locate

Delphi declaration

```
Locate(KeyFields: ; KeyValues: Variant; Options:
TLocateOptions): Boolean; ;
```

Locate a record in the table

You can specify any fields to search on in the *KeyFields* parameter. If the fields matches the current index, the index will be used, otherwise a sequential scan will be performed.

You can specify several field modifiers which control how each of the specified key fields should be handled.
KeyValues should contain the value or values to look for.
Options will set the locate options for the key as a whole.
The function will return true if a record was found which matches the search criteria, otherwise false.
If a record is found, it will be made current record.

Delphi Example
Example 1 - Locate on a single field for the value 22.
if mt.Locate('IntField1',22,[]) then ShowMessage('Found');

Example 2 - Locate on multiple fields for the string value 'sometext' and the integer value 22.
if mt.Locate('StringField1;IntField1',['sometext',22],[]) then ShowMessage('Found');

Example 3 - Specifying field modifers for locating case insensitively on one string field and case sensitively on another.
if mt.Locate('StringField1:C;StringField2',['sometext','anothertext'],[]) then ShowMessage('Found');

See also:
LocateRecord Lookup

Lock

Delphi declaration
```
Lock; ;
```

If you have a multithreaded application where each thread is accessing one memory table directly not using Attached tables, you must use the Lock statement before each operation on the table.
Beware that the VCL and data aware controls are not threadsafe. But Lock'ing the memory table will try to remedy that, so its even possible to have several threads accessing the same memory table which is connected to a DBGrid. Under normal circumstances, you would get an A/V exception because of thread incompabilities in the VCL and DBGrid, but using Lock prevents that.

Remember to Unlock the table after the operation(s) are finished.
Make the number of operations between a Lock/Unlock pair as short as possible not to block other threads for too long.

Delphi Example

```
mt.Lock;
try
    mt.Append;
    mt...
    mt.Post;
finally
    mt.Unlock;
end;
```

See also:
Unlock

Lookup

Delphi declaration

```
Lookup(KeyFields: ; KeyValues: Variant; ResultFields:
): Variant; ;
```

Lookup a field value from a record in the table using a search criteria.

You can specify any fields to search on in the *KeyFields* parameter. If the fields matches the current index, the index will be used, otherwise a sequential scan will be performed.

You can specify several field modifiers_ ERROR: Unknown Link _0 which control how each of the specified key fields should be handled.
KeyValues should contain the value or values to look for.
Options will set the locate options for the key as a whole.
ResultFields specifies which field values should be returned if a matching record is found.
The function will return varNull if no record was found.
Otherwise it will return either a variant containing the contents of a field if only one result field was specified. It more than one result field was specified, a variant array with all the field values will be returned.
The current record doent change in any case.

Delphi Example
Example 1 - Lookup a single field searching on a single field for the value 22.
v:=mt.Lookup('IntField1',22,'TextField');
if v<>varNull then ShowMessage('Found '+v);

Example 2 - Lookup a single field searching on multiple fields for the string value 'sometext' and the integer value 22.
v:=mt.Lookup('StringField1;IntField1',['sometext',22],'TextField');
if v<>varNull then ShowMessage('Found '+v);

Example 3 - Lookup several fields searching partially on a single field for the value 22.
v:=mt.Lookup('IntField1:P',22,'TextField1;TextField2');
if v<>varNull then ShowMessage('Found '+v[0]+', '+v[1]);

See also:
Locate LocateRecord

LookupByIndex

Delphi declaration
```
LookupByIndex(IndexName: ; KeyValues: Variant;
ResultFields: ; RespFilter: boolean): Variant;
```

Same as Lookup, but allow for specifying a specific index to use for the lookup.
The index doesnt have to be current, aslong its up to date.

MoveCurRecord

Delphi declaration
```
MoveCurRecord(Destination: Longint): Boolean;
```

Moves the record which is the current record to the specified Destination recordnumber which is 1..RecordCount.
Its only possible to move around with records on the row order index, since the index definitions of all other indexes determines the actual order of the records.

See also:
MoveRecord

MoveRecord

Delphi declaration
```
MoveRecord(Source, Destination: Integer): Boolean;
```

Moves the record specified with the Source record number to the place specified with the Destination record number.
Both values are in the range 1..RecordNumber.
Notice that the Source record is deleted from the row order index list before it will be inserted at the Destination record number. Thus the Destination record number should be specified one less that the record number actually seen before the operation.
Its only possible to move around with records on the row order index, since the index definitions of all other indexes determines the actual order of the records.

Delphi Example
Example 1 - Move a record from recordnumber 4 to record number 10 (as seen in f.ex. a grid)
mt.IndexFields:=''; // Switch to the row order index.
mt.MoveRecord(4,10-1); // Since it will first be deleted from the roworder index, the destination have to be one less.

See also:
MoveCurRecord

PackTable

Delphi declaration
```
PackTable;
```

Method which will remove all open spots from the physical table record list.
Usually when a record is deleted, the reference to the record is removed from all indexes, and the record buffer is deallocated, but the spot in the physical table record list is only cleared out, and a reference to the open spot is put in a list of open spots for later reuse.
This behaviour improves performance for both record spot allocation and positioning to a bookmark.
If you delete a large number of records, and you know you will not insert another bunch of records and you dont mind invalidating your bookmarks and removing all previous versions of records (a CheckPoint is automatically performed), you can call PackTable which will remove the open spots and clear the deleted records reference list.

Progress

Delphi declaration
```
Progress(Pct: integer; Code: TkbmProgressCode); ;
```

Call this function to trigger the OnProgress event with the specified progress code and a percentage.

Delphi Example
Example 1
mt.Progress(25,mtpcUpdate)

Reset

Delphi declaration
```
Reset;
```

Reset the memory table.
It will close the table if its open. And if any other tables are attached to it, they will be closed too.
All fields and field definitions will be deallocated.
All index and index definitions will be deallocated.
Master/Detail relations will be deallocated.

ResetAutoInc

Delphi declaration
```
ResetAutoInc;
```

Resets the next auto increment value to AutoIncMin.
Normally you would not need to use this method.

See also:
AutoIncMinValue AutoIncValue

Rollback

Delphi declaration
```
Rollback; ;
```

Rollback a transaction started with StartTransaction.
This way you can make several modifications to as many fields and records you want, and then undo it all in one go if you regrets the changes.
Notice that the operation requires a seqential scan through all the physical records, and thus is a relatively expensive operation.
Versioning needs to be enabled and versioning mode should normally be set to mtvmAllSinceCheckPoint for the transaction support to work and have any value.

See also:

Commit EnableVersioning StartTransaction TransactionLevel VersioningMode

SavePersistent

Delphi declaration

```
SavePersistent;
```

Internal support method which check if the memory table should be persistent, eg. that Persistent is true and a PersistentFile name is set.
If thats the case, it will make a backup of any files with the same name if needed according to the properties PersistentBackup , PersistentBackupExt .
Then the memory table will be saved using the PersistentSaveFormat and ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.PersistentSaveOptions PersistentSaveOptions .
The method will be automatically called during destruction or closure of the table.

See also:

LoadPersistent Persistent PersistentBackup PersistentBackupExt PersistentFile PersistentSaveFormat ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.PersistentSaveOptions PersistentSaveOptions ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryFile SaveToBinaryFile SaveToFile

SaveToDataSet

Delphi declaration

```
SaveToDataSet(Destination: TDataSet; CopyOptions:
TkbmMemTableCopyTableOptions=[]; Mapping: =''); ;
```

Save all the records from the memory table to the dataset specified with Destination.
Filtering, ranges, master/detail filters and indexes are honored.

See also:
ERROR: INVALID LINK:
kbmMemTable.TkbmCustomMemTable.SaveToBinaryFile
SaveToBinaryFile ERROR: INVALID LINK:
kbmMemTable.TkbmCustomMemTable.SaveToBinaryStream
SaveToBinaryStream SaveToFile SaveToStream

SaveToFile

Delphi declaration
```
SaveToFile(FileName: );
```

Save the contents of the memory table according to the settings of the DefaultFormatter to a file.
See TkbmMemTableSaveFlag_ ERROR: Unknown Link _0 for a description on the many different options for save.

Delphi Example
Example 1 - Save the contents of the memory table to a text (CSV) file.
mt.SaveToFile('myFile.dat',[mtfSaveDef,mtfSaveIndexDef,mtfSaveData,mtfSaveFiltered]);

See also:
ERROR: INVALID LINK:
kbmMemTable.TkbmCustomMemTable.InternalLoadFromBinaryStream
InternalLoadFromBinaryStream ERROR: INVALID LINK:
kbmMemTable.TkbmCustomMemTable.InternalLoadFromStream
InternalLoadFromStream ERROR: INVALID LINK:
kbmMemTable.TkbmCustomMemTable.InternalSaveToBinaryStream
InternalSaveToBinaryStream ERROR: INVALID LINK:
kbmMemTable.TkbmCustomMemTable.InternalSaveToStream
InternalSaveToStream ERROR: INVALID LINK:
kbmMemTable.TkbmCustomMemTable.LoadFromBinaryFile
LoadFromBinaryFile ERROR: INVALID LINK:
kbmMemTable.TkbmCustomMemTable.LoadFromBinaryStream
LoadFromBinaryStream LoadFromFile LoadFromStream ERROR: INVALID LINK:
kbmMemTable.TkbmCustomMemTable.SaveToBinaryFile
SaveToBinaryFile ERROR: INVALID LINK:
kbmMemTable.TkbmCustomMemTable.SaveToBinaryStream
SaveToBinaryStream SaveToDataSet SaveToStream

SaveToFileViaFormat

Delphi declaration
```
SaveToFileViaFormat(FileName: ; AFormat:
TkbmCustomStreamFormat);
```

Save the contents of the memory table according to the settings of the specified streamformatter to a file.

SaveToStream

Delphi declaration

```
SaveToStream(Stream: TStream);
```

Save the contents of the memory table according to the settings of the DefaultFormatter to a stream.ne See TkbmMemTableSaveFlag_ ERROR: Unknown Link _0 for a description on the many different options for save.

Delphi Example

Example 1 - Save the contents of the memory table to a newly created memory stream in text (CSV) format.

```
var
  AStream:TMemoryStream;
begin
  AStream:=TMemoryStream.Create;
  try

mt.SaveToStream(AStream,[mtfSaveDef,mtfSaveIndexDef,mtfSaveData,mtfSaveFiltered]);
    AStream.Position:=0; // If you want to do something with the stream after the save.
    ..                   // The position would otherwise point at the end of the stream.
  finally
    AStream.Free;
  end;
end;
```

See also:

ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryFile LoadFromBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryStream LoadFromBinaryStream LoadFromDataSet LoadFromFile LoadFromStream ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryFile SaveToBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryStream SaveToBinaryStream SaveToDataSet SaveToFile

SaveToStreamViaFormat

Delphi declaration

```
SaveToStreamViaFormat(Stream: TStream; AFormat:
TkbmCustomStreamFormat);
```

Save the contents of the memory table according to the settings of the specified streamformatter to a stream.

SetFieldData

Delphi declaration

```
SetFieldData(Field: TField; Buffer:
PSetFieldDataBuffer); ;
```

Internal TDataset_ ERROR: Unknown Link _0 support method. The purpose of the method is to set the raw data of a specific field in the current record.
Buffer should point to the data that the field should contain.
Generally do not use this function directly. Instead use the normal TDataset.As..... functions.

See also:
GetFieldData ERROR: INVALID LINK: TDataset.SetFieldData
TDataset.SetFieldData

SetFieldListOptions

Delphi declaration

```
SetFieldListOptions(AList: TkbmFieldList; AOptions:
TkbmifoOption; AFieldNames: );
```

Internal support method which associates field modifiers to a list of fields.
AList contain the list of fields
AOptions contain the options to be set.
AFieldNames contain the names of the fields in the list that the options should be set. Eg. 'Field1;Field2'
Generally dont use this method directly. Instead specify the modifers where appropriate together with the fieldname.

SetKey

Delphi declaration

```
SetKey;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.SetKey,1) TTable.SetKey

See also:
EditKey FindKey GotoKey

SetRange

Delphi declaration

```
SetRange(AFields: ; StartValues, EndValues: variant); ;
SetRange(StartValues, EndValues: ); ;

SetRange(AFields: ; StartValues, EndValues: variant); ;
SetRange(StartValues, EndValues: ); ;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.SetRange,1) TTable.SetRange .

See also:
ApplyRange EditRangeEnd EditRangeStart SetRangeEnd SetRangeStart

SetRangeEnd

Delphi declaration

```
SetRangeEnd;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.SetRangeEnd,1) TTable.SetRangeEnd .

See also:
ApplyRange EditRangeEnd EditRangeStart SetRange SetRangeStart

SetRangeStart

Delphi declaration

```
SetRangeStart;
```

Please refer to ERROR: INVALID LINK: !Alink(TTable.SetRangeStart,1) TTable.SetRangeStart .

See also:
ApplyRange EditRangeEnd EditRangeStart SetRange SetRangeEnd

SetVersionFieldData

Delphi declaration

```
SetVersionFieldData(Field: TField; AVersion: integer;
AValue: variant): variant;
```

Method that allows for a forcing a specific value into a field in a specified version of the record.
Newest version is 0, oldest is 999999.

SetVersionStatus

Delphi declaration

```
SetVersionStatus(AVersion: integer; AUpdateStatus:
TUpdateStatus): TUpdateStatus;
```

Set the UpdateStatus of a specific version of a record.
Newest version is 0, oldest is 999999.

Sort

Delphi declaration

```
Sort(Options: TkbmMemTableCompareOptions);
```

Sort the table on the fields set by SortFields using the sort options given in the parameter *Options*.
See SortOn .

See also:
LanguageID LocaleID SortDefault SortID SortOn SubLanguageID

SortDefault

Delphi declaration

```
SortDefault;
```

Sort the table on the fields set by SortFields using the sort options set by the property SortOptions .
See SortOn_ ERROR: Unknown Link _0 .

See also:
LanguageID LocaleID SortID SubLanguageID

SortOn

Delphi declaration
```
SortOn(FieldNames: ; Options:
TkbmMemTableCompareOptions);
```

Sort the table on the fields given by the parameter *FieldNames* using the sort options given by the parameter *Options*.

In fact the records are really not reordered, but another temporary sort index is created and automatically switched to.
At all times you can switch instantly to any other of your indexes again if you want.
The given fieldnames can include field modifiers .
Notice that the sort is done according to the current LocaleID .

See also:
LanguageID LocaleID Sort SortDefault SortFields SortID SortOptions SubLanguageID

StartTransaction

Delphi declaration
```
StartTransaction; ;
```

Start a transaction.
ne Transactions are used for atomic updates of multiple seperate record operations.
F.ex. you can start a transaction modify a few records, delete some others and perhaps insert some too, then you figure out that something was not quite right, and you would like to undo all the modifications.
This is done using Rollback . If instead you decided to keep all the changes made since the start of the transaction call Commit to make the updates/changes stick.

For transactions to work, EnableVersioning need to be true and VersioningMode should be set to mtvmAllSinceCheckPoint.
You can nest several transactions if you so wish, by doing another StartTransaction before committing or rolling any other transaction back. Remember to have as many rollback/commits as you have StartTransaction. They must match one to one.
At any time you can check what is the current level of transaction using the property TransactionLevel_ ERROR: Unknown Link _0 .

SwitchToIndex

Delphi declaration
```
SwitchToIndex(: TkbmIndex);
```

Internal method that switch the current index to another already existing index.

TestFilter

Delphi declaration

```
TestFilter(AFilter: ; AFilterOptions: TFilterOptions):
boolean;
```

Tests a specific filter on the current record, and returns false if the record would be filtered out.

Undo

Delphi declaration

```
Undo;
```

Undo the latest change on the current record.
It requires that versioning is enabled.
For multiple levels undo, VersioningMode must be all since checkpoint.

Unlock

Delphi declaration

```
Unlock; ;
```

Unlock access to the table. Must be used after using Lock.

See also:
Lock

UpdateFieldVariables

Delphi declaration

```
UpdateFieldVariables;
```

Scans the form that the memtable is placed on, and removes all old field variables for persistantly defined fields, and creates new.

UpdateIndexes

Delphi declaration

```
UpdateIndexes;
```

For performance reasons, it is better to disable all or certain indexes while a large batch insert/modification/deletion operation is going on,

and then after the batch has run, reenable the index/indexes and call UpdateIndexes to get the indexes up to date.

Delphi Example

Example 1 - Disable indexes, perform insertion, reenable indexes and update them. 7-8 times faster than updating the indexes for every insertion which is the default.

```
mt.EnableIndexes:=false;
try
 for i:=1 to 10000 do
     ... // Insert 10000 records.
finally
 mt.EnableIndexes:=true;
 mt.UpdateIndexes;
end;
```

UpdateStatus

Delphi declaration

```
UpdateStatus: TUpdateStatus; ;
```

Return the update status flag from the current record.

UpdateToDataSet

Delphi declaration

```
UpdateToDataSet(Destination: TDataSet; KeyFields: ;
Mapping: =''); ; ;
UpdateToDataSet(Destination: TDataSet; KeyFields: ;
Flags: TkbmMemTableUpdateFlags; Mapping: =''); ; ;

UpdateToDataSet(Destination: TDataSet; KeyFields: ;
Mapping: =''); ; ;
UpdateToDataSet(Destination: TDataSet; KeyFields: ;
Flags: TkbmMemTableUpdateFlags; Mapping: =''); ; ;
```

Updates the Destination dataset with the records in the memory table using the fields specified in KeyFields as a key.

See UpdateRecords .

Protected Methods/Functions

_InternalBeforeInsert

Delphi declaration

```
_InternalBeforeInsert(DataSet: TDataSet);
```

An internal event handler for the standard BeforeInsert TDataSet event. It records the record number of the current record which is needed to correctly update indexes with the newly inserted record later on.

_InternalFirst

Delphi declaration

```
_InternalFirst; ;
```

_InternalLast

Delphi declaration

```
_InternalLast; ;
```

_InternalNext

Delphi declaration

```
_InternalNext(ForceUseFilter: boolean): boolean; ;
```

_InternalPrior

Delphi declaration

```
_InternalPrior(ForceUseFilter: boolean): boolean; ;
```

AllocRecBuf

Delphi declaration

```
AllocRecBuf: TRecBuf; ;
```

AllocRecordBuffer

Delphi declaration

```
AllocRecordBuffer: PkbmAllocRecordBuffer; ;
```

Its an internal routine used by many methods when space for a new internal record structure is needed.
Usually you will not need to use it directly.

See also:
FreeRecordBuffer

BuildFilter

Delphi declaration
```
BuildFilter(AFilterParser: TkbmExprParser; AFilter: ;
AFilterOptions: TFilterOptions);
```

Compiles a filter expression given as a string.

See also:
TkbmCustomMemTable.FreeFilter

CheckActive

Delphi declaration
```
CheckActive; ;
```

Check if the table is active.
If its closed, an error will be raised.
Primarely for internal use.

CheckAutoInc

Delphi declaration
```
CheckAutoInc: boolean;
```

Checks if any auto increment field (ftAutoInc) is available in the current table definition.
Return true if there is, false if no field found.
Primarely for internal use.

CheckInActive

Delphi declaration
```
CheckInactive; ;
```

Check if the table is inactive.
If its open, an error will be raised.
Primarely for internal use.

ClearCalcFields

Delphi declaration
```
ClearCalcFields(Buffer: PkbmClearCalcFields); ;
```

CloseBlob

Delphi declaration

```
CloseBlob(Field: TField); ;
```

Internal blob support routine for compatibility with standard ERROR: INVALID LINK: !Alink(TDataset,1) TDataset operations.

CreateDetailIndex

Delphi declaration

```
CreateDetailIndex;
```

CreateFieldDefs

Delphi declaration

```
CreateFieldDefs;
```

Creates a new set of fielddefs from the current set of fields within the memtable.

CreateGroupIndex

Delphi declaration

```
CreateGroupIndex(AGroupFields: );
```

CreateRangeIndex

Delphi declaration

```
CreateRangeIndex(AFieldNames: ='');
```

DataConvert

Delphi declaration

```
DataConvert(Field: TField; Source, Dest: Pointer;
ToNative: Boolean); ;
```

A method inherited from TDataSet that provides the option to convert data for a field to a generalized value useful for storing in the record buffer.
In kbmMemTable its used to handle WideString data types.

DataEvent

Delphi declaration

```
DataEvent(Event: TDataEvent; Info: TkbmDataEventInfo);
;
```

Please refer to ERROR: INVALID LINK: !Alink(TDataset.DataEvent,1) TDataset.DataEvent

DefineProperties

Delphi declaration

```
DefineProperties(Filer: TFiler); ;
```

Used internally for defining storage of memory table data in the DFM (Delphi Form Module) at designtime controlled by StoreDataOnForm

DoAfterDelete

Delphi declaration

```
DoAfterDelete; ;
```

DoAfterOpen

Delphi declaration

```
DoAfterOpen; ;
```

DoAfterPost

Delphi declaration

```
DoAfterPost; ;
```

DoBeforeClose

Delphi declaration

```
DoBeforeClose; ;
```

DoBeforeOpen

Delphi declaration

```
DoBeforeOpen; ;
```

DoBeforePost

Delphi declaration

```
DoBeforePost; ;
```

DoCheckInActive

Delphi declaration

```
DoCheckInActive; ;
```

Internal support method which will be called to make sure all attached tables are closed. If any are open, they will be closed.

Delphi Example

An internal method which will close any open tables attached to the same base table.

DoOnFilterRecord

Delphi declaration

```
DoOnFilterRecord(ADataset: TDataset; AFiltered:
boolean); ;
```

DoOnNewRecord

Delphi declaration

```
DoOnNewRecord; ;
```

DrawAutoInc

Delphi declaration

```
DrawAutoInc(ARecord: PkbmRecord=);
```

FilterExpression

Delphi declaration

```
FilterExpression(ARecord: PkbmRecord; AFilterParser:
TkbmExprParser): boolean;
```

Internal function which returns true if the record given in ARecord is acceptable according to the filter expression tree given in AFilterParser, else false.

FilterMasterDetail

Delphi declaration

```
FilterMasterDetail(ARecord: PkbmRecord): boolean;
```

Internal function which returns true if the record given with ARecord is a detail record of the master/detail relation, else false.

FilterRange

Delphi declaration

```
FilterRange(ARecord: PkbmRecord): Boolean;
```

Internal function which returns true if the record given by ARecord is within the currently set range.

See also:
ApplyRange EditRangeEnd EditRangeStart SetRange SetRangeEnd SetRangeStart

FindRecord

Delphi declaration

```
FindRecord(Restart, GoForward: Boolean): Boolean; ;
```

Internal TDataset support function which is used to reposition the cursor in a filtered table.
Return true if the reposition was successfull, else false.

FreeDetailIndex

Delphi declaration

```
FreeDetailIndex;
```

FreeFilter

Delphi declaration

```
FreeFilter(AFilterParser: TkbmExprParser);
```

Free's a previously compiled filter expression.

See also:
TkbmCustomMemTable.BuildFilter

FreeGroupIndex

Delphi declaration
```
FreeGroupIndex;
```

FreeRangeIndex

Delphi declaration
```
FreeRangeIndex;
```

FreeRecBuf

Delphi declaration
```
FreeRecBuf(Buffer: TRecBuf); ;
```

FreeRecordBuffer

Delphi declaration
```
FreeRecordBuffer(Buffer: PkbmFreeRecordBuffer); ;
```

Internal TDataset support routine which deallocates all ressources for a physical record. The free deals with the actual record buffer, incl. blobs and indirect fields like long strings etc. but not with the indexes referencing the record.

See also:
AllocRecordBuffer

GetActiveRecord

Delphi declaration
```
GetActiveRecord: PkbmRecord;
```

Returns the record buffer for the currently active record.

GetAllData

Delphi declaration
```
GetAllData: variant;
```

Getter method which returns all the memtable data (according to the AllDataFormat) as a variant.

GetAttachCount

Delphi declaration

```
GetAttachCount: integer;
```

Getter method that returns the current number of attached tables.

GetAttachMaxCount

Delphi declaration

```
GetAttachMaxCount: integer;
```

Getter method that returns the max defined number of attached tables.

GetAutoIncMin

Delphi declaration

```
GetAutoIncMin: longint;
```

Getter method that returns the defined smallest allowed autoinc value.

GetAutoIncValue

Delphi declaration

```
GetAutoIncValue: longint;
```

Getter method that returns the current max autoinc value for the table

GetBookmarkData

Delphi declaration

```
GetBookmarkData(Buffer: PkbmGetBookmarkData; Data:
Pointer); ;
```

Internal TDataset_ ERROR: Unknown Link _0 support method which returns the bookmark data associated with the specified record. Actually Buffer points to a PkbmRecord_ ERROR: Unknown Link _0 and Data to an area with minimum size of BookmarkSize bytes.
The bookmark data in reality contains a direct pointer to the physical record. Thus if a record is deleted, the bookmark will be invalid and BookmarkValid will return false.
Switching index will essentially make all bookmarks invalid, and unpredictable.

See also:
SetBookmarkData

GetBookmarkFlag

Delphi declaration
```
GetBookmarkFlag(Buffer: PkbmGetBookmarkFlag):
TBookmarkFlag; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support function which returns the bookmark flag currently associated with the specified record.
In reality the Buffer can be directly cast to point to a ERROR: INVALID LINK: kbmMemTable.PkbmRecord PkbmRecord .
The flag returned has nothing directly to do with a bookmark and the bookmark functionality despite the name of the function.
Every time an existing record is fetched from the physical table, the flag is set to **bfCurrent** which signals that the record buffer points at the current record. In praxis, more than one recordbuffer will be marked as **bfCurrent** even if only one of them actually is current. The marking happens on each and every fetch of a physical record.
A newly allocated record which has not yet been inserted into the table of records will have the flag set to **bfInserted**.
A record which is appended to the table of records will have its flag changed from **bfInserted** to **bfEOF**, indicating that the record is the last record in the current view of the table.
A record which is inserted into the table of records will have its flag changed depending on where the insert is made.
If the insert is before first record according to current index and sortorder, it will be marked with **bfBOF**, indicating that the record is the first record in the current view of the table. If the insert is made after the last record in the table, the record will be marked with **bfEOF**.

Remember that all these markings are very volatile, and will be updated immediately on record fetch (GetRecord). They hold no historical value at all but is a temporary storage allowing ERROR: INVALID LINK: !Alink(TDataset,1) TDataset more easily to synchronize its notion of current record with the actual table.

See also:
SetBookmarkFlag

GetCanModify

Delphi declaration
```
GetCanModify: Boolean; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support function which return true if the memory table is not marked as read only.
See the property ReadOnly .

GetCapacity

Delphi declaration
```
GetCapacity: longint;
```

Getter method that returns the define record list capacity.
The capacity will automatically be increased when needed. However predefining a correct starting capacity can provide a slight performance improvement in some cases.

GetCommaText

Delphi declaration
```
GetCommaText: ;
```

Getter method that returns the memtable data as a string according to the CommaTextFormat.

GetDataSource

Delphi declaration
```
GetDataSource: TDataSource; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support routine used to obtain the datasource used for master tables.
See the MasterSource property.

GetDeletedRecordsCount

Delphi declaration
```
GetDeletedRecordsCount: TkbmNativeInt;
```

Getter methods that returns the size of the internal deleted record list.

GetEnableVersioning

Delphi declaration
```
GetEnableVersioning: boolean;
```

Getter method that returns a boolean value indicating if record versioning is enabled.

GetIndexByName

Delphi declaration

```
GetIndexByName(IndexName: ): TkbmIndex;
```

Return an already defined index by searching on the index name. The search is not case sensitive.

GetIndexes

Delphi declaration

```
GetIndexes: TkbmIndexes;
```

Getter method that returns the index list structure containing all indexes.

GetIndexField

Delphi declaration

```
GetIndexField(: integer): TField;
```

Getter method that returns field n of the current index.
n must be between 0 and IndexFieldCount-1

GetIsDataModified

Delphi declaration

```
GetIsDataModified: boolean;
```

Getter method that returns a boolean value indicating if the memtable data has been changed since the flag was reset.

GetIsIndexField

Delphi declaration

```
GetIsIndexField(Field: TField): Boolean; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support function which returns true if the specified field is part of the current index.

GetIsVersioning

Delphi declaration

```
GetIsVersioning: boolean;
```

Getter method that returns true if record versioning is enabled.

GetLanguageID

Delphi declaration
```
GetLanguageID: integer;
```

Getter method that returns the currently selected language ID

See also:
TkbmCustomMemTable.GetLocaleID
TkbmCustomMemTable.GetSortID
TkbmCustomMemTable.GetSubLanguageID

GetLocaleID

Delphi declaration
```
GetLocaleID: TkbmLocaleID;
```

Getter method that returns the currently selected locale ID

See also:
TkbmCustomMemTable.GetLanguageID
TkbmCustomMemTable.GetSortID
TkbmCustomMemTable.GetSubLanguageID

GetMasterDataLinkClass

Delphi declaration
```
GetMasterDataLinkClass: TkbmMasterDataLinkClass; ;
```

GetMasterFields

Delphi declaration
```
GetMasterFields: ;
```

Getter method that returns the currently selected master fields.

GetModifiedFlags

Delphi declaration
```
GetModifiedFlags(i: integer): boolean;
```

Getter method that returns a boolean value indicating if a specific field has been changed since the current record was navigated to. The index must be between 0 and FieldCount-1.

GetPerformance

Delphi declaration

```
GetPerformance: TkbmPerformance;
```

Getter method that returns the currently selected performance mode

GetRangeActive

Delphi declaration

```
GetRangeActive: boolean;
```

GetRecNo

Delphi declaration

```
GetRecNo: integer; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support function which returns the recordnumber of the current record according to the current index and sort order.
Use the property RecNo instead to get the recordnumber.
It will return -1 if there isnt no current record or the table is empty.
Else it will return a record number between 1 and ERROR: INVALID LINK: !Alink(TDataset.RecordCount,1) RecordCount .

See also:
SetRecNo ERROR: INVALID LINK: TDataset.RecNo TDataset.RecNo

GetRecord

Delphi declaration

```
GetRecord(Buffer: PkbmGetRecord; GetMode: TGetMode;
DoCheck: Boolean): TGetResult; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support function for getting next, prior or current record.
The function will take all filters, ranges, setkey's and master/detail relations into account when delivering the resulting record.
In effect the function adjusts the current record number reference to point to the desired record.
It will return:
grEOF if no more records are available and the *GetMode* is **gmNext**, or if *GetMode* is **gmCurrent**
and the current record is pointing after the last record.
grBOF If no more records are available and the *GetMode* is **gmPrior**, or if *GetMode* is **gmCurrent**
and the current record is pointing before the first record.

grOK if a record is available using any get mode.
It will sequentially scan for an available record with any filter, range, setkey's or master/detail relation is enabled.

Do not use this function directly. Instead use the Next, Prior, First and Last methods.

GetRecordCount

Delphi declaration

```
GetRecordCount: integer; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support function which will return the number of records taking the current index, filters, ranges, setkey's and master/detail relations into account.
See ERROR: INVALID LINK: !Alink(TDataset.RecordCount,1) RecordCount .

GetRecordSize

Delphi declaration

```
GetRecordSize: Word; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support function which will return the total record size which is the size that needs to be allocated for a basic record structure.
See ERROR: INVALID LINK: !Alink(TBDEDataset.RecordSize,1) TBDEDataset.RecordSize .

GetRecordTag

Delphi declaration

```
GetRecordTag: longint;
```

Getter method that returns the user defined tag value for the current record.

GetSortID

Delphi declaration

```
GetSortID: integer;
```

Getter method that returns the currently selected sort ID

See also:
TkbmCustomMemTable.GetLanguageID
TkbmCustomMemTable.GetLocaleID
TkbmCustomMemTable.GetSubLanguageID

GetStandalone

Delphi declaration
```
GetStandalone: boolean;
```

Getter method returning if the memtable currently is setup as a standalone table.

GetSubLanguageID

Delphi declaration
```
GetSubLanguageID: integer;
```

Getter method that returns the currently selected sub language ID

See also:
TkbmCustomMemTable.GetLanguageID
TkbmCustomMemTable.GetLocaleID
TkbmCustomMemTable.GetSortID

GetTransactionLevel

Delphi declaration
```
GetTransactionLevel: integer;
```

Getter method that returns the current transaction level.
The transaction level is incremented when doing a StartTransaction, and is decremented when doing either Commit or Rollback.

GetVersion

Delphi declaration
```
GetVersion: ;
```

Getter method returning the current version of the component as a string.

GetVersioningMode

Delphi declaration
```
GetVersioningMode: TkbmVersioningMode;
```

Getter method returning the currently specified versioning mode.

IndexFreed

Delphi declaration

```
IndexFreed(AIndex: TkbmIndex); ;
```

InternalAddRecord

Delphi declaration

```
InternalAddRecord(Buffer: PkbmInternalAddRecord;
Append: Boolean); ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which inserts or appends a record structure to the memory table, and makes sure to update all indexes as needed.
Do not use directly.

InternalBookmarkValid

Delphi declaration

```
InternalBookmarkValid(Bookmark:
PkbmInternalBookmarkValid): boolean;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support function which checks if a bookmark obtained by ERROR: INVALID LINK: !Alink(TDataset.GetBookmark,1) GetBookmark is a valid bookmark.
A bookmark can be invalidated by deleting the record, changing the index, emptying the table and several other reasons.
Bookmarks are relatively volatile, and should only be used as a short term reference. For longer term references, ERROR: INVALID LINK: !Alink(TBDEDataset.Locate,1) Locate on an unique value is a better choise.
If the bookmark is valid, true is returned, else false.
Use BookmarkValid instead.

InternalCancel

Delphi declaration

```
InternalCancel; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support routine which is called when the Cancel method is invoked.

InternalClose

Delphi declaration

```
InternalClose; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which is called when the table is closed.
It will be called when the Close method is invoked.
If the memory table is the base table of other attached memory tables, all attached memory tables are also closed and the physical table structures (TkbmCommon) deleted.
All indexes except the row order index is removed although the index definitions will be kept.
All fields are removed although the field definitions will be kept.
Do not call directly.

See also:
InternalOpen

InternalCreateFieldsOnOpen

Delphi declaration

```
InternalCreateFieldsOnOpen; ;
```

InternalDelete

Delphi declaration

```
InternalDelete; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which will be called when the Delete method is invoked.
It will remove the current record from all indexes (also in all attached tables if any) and mark the spot where the record was stored in the table as open/vacant.
The actual freeing of the allocated record structure will be handled by TDataset itself.

See also:
DeletedRecordsCount

InternalEdit

Delphi declaration

```
InternalEdit; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which will be called when the Edit method is invoked.

InternalEmptyTable

Delphi declaration

```
InternalEmptyTable;
```

Internal method which is called when EmptyTable is issued or the memory table is closed.
Its purpose is to clear all index references and dataaware controls references to any physical records.
InternalEmptyTable does not remove the physical records.

See also:
TkbmCommon.EmptyTables

InternalFirst

Delphi declaration

```
InternalFirst; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which is called when the method First is invoked.
It simply sets the record number to point before the first record.

InternalGotoBookmark

Delphi declaration

```
InternalGotoBookmark(Bookmark: Pointer); ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which is called when the method GotoBookmark is invoked.
Its job is to set the record number to the record pointed to by the bookmark.
Its not a trivial and simple operation. Normally it would require a sequential scan of the complete table to figure out the exact record number since some records could be filtered out etc.
To speed up the operation, it will try to do a binary search as close as possible to the record, which is pointed to by the bookmark, if the current index is ordered and up to date. Then it will do a small sequential scan to position it precisely.

InternalHandleException

Delphi declaration

```
InternalHandleException; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which determines what to do if an exception occur.

Current operation is simply to invoke the exception mechanism of the ERROR: INVALID LINK: !Alink(TApplication.HandleException,1) application .

InternalInitFieldDefs

Delphi declaration

```
InternalInitFieldDefs; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which is invoked during opening a table or in other situations where the table field definition needs to be up to date. The field definitions determines the physical layout of a record buffer.
If the memory table is attached to another table, the field definitions will be fetched from the base table
else if the memory table doesnt contain fielddefs, but instead designtime defined TFields (created in the field editor), those fields will be used as a template for the physical record layout.

InternalInitFieldDefsOnOpen

Delphi declaration

```
InternalInitFieldDefsOnOpen; ;
```

InternalInitRecord

Delphi declaration

```
InternalInitRecord(Buffer: PkbmInternalInitRecord); ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which is invoked when a record buffer needs to be cleared, but not deallocated.

InternalInsert

Delphi declaration

```
InternalInsert; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which will be called when the method Insert is invoked.

InternalLast

Delphi declaration

```
InternalLast; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which will be called when the method Last is invoked. It will simply change the current record to point after the last record in the current index.

InternalLoadFromStreamViaFormat

Delphi declaration

```
InternalLoadFromStreamViaFormat(AStream: TStream;
AFormat: TkbmCustomStreamFormat); ;
```

Loads data into the memtable according to the specified streamformatter.

InternalOpen

Delphi declaration

```
InternalOpen; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which will be called when the method Open is invoked.
If this is a base table (its not attached to any other table), the physical record table structures (TkbmCommon) are generated and opened.
Else if this is attached to some other base table, the physical record table structures of that table is used.
Then all defined indexes are generated and master/detail information etc. prepared.
Do not call directly.

See also:
InternalClose

InternalPost

Delphi declaration

```
InternalPost; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which is called when the method Post is invoked.
Its primary objective is to generate a new version of the current record (if versioning is enabled) and update all indexes in all tables with the new field values.

See also:
InternalCancel InternalEdit

InternalSaveToStreamViaFormat

Delphi declaration

```
InternalSaveToStreamViaFormat(AStream: TStream;
AFormat: TkbmCustomStreamFormat); ;
```

Save the memtable's data according to the specified streamformatter.

InternalSetToRecord

Delphi declaration

```
InternalSetToRecord(Buffer: PkbmInternalSetToRecord);
;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which will set the current record to point to the specified physical record.
Do not call directly.

IsCursorOpen

Delphi declaration

```
IsCursorOpen: Boolean; ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which returns true if the memory table is open and false if its not.

Loaded

Delphi declaration

```
Loaded; ;
```

Internal ERROR: INVALID LINK: !Alink(TComponent,1) TComponent support method.

LocateRecord

Delphi declaration

```
LocateRecord(KeyFields: ; KeyValues: Variant; Options:
TLocateOptions): integer;
```

Internal support function which is used by Locate and Lookup and in effect is doing all the work.

See also:
Locate Lookup

MasterChanged

Delphi declaration
```
MasterChanged(Sender: TObject); ;
```

Internal ERROR: INVALID LINK: !Alink(TMasterDataLink,1) TMasterDataLink support method which is called when the table is setup as a detail table in a master/detail relationship, and the current record of the master table changes.
Its purpose is to setup the correct search key on the detail table matching the defined M/D relation, and then force an update of the detail table.

See also:
DetailFields IndexFields MasterDisabled MasterFields MasterSource

MasterDisabled

Delphi declaration
```
MasterDisabled(Sender: TObject); ;
```

Internal ERROR: INVALID LINK: !Alink(TMasterDataLink,1) TMasterDataLink support method which is called when the table is setup as a detail table in a master/detail relationship, and the master detail relationship is disabled.
Its purpose is to refresh the detail table without any M/D filtering.

See also:
DetailFields IndexFields MasterChanged MasterFields MasterSource

Notification

Delphi declaration
```
Notification(AComponent: TComponent; Operation:
TOperation); ;
```

Internal ERROR: INVALID LINK: !Alink(TComponent,1) TComponent support method which is called every time another component is inserted or removed from a designtime form.
Its purpose is to clear out references to deleted components. Eg. the MasterSource property must be cleared if the actual dataset pointed to is removed.

Do not call directly.

ParseFilter

Delphi declaration

```
ParseFilter(FilterExpr: TkbmExprParser): variant;
```

Parses a compiler filter expression and returns a variant as result.

PopulateField

Delphi declaration

```
PopulateField(ARecord: PkbmRecord; Field: TField;
AValue: Variant);
```

Internal support method which inserts a value directly into a field in the specified record buffer.

PopulateRecord

Delphi declaration

```
PopulateRecord(ARecord: PkbmRecord; Fields: ; Values:
variant);
```

Internal support method which directly sets the values of the specified fields in a recordbuffer.

PopulateVarLength

Delphi declaration

```
PopulateVarLength(ARecord: PkbmRecord; Field: TField;
Buffer; Size: Integer);
```

Internal support method which directly sets the contents of a variable length field in a record buffer.

PostAutoInc

Delphi declaration

```
PostAutoInc(ARecord: PkbmRecord=);
```

Called automatically when new records are posted.

It checks for the records autoinc value. If the value is higher than the previously highest autoinc value, the recorded highest value is modified.

PrepareKeyRecord

Delphi declaration

```
PrepareKeyRecord(KeyRecordType: integer; Clear:
boolean);
```

Internal support method which is called when SetKey, EditKey, FindKey, FindNearest, GotoKey and ranges are set.
Its essentially prepareing the correct internal key record buffer for update by normal field access methods (.AsString etc).

ReadData

Delphi declaration

```
ReadData(Stream: TStream);
```

Used internally by the TCustomForm streaming mechanisms loading a table with data directly from the form.

See also:
DefineProperties StoreDataOnForm WriteData

RebuildFieldLists

Delphi declaration

```
RebuildFieldLists; ;
```

SetActive

Delphi declaration

```
SetActive(Value: boolean); ;
```

Setter method that activates (opens) or deactivates (closes) the memtable.

SetAllData

Delphi declaration

```
SetAllData(AVariant: variant);
```

Setter method which populates a memtable from a variant according to the AllDataFormat.

SetAttachedTo

Delphi declaration

```
SetAttachedTo(Value: TkbmCustomMemTable);
```

Setter method that controls which other memtable this memtable is attached to.

SetAttachMaxCount

Delphi declaration

```
SetAttachMaxCount(AValue: integer);
```

Setter method that define the max number of tables that can be attaced to this table at runtime while the table is open.

SetAutoIncMinValue

Delphi declaration

```
SetAutoIncMinValue(AValue: longint);
```

Setter method setting the lowest integer value the autoinc field can have.

SetAutoUpdateFieldVariables

Delphi declaration

```
SetAutoUpdateFieldVariables(AValue: boolean);
```

Setter method setting if TForm/TDatamodule hosted field variables (the ones automatically created when defining persistent fields at designtime) should be automatically updated when the table gets a new definition (for example by loading data with definition into it).
If the field variables are not updated when the table structure changes, it will usually result in access violations because dataaware controls often will be hooked to those fields (which now may not exist) at designtime.

SetBlockReadSize

Delphi declaration

```
SetBlockReadSize(Value: Integer); ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method (D5/BCB5 or newer only) which sets the number of records that are returned in one block.
Its mainly available for Midas compatibility.

SetBookmarkData

Delphi declaration

```
SetBookmarkData(Buffer: PkbmSetBookmarkData; Data:
Pointer); ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which associates a bookmark with the specified record.
Buffer points to a ERROR: INVALID LINK: kbmMemTable.PkbmRecord PkbmRecord and Data to a PPkbmRecord .
The bookmark data in reality contains a direct pointer to the physical record. Thus if a record is deleted, the bookmark will be invalid and BookmarkValid will return false.
Switching index will essentially make all bookmarks invalid, and unpredictable.

See also:
GetBookmarkData

SetBookmarkFlag

Delphi declaration

```
SetBookmarkFlag(Buffer: PkbmSetBookmarkFlag; Value:
TBookmarkFlag); ;
```

Internal TDataset_ ERROR: Unknown Link _0 support method which sets the bookmark flag currently associated with the specified record.
In reality the Buffer can be directly cast to point to a PkbmRecord_ ERROR: Unknown Link _0 .
The flag returned has nothing directly to do with a bookmark and the bookmark functionality despite the name of the method.

See also:
GetBookmarkFlag

SetCapacity

Delphi declaration
```
SetCapacity(AValue: longint);
```

Setter method that specifies the capacity of the internal record list.

SetCommaText

Delphi declaration
```
SetCommaText(AString: );
```

Setter method that populates the memtable from a string which is formatted according to the CommaTextFormat.

SetDataSource

Delphi declaration
```
SetDataSource(Value: TDataSource);
```

Setter method that specifies which datasource should be this tables master table.

SetDeltaHandler

Delphi declaration
```
SetDeltaHandler(AHandler: TkbmCustomDeltaHandler);
```

Setter method that specifies which delta handler should be connected to this memtable.

SetDetailFields

Delphi declaration
```
SetDetailFields(Value: );
```

Setter method which specifies which fields should be used as the index in the detail table when doing master/detail operations.
Its an alternative to the standard TDataSet IndexFieldNames, and allow you to have one index defined for sorting and others, and another setting defined for master/detail relations at the same time.

SetEnableVersioning

Delphi declaration

```
SetEnableVersioning(AValue: boolean);
```

Setter method that controls if record versioning is enabled.

SetFiltered

Delphi declaration

```
SetFiltered(Value: boolean); ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which is called everytime the *Filtered* property is changed.

SetFilterText

Delphi declaration

```
SetFilterText(Value: ); ;
```

Internal ERROR: INVALID LINK: !Alink(TDataset,1) TDataset support method which are called whenever the *Filter* property is modified.
Its purpose is to compile the filter expression into a parse tree which is what the memory table interpret during filtering.

SetIndexDefs

Delphi declaration

```
SetIndexDefs(Value: TIndexDefs);
```

Setter method setting the TIndexDefs of the memtable.

SetIndexField

Delphi declaration

```
SetIndexField(: integer; Value: TField);
```

SetIndexFieldNames

Delphi declaration

```
SetIndexFieldNames(FieldNames: );
```

SetIndexName

Delphi declaration

```
SetIndexName(IndexName: );
```

SetIsDataModified

Delphi declaration

```
SetIsDataModified(AValue: boolean);
```

SetIsFiltered

Delphi declaration

```
SetIsFiltered; ;
```

SetLanguageID

Delphi declaration

```
SetLanguageID(Value: integer);
```

SetLoadedCompletely

Delphi declaration

```
SetLoadedCompletely(Value: boolean);
```

Protected method which can be called to set the LoadedCompletely flag.
This is mostly interesting for components inheriting from kbmMemTable.

Delphi Example

mt.SetLoadedCompletely(true);

SetLocaleID

Delphi declaration

```
SetLocaleID(Value: TkbmLocaleID);
```

SetMasterFields

Delphi declaration

```
SetMasterFields(Value: );
```

SetOnFilterRecord

Delphi declaration

```
SetOnFilterRecord(Value: TFilterRecordEvent); ;
```

SetPerformance

Delphi declaration

```
SetPerformance(AValue: TkbmPerformance);
```

SetRecNo

Delphi declaration

```
SetRecNo(Value: integer); ;
```

Internal TDataset_ ERROR: Unknown Link _0 support method which repositions to the specified record number honoring the current index, sort order and filteres.
Use the property RecNo instead to set the recordnumber.
Value must be a number between 1 and RecordCount_ ERROR: Unknown Link _0 .

See also:
GetRecNo ERROR: INVALID LINK: TDataset.RecNo TDataset.RecNo

SetRecordTag

Delphi declaration

```
SetRecordTag(Value: longint);
```

Setter method setting the user defined tag value for the current record.

SetSortID

Delphi declaration

```
SetSortID(Value: integer);
```

SetStandalone

Delphi declaration

```
SetStandalone(AValue: boolean);
```

SetStatusFilter

Delphi declaration

```
SetStatusFilter(Value: TUpdateStatusSet);
```

Setter method setting the TUpdateStatus filter that should be applied to the current view.

SetSubLanguageID

Delphi declaration

```
SetSubLanguageID(Value: integer);
```

SetTableState

Delphi declaration

```
SetTableState(AValue: TkbmState);
```

Setter method setting the tables current state.

SetVersioningMode

Delphi declaration

```
SetVersioningMode(AValue: TkbmVersioningMode);
```

UpdateRecords

Delphi declaration

```
UpdateRecords(Source, Destination: TDataSet;
KeyFields: ; Count: longint; Flags:
TkbmMemTableUpdateFlags; Mapping: ): longint;
```

Internal support function which tries to update the *Destination* dataset with *Count* records from the *Source*.
Using the specified *KeyFields* field names for both the source and the destination dataset, it will insert records not existing in the destination and update those which is allready there.
No automatic deletion will happen.
A better and faster way is to use a deltahandler .

WriteData

Delphi declaration
```
WriteData(Stream: TStream);
```

Used internally by the TCustomForm streaming mechanisms saving a table directly to the form definition.

See also:
DefineProperties ReadData StoreDataOnForm

Public Events

OnCompareFields

Delphi declaration
```
OnCompareFields: TkbmOnCompareFields FOnCompareFields
FOnCompareFields;
```

Write an eventhandler if you want to make your own special comparison of two records.

See also:
CompareFields

OnCompressBlobStream

Delphi declaration
```
OnCompressBlobStream: TkbmOnCompress
FOnCompressBlobStream FOnCompressBlobStream;
```

Write an eventhandler here if you want to compress blobs in memory.

See also:
OnDecompressBlobStream

OnCompressField

Delphi declaration
```
OnCompressField: TkbmOnCompressField FOnCompressField
FOnCompressField;
```

Write an event handler here if you want to make your own field compression algorithm.

See also:
TkbmCommon.CompressFieldBuffer OnDecompressField

OnCompressSave

Delphi declaration
property OnCompressSave: TkbmOnCompress read FOnCompressSave write FOnCompressSave;

Write an event handler here if you want to compress the data before its saved to an external storage.

See also:
OnDecompressLoad ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryFile SaveToBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryStream SaveToBinaryStream

OnDecompressBlobStream

Delphi declaration
```
OnDecompressBlobStream: TkbmOnDecompress
FOnDecompressBlobStream FOnDecompressBlobStream;
```

Write an eventhandler here if you want to de compress blobs in memory. Use it in combination with OnCompressBlobStream.

See also:
OnCompressBlobStream

OnDecompressField

Delphi declaration
```
OnDecompressField: TkbmOnDecompressField
FOnDecompressField FOnDecompressField;
```

Write your own event handler here if you want to make your own decompression algorithm for fields.

See also:
TkbmCommon.DecompressFieldBuffer OnCompressField

OnDecompressLoad

Delphi declaration
property OnDecompressLoad: TkbmOnDecompress read FOnDecompressLoad write FOnDecompressLoad;

Write an event handler here if you want to de compress the data before its loaded from an external storage.
Use the event in combination with the OnCompressSave event.

See also:
OnCompressSave

OnFilterIndex

Delphi declaration
```
OnFilterIndex: TkbmOnFilterIndex FOnFilterIndex
FOnFilterIndex;
```

Write an event handler here if you want to maintain a filtered index by using code.

OnLoad

Delphi declaration
```
OnLoad: TkbmOnLoad FOnLoad FOnLoad;
```

Called before LoadFromFile, LoadFromBinaryFile, LoadFromStream, LoadFromBinaryStream and LoadFromDataset is actually starting to load anything.

See also:
ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryFile LoadFromBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryStream LoadFromBinaryStream LoadFromFile LoadFromStream

OnLoadField

Delphi declaration
```
OnLoadField: TkbmOnLoadField FOnLoadField
FOnLoadField;
```

Called right after a field has been loaded with data using either LoadFromFile, LoadFromStream, LoadFromBinaryFile, LoadFromBinaryStream, LoadFromDataset and CopyRecords.

See also:
CopyRecords ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryFile LoadFromBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryStream

LoadFromBinaryStream LoadFromDataSet LoadFromFile LoadFromStream

OnLoadRecord

Delphi declaration

```
OnLoadRecord: TkbmOnLoadRecord FOnLoadRecord FOnLoadRecord;
```

Called after all the fields has been loaded, but before the record has been posted using LoadFromFile, LoadFromBinaryFile, LoadFromStream, LoadFromBinaryStream, CopyRecords and LoadFromDataset.
If you set the Accept flag to false, the record will be skipped.

See also:
CopyRecords ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryFile LoadFromBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.LoadFromBinaryStream LoadFromBinaryStream LoadFromDataSet LoadFromFile LoadFromStream

OnProgress

Delphi declaration

```
OnProgress: TkbmOnProgress FOnProgress FOnProgress;
```

Called during lengthy operations f.ex. to notify the user about something is happening.

OnSave

Delphi declaration

```
OnSave: TkbmOnSave FOnSave FOnSave;
```

Called right before SaveToStream, SaveToFile, SaveToBinaryStream, SaveToBinaryFile, SaveToDataset and UpdateToDataset is starting to save.

See also:
ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryFile SaveToBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryStream SaveToBinaryStream SaveToDataSet SaveToFile SaveToStream UpdateToDataSet

OnSaveField

Delphi declaration
```
OnSaveField: TkbmOnSaveField FOnSaveField
FOnSaveField;
```

Called right before a field is about to be saved using either SaveToFile, SaveToStream, SaveToBinaryFile, SaveToBinaryStream, SaveToDataset and UpdateToDataset.

See also:
ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryFile SaveToBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryStream SaveToBinaryStream SaveToDataSet SaveToFile SaveToStream UpdateToDataSet

OnSaveRecord

Delphi declaration
```
OnSaveRecord: TkbmOnSaveRecord FOnSaveRecord
FOnSaveRecord;
```

Called right before a record is about to be saved using either SaveToFile, SaveToStream, SaveToBinaryFile, SaveToBinaryStream, SaveToDataset and UpdateToDataset.

See also:
ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryFile SaveToBinaryFile ERROR: INVALID LINK: kbmMemTable.TkbmCustomMemTable.SaveToBinaryStream SaveToBinaryStream SaveToDataSet SaveToFile SaveToStream UpdateToDataSet

OnSetupField

Delphi declaration

```
OnSetupField: TkbmOnSetupField FOnSetupField
FOnSetupField;
```

Called before a table is opened to allow you to change how a field should be handled and stored in memory.

Each field can have one or several flags set during time of open:
kbmffIndirect, **kbmffCompress**, **kbmffModified**

kbmffIndirect means that the field is not directly stored in the record. Instead a pointer is stored, pointing to the actual data of the field. This is usefull if the size of that specific field contents is varying allot. If f.ex. 100 records each having a string field of 200 bytes are stored in a table, and the field contents are stored directly in a record, the record would be 200+ bytes long and thus the table would take up 100x200+ or 20000+ bytes, even if the stringfield in all records is empty.
Instead it would be wise to set this field up as a kbmffIndirect field which means that only a pointer will be stored in each record in the example. The resulting memory usage for those 100 records would then be: 100x4+ or 400+ bytes.
kbmffIndirect will automatically be set for all fields which are considered *kbmVarLengthNonBlobTypes* and which *Size* > 12, unless the *Performance* property is set to *mtpfFast*.
If the field is one of *kbmBlobTypes*, then *kbmffIndirect* will be set in all cases.

kbmffCompress can be set to enable additional compression of indirect fields. This can slow down operation a little bit, but can also speed up operations if memoryspace is short. *kbmffCompress* will automatically be set on all indirect fields if *Performance* is set to *mtpfSmall*.

kbmffModified will be set on a field if its modified. It will be a temporary setting which has no meaning during table open. Dont use directly. Instead use *IsFieldModified*. The flag will be cleared during a *Cancel* or *Post*.

See also:
kbmBinaryTypes kbmBlobTypes kbmNonBlobTypes kbmStringTypes kbmSupportedFieldTypes kbmVarLengthNonBlobTypes OnCompressField OnDecompressField OnSetupFieldProperties

OnSetupFieldProperties

Delphi declaration

```
OnSetupFieldProperties: TkbmOnSetupFieldProperties
FOnSetupFieldProperties FOnSetupFieldProperties;
```

Called during copying of table structure from another source using CreateTableAs or LoadFromDataset.

See also:
CreateTableAs LoadFromDataSet

OnUserAggregate

Delphi declaration

```
OnUserAggregate: TkbmOnUserAggregate FOnUserAggregate
FOnUserAggregate;
```

TkbmCustomStreamFormat File: kbmMemTable

This class is used as base for all streaming of kbmMemTable contents to external medias. Base on this, kbmMemTable bundles CSV and binary streaming classes.

Delphi declaration
```
TkbmCustomStreamFormat=Class(TComponent);
```

Public Properties/Variables

FormatSettings

Delphi declaration
```
FormatSettings: TFormatSettings FFormatSettings
FFormatSettings;
```

Protected Properties/Variables

OrigStream

Delphi declaration
```
OrigStream: TStream FOrigStream FOrigStream;
```

This is the original stream provided by the user for which the save/load operation is planned to happen.
It differs from the WorkStream when compression/decompression is in operation.

See also:
TkbmCustomStreamFormat.WorkStream

sfAppend

Delphi declaration
```
sfAppend: TkbmStreamFlagsAppend FsfAppend FsfAppend;
```

Controls if data are appended to the storage, or if the storage is overwritten.

sfBlobs

Delphi declaration
```
sfBlobs: TkbmStreamFlagsBlobs FsfBlobs FsfBlobs;
```

Controls if blobs are saved to/loaded from the storage.

sfCalculated

Delphi declaration

```
sfCalculated: TkbmStreamFlagsCalculated FsfCalculated
FsfCalculated;
```

Controls if contents of calculated fields are saved or loaded.

sfData

Delphi declaration

```
sfData: TkbmStreamFlagsData FsfData FsfData;
```

Controls if data (i.e. actual field values) is saved or loaded

sfDef

Delphi declaration

```
sfDef: TkbmStreamFlagsDef FsfDef FsfDef;
```

Controls if definitions (field definitions etc.) are saved/loaded.

sfDeltas

Delphi declaration

```
sfDeltas: TkbmStreamFlagsDeltas FsfDeltas FsfDeltas;
```

Controls if all changes for a record is saved/loaded. If not, only newest record is stored/read.
For this flag to be operational, sfData must be set appropriately.

See also:
TkbmCustomStreamFormat.sfData

sfDisplayWidth

Delphi declaration

```
sfDisplayWidth: TkbmStreamFlagsDisplayWidth
FsfDisplayWidth FsfDisplayWidth;
```

sfDontFilterDeltas

Delphi declaration

```
sfDontFilterDeltas: TkbmStreamFlagsDontFilterDeltas
FsfDontFilterDeltas FsfDontFilterDeltas;
```

Controls if deltas will be filtered when saved.

Filtering ensures that records that are first inserted and later deleted wont be saved/loaded.

sfFieldKind

Delphi declaration

```
sfFieldKind: TkbmStreamFlagsFieldKind FsfFieldKind
FsfFieldKind;
```

Controls if the fieldkind (fkData, fkLookup etc.) is saved/loaded.
If not, the streamed understanding of the field kind will automatically be fkData.

sfFiltered

Delphi declaration

```
sfFiltered: TkbmStreamFlagsFiltered FsfFiltered
FsfFiltered;
```

Controls if any active filters will affect which records that are being saved.
If specified, then only records that are accepted by the current filter settings will be saved.

sfFromStart

Delphi declaration

```
sfFromStart: TkbmStreamFlagsFromStart FsfFromStart
FsfFromStart;
```

Controls if loading will load from start of the storage/stream or from the current position.

sfIgnoreMasterDetail

Delphi declaration

```
sfIgnoreMasterDetail:
TkbmStreamFlagsIgnoreMasterDetail
FsfIgnoreMasterDetail FsfIgnoreMasterDetail;
```

Controls if data saved obeys master/detail settings. If yes and the save is based on detail data, then only the actual detail data will be saved, not the complete contents of the detail table.

sfIgnoreRange

Delphi declaration

```
sfIgnoreRange: TkbmStreamFlagsIgnoreRange
FsfIgnoreRange FsfIgnoreRange;
```

Controls if defined ranges should be obeyed when saving data.

sfIndexDef

Delphi declaration

```
sfIndexDef: TkbmStreamFlagsIndexDef FsfIndexDef
FsfIndexDef;
```

Controls if index definitions must be saved or loaded.
For the flag to have effect, sfDef must be set appropriately.

sfLookup

Delphi declaration

```
sfLookup: TkbmStreamFlagsLookup FsfLookup FsfLookup;
```

Controls if contents of lookup fields are saved/loaded.

sfNonVisible

Delphi declaration

```
sfNonVisible: TkbmStreamFlagsNonVisible FsfNonVisible
FsfNonVisible;
```

Controls if contents for non visible fields are saved/loaded.

Version

Delphi declaration

```
Version: GetVersion SetVersion;
```

Provides access to the version of the stream.

WorkStream

Delphi declaration

```
WorkStream: TStream FWorkStream FWorkStream;
```

This is the actual stream that is being read from or written to.
It differs from the OrigiStream when compression/decompression is in operation.

See also:
TkbmCustomStreamFormat.OrigStream

Public Methods/Functions

Assign

Delphi declaration
```
Assign(Source: TPersistent); ;
```

Create

Delphi declaration
```
Create(AOwner: TComponent); ;
```

Protected Methods/Functions

AfterLoad

Delphi declaration
```
AfterLoad(ADataset: TkbmCustomMemTable); ;
```

A method which is called after a load operation has finished.
It can be used to clean up settings made in the BeforeLoad method.
Remember to call inherited AfterLoad(ADataSet).

See also:
TkbmCustomStreamFormat.AfterSave
TkbmCustomStreamFormat.BeforeLoad
TkbmCustomStreamFormat.BeforeSave

AfterSave

Delphi declaration
```
AfterSave(ADataset: TkbmCustomMemTable); ;
```

A method which is called after a save operation has finished.
It can be used to clean up settings made in the BeforeSave method.
Remember to call inherited AfterSave(ADataSet).

See also:
TkbmCustomStreamFormat.AfterLoad
TkbmCustomStreamFormat.BeforeLoad
TkbmCustomStreamFormat.BeforeSave

BeforeLoad

Delphi declaration
```
BeforeLoad(ADataset: TkbmCustomMemTable); ;
```

A method which is called before a load operation is starting.
It can be used to initialize things needed for the load operation.
The AfterLoad method can be used to clean it up afterwards.
Remember to call inherited BeforeLoad(ADataSet).

See also:
TkbmCustomStreamFormat.AfterLoad
TkbmCustomStreamFormat.AfterSave
TkbmCustomStreamFormat.BeforeSave

BeforeSave

Delphi declaration
```
BeforeSave(ADataset: TkbmCustomMemTable); ;
```

A method which is called before a save operation is starting.
It can be used to initialize things needed for the save operation.
The AfterSave method can be used to clean it up afterwards.
Remember to call inherited BeforeSave(ADataSet).

See also:
TkbmCustomStreamFormat.AfterLoad
TkbmCustomStreamFormat.AfterSave
TkbmCustomStreamFormat.BeforeLoad

DetermineLoadFieldIDs

Delphi declaration
```
DetermineLoadFieldIDs(ADataset: TkbmCustomMemTable;
AList: TStringList; Situation:
TkbmDetermineLoadFieldsSituation); ;
```

Method which must return a list of field identifiers (typically the field names or display names) that are to be loaded.

AList must contain the list of names.
Situation provides information about in what situation this method is called.
It can be dlfBeforeLoad or dlfAfterLoadDef which respectively indicates if its before the load operation really starts or if its right after the table definition has been loaded.
This will often be overridden by custom streamformatters.

DetermineLoadFieldIndex

Delphi declaration

```
DetermineLoadFieldIndex(ADataset: TkbmCustomMemTable;
ID: ; FieldCount: integer; OrigIndex: integer; NewIndex:
integer; Situation: TkbmDetermineLoadFieldsSituation);
;
```

A method that maps between the relative index of the field to be loaded in the datastream and the current field order in the memtable.
Given the field identifier string (ID) which is produced in DetermineLoadFieldIDs this method should set the NewIndex to an index, from 0 to FieldCount-1, which is the index of a currently existing field in the memtable in which the data from this streamed field should be placed.
This will often be overridden by custom streamformatters.

DetermineLoadFields

Delphi declaration

```
DetermineLoadFields(ADataset: TkbmCustomMemTable;
Situation: TkbmDetermineLoadFieldsSituation); ;
```

Method that construct the complete mapping between streamed fields and actual fields in the memtable.
It internally use DetermineLoadFieldIDs and DetermineLoadFieldIndex which should be provided by custom streamformatters.
The generated mapping is stored in a protected integer array named LoadFields.
If an entry in LoadFields has the value of -1, that particular field wont be loaded.
This method will usually not be overridden by custom streamformatters.

DetermineSaveFields

Delphi declaration

```
DetermineSaveFields(ADataset: TkbmCustomMemTable); ;
```

Method that construct the complete mapping between actual fields in the memtable and their order in the stream.

The generated mapping is stored in a protected integer array named SaveFields.
If an entry in SaveFields has the value of -1, that particular field wont be saved.
This method will usually not be overridden by custom streamformatters.

GetVersion

Delphi declaration
```
GetVersion: ; ;
```

Return the version of the streamformatter.
It should normally be overridden by custom streamformatters.

Load

Delphi declaration
```
Load(ADataset: TkbmCustomMemTable); ;
```

Method responsible for loading data into the memtable.
It internaly calls several other methods which should be overrridden, like LoadData, LoadDef, DetermineLoadFieldIDs and DetermineLoadFieldIndex.
This method is usually not overridden by custom streamformatters.

See also:
TkbmCustomStreamFormat.DetermineLoadFieldIDs
TkbmCustomStreamFormat.DetermineLoadFieldIndex
TkbmCustomStreamFormat.DetermineLoadFields
TkbmCustomStreamFormat.LoadData
TkbmCustomStreamFormat.LoadDef TkbmCustomStreamFormat.Save

LoadData

Delphi declaration
```
LoadData(ADataset: TkbmCustomMemTable); ;
```

This method is responsible of loading the record data from the stream.
It must be overridden by custom streamformatters.
For an example of how to use, please see the kbmMemBinaryStreamFormat.pas or kbmMemCSVStreamFormat source files.

LoadDef

Delphi declaration
```
LoadDef(ADataset: TkbmCustomMemTable); ;
```

This method is responsible of loading the table definition from the stream.
It must be overridden by custom streamformatters.
For an example of how to use, please see the kbmMemBinaryStreamFormat.pas or kbmMemCSVStreamFormat source files.

Refresh

Delphi declaration

```
Refresh(ADataset: TkbmCustomMemTable); ;
```

Reposition

Delphi declaration

```
Reposition(ADataset: TkbmCustomMemTable); ;
```

Save

Delphi declaration

```
Save(ADataset: TkbmCustomMemTable); ;
```

Method responsible for saving the memtable data to a stream.
It internaly calls several other methods which should be overrridden, like SaveData, SaveDef.
This method is usually not overridden by custom streamformatters.

See also:
TkbmCustomStreamFormat.Load
TkbmCustomStreamFormat.SaveData
TkbmCustomStreamFormat.SaveDef

SaveData

Delphi declaration

```
SaveData(ADataset: TkbmCustomMemTable); ;
```

This method is responsible of saving the record data to a stream.
It must be overridden by custom streamformatters.
For an example of how to use, please see the kbmMemBinaryStreamFormat.pas or kbmMemCSVStreamFormat source files.

SaveDef

Delphi declaration

```
SaveDef(ADataset: TkbmCustomMemTable); ;
```

This method is responsible of saving the table definition to a stream.
It must be overridden by custom streamformatters.
For an example of how to use, please see the kbmMemBinaryStreamFormat.pas or kbmMemCSVStreamFormat source files.

SetIgnoreAutoIncPopulation

Delphi declaration

```
SetIgnoreAutoIncPopulation(ADataset:
TkbmCustomMemTable; Value: boolean);
```

Public Events

OnAfterLoad

Delphi declaration

```
OnAfterLoad: TNotifyEvent FOnAfterLoad FOnAfterLoad;
```

This event is triggered when the data has been loaded

OnAfterSave

Delphi declaration

```
OnAfterSave: TNotifyEvent FOnAfterSave FOnAfterSave;
```

This event is triggered when the data has been saved/stored.

OnBeforeLoad

Delphi declaration

```
OnBeforeLoad: TNotifyEvent FOnBeforeLoad
FOnBeforeLoad;
```

This event is triggered right before the data will be loaded.

OnBeforeSave

Delphi declaration

```
OnBeforeSave: TNotifyEvent FOnBeforeSave
FOnBeforeSave;
```

This event is triggered right before the data is being saved/stored.

OnCompress

Delphi declaration

```
OnCompress: TkbmOnCompress FOnCompress FOnCompress;
```

This event is triggered when the stored data is to be compressed.

OnDeCompress

Delphi declaration

```
OnDeCompress: TkbmOnDecompress FOnDecompress
FOnDecompress;
```

This event is triggered when the saved/stored data is to be decompressed for loading.

TkbmDataEventInfo File: kbmMemTypes

Delphi declaration

```
TkbmDataEventInfo=Longint;
```

TkbmDeltaHandlerGetValue File: kbmMemTable

An event handler method that allows for changing the value that will be resolved for a specific field.

Delphi declaration

```
TkbmDeltaHandlerGetValue=(ADeltaHandler: TkbmCustomDeltaHandler;
AField: TField; AValue: variant) ;
```

TkbmDetermineLoadFieldsSituation File: kbmMemTable

Value	Meaning
dlfBeforeLoad:	
dlfAfterLoadDef:	

Delphi declaration

```
TkbmDetermineLoadFieldsSituation =(dlfBeforeLoad, dlfAfterLoadDef);
```

TkbmExprParser File: kbmMemTable

Delphi declaration

```
TkbmExprParser=Class(TExprParser);;
```

TkbmFieldList File: kbmMemTable

A class use for maintaining a list of fields and the field mofiers .

Delphi declaration
```
TkbmFieldList=Class
```

Public Properties/Variables

Count

Delphi declaration
```
Count: Integer FCount;
```

Return the number of fields in the fieldlist.

FieldCount

Delphi declaration
```
FieldCount: integer;
```

FieldNo

Delphi declaration
```
FieldNo: integer;
```

Direct access into the fieldlist to access the fieldno of a specific field in the list.

FieldOfs

Delphi declaration
```
FieldOfs: integer;
```

Direct access into the fieldlist to access the fieldofs of a specific field in the list.

Fields

Delphi declaration
```
Fields: TField;
```

Direct access into the fieldlist to access the TField of a specific field in the list.

Items

Delphi declaration
property Items[Index: Integer]: Pointer read Get write Put;

Return a field reference in the field list using an index into the list.
The returned pointer is actually a TField reference.

LocateOptions

Delphi declaration
```
LocateOptions: TLocateOptions;
```

Options

Delphi declaration
```
Options: TkbmifoOptions;
```

Direct access into the fieldlist to access the options for a specific field in the list.

PrimaryCount

Delphi declaration
```
PrimaryCount: Integer FPrimaryCount;
```

Values

Delphi declaration
property Values[Index: Integer]: integer read GetVal write PutVal;

Return the field modifier bitmask for a specific field index in the field list.

See also:
Add

Public Methods/Functions

Add

Delphi declaration
```
Add(ADataSet: TkbmCustomMemTable; AField: TField;
AValue: TkbmifoOptions): Integer;
```

Add a field and a field modifier flag to a field list.
AField should actually be a *TField* reference.
AValue should be a bitmask of modifier flags:

mtifoDescending=1
mtifoCaseInsensitive=2
mtifoPartial=4
mtifoIgnoreNull=8
mtifoIgnoreLocale=16

The return value will be the index of the field in the field list.

AssignTo

Delphi declaration
```
AssignTo(AFieldList: TkbmFieldList);
```

Assign the contents of this field list to another field list, overwriting all contents of the other list first.

Build

Delphi declaration
```
Build(ADataset: TkbmCustomMemTable; AFieldNames: ;
AAggregateFieldNaming: boolean=false);
```

Clear

Delphi declaration
```
Clear; ;
```

Clear a fieldlist.
The fields referenced are not freed. That is someone else job.

ClearOptions

Delphi declaration
```
ClearOptions;
```

Clear all field options from all fields without clearing the fields themself.

Create

Delphi declaration
```
Create; ;
```

Constructor for a field list.

DefineAdditionalOrderFields

Delphi declaration

```
DefineAdditionalOrderFields(ADataSet:
TkbmCustomMemTable; AFieldList: TkbmFieldList);
```

Destroy

Delphi declaration

```
Destroy; ;
```

Destructor of a fieldlist.

FindField

Delphi declaration

```
FindField(AFieldName: ): TField;
```

GetAsString

Delphi declaration

```
GetAsString: ;
```

IndexOf

Delphi declaration

```
IndexOf(Item: TField): Integer;
```

Find index of a field in the fieldlist.
Item should actually be a TField reference.

IsEqualTo

Delphi declaration

```
IsEqualTo(AList: TkbmFieldList; ASameCase: boolean):
boolean;
```

MergeOptionsTo

Delphi declaration

```
MergeOptionsTo(AFieldList: TkbmFieldList);
```

Merge field options of this field list into another field list that, appart from the options, must be identical.

SetOptions

Delphi declaration

```
SetOptions(ADataSet: TkbmCustomMemTable; AOptions:
TkbmifoOption; AFieldNames: );
```

StartsWith

Delphi declaration

```
StartsWith(AList: TkbmFieldList; ASameCase: boolean):
boolean;
```

TkbmFieldTypes File: kbmMemTable

A type which can hold a set of field types.

Delphi declaration

```
TkbmFieldTypes=DB.TFieldType;
```

TkbmifoOption File: kbmMemTable

Value	Meaning
mtifoDescending:	
mtifoCaseInsensitive:	
mtifoPartial:	
mtifoIgnoreNull:	
mtifoIgnoreLocale:	
mtifoAggregate:	
mtifoAggSum:	
mtifoAggMin:	
mtifoAggMax:	
mtifoAggCount:	
mtifoAggAvg:	
mtifoAggUsr1:	
mtifoAggUsr2:	
mtifoAggUsr3:	
mtifoIgnoreNonSpace:	
mtifoIgnoreKanatype:	
mtifoIgnoreSymbols:	
mtifoIgnoreWidth:	

Delphi declaration

```
TkbmifoOption =(mtifoDescending, mtifoCaseInsensitive, mtifoPartial, mtifoIgnoreNull,
             mtifoIgnoreLocale, mtifoAggregate, mtifoAggSum, mtifoAggMin, mtifoAggMax,
             mtifoAggCount, mtifoAggAvg, mtifoAggUsr1, mtifoAggUsr2, mtifoAggUsr3,
             mtifoIgnoreNonSpace, mtifoIgnoreKanatype, mtifoIgnoreSymbols,
             mtifoIgnoreWidth);
```

TkbmifoOptions File: kbmMemTable

A set of field index options.

Delphi declaration

```
TkbmifoOptions=TkbmifoOption;
```

TkbmIndex File: kbmMemTable

This class contain all the logic needed to maintain one single index.
You will not need to create the TkbmIndex class yourself directly. Instead use the AddIndex or AddFilteredIndex method of TkbmCustomMemTable .

Delphi declaration
```
TkbmIndex=Class
```

Public Properties/Variables

Base

Delphi declaration
```
Base: TkbmIndex FBaseIndex;
```

Dataset

Delphi declaration
```
Dataset: TkbmCustomMemTable FDataSet FDataSet;
```

Specifies the memory table the index is created on.
Generally dont change this directly. Instead recreate the index on another memory table using AddIndex or AddFilteredIndex.

DependantIndexes

Delphi declaration
```
DependantIndexes: TkbmIndexes FDependantIndexes;
```

Enabled

Delphi declaration
```
Enabled: boolean FEnabled SetEnabled;
```

If set to true the index will be updated on every update occation.
If set to false, all updates of physical records will not affect this index.
It can be used for performance improvements during batch insert/updates.
Remember to rebuild the index after the batch operation is finished.

IndexFieldList

Delphi declaration
```
IndexFieldList: TkbmFieldList FIndexFieldList
FIndexFieldList;
```

References the list of fields which is part of this index.

IndexFields

Delphi declaration
```
IndexFields: FIndexFields FIndexFields;
```

Contains the names of the fields which the index has been build upon.

IndexOfs

Delphi declaration
```
IndexOfs: integer FIndexOfs FIndexOfs;
```

Returns the absolute index into the TkbmIndexes list of this index.
Return value 0..Indexes.Count-1 or -1 if the index is not in the index list.

IndexOptions

Delphi declaration
```
IndexOptions: TkbmMemTableCompareOptions FIndexOptions
FIndexOptions;
```

Specifies the options for the index.
The options are working on the complete field, not on specific fields.
If options should be set on specific fields, the index should be created with a field list including field modifiers .

IndexType

Delphi declaration
```
IndexType: TkbmIndexType FType FType;
```

Specifies if the index is supposed to be sorted or not.

IsDirty

Delphi declaration
```
IsDirty: boolean FDirty FDirty;
```

IsFiltered

Delphi declaration
```
IsFiltered: boolean FIsFiltered FIsFiltered;
```

Specifies if the index is filtered or not.

IsFullScope

Delphi declaration
```
IsFullScope: boolean GetIsFullScope;
```

IsInternal

Delphi declaration
```
IsInternal: boolean FInternal FInternal;
```

Specifies if the index is an internal index or not.
The row order index is an internal index. Others are not.

IsOrdered

Delphi declaration
```
IsOrdered: boolean GetIsOrdered;
```

Specifies if the index is up to date and ordered.

IsRange

Delphi declaration
```
IsRange: boolean FIsRange FIsRange;
```

IsRowOrder

Delphi declaration
```
IsRowOrder: boolean FRowOrder FRowOrder;
```

Specifies if the index is the row order index or not.

IsSorted

Delphi declaration
```
IsSorted: boolean FSorted FSorted;
```

IsView

Delphi declaration
```
IsView: boolean FIsView FIsView;
```

Specifies if the index is a view or not.
The view model is not fully implemented, but the idea is that a view should not reflect changes to an index, except deletions.

Do not use for the time being.

Name

Delphi declaration
```
Name: FName FName;
```

The name of the index.

RangeEndKey

Delphi declaration
```
RangeEndKey: PkbmRecord FRangeEndKey SetRangeEndKey;
```

RangeStartKey

Delphi declaration
```
RangeStartKey: PkbmRecord FRangeStartKey
SetRangeStartKey;
```

References

Delphi declaration
```
References: TkbmList FReferences FReferences;
```

The list of physical record references ordered according to the rules of the index.

UpdateStatus

Delphi declaration
```
UpdateStatus: TUpdateStatusSet FUpdateStatus
FUpdateStatus;
```

Property that controls which update status filter should be applied to the index.

Public Methods/Functions

Clear

Delphi declaration
```
Clear;
```

Empty an index.
If you are switching to an empty index you will see no records in the table.
After emptying you need to rebuild the index.

See also:
Rebuild

Create

Delphi declaration

```
Create(ABase: TkbmIndex; IndexDef: TIndexDef; ADataSet:
TkbmCustomMemtable); ; ; ;
Create(ABase: TkbmIndex; : ; ADataSet:
TkbmCustomMemtable; Fields: ; Options:
TkbmMemTableCompareOptions; IndexType: TkbmIndexType);
; ; ;

Create(ABase: TkbmIndex; IndexDef: TIndexDef; ADataSet:
TkbmCustomMemtable); ; ; ;
Create(ABase: TkbmIndex; : ; ADataSet:
TkbmCustomMemtable; Fields: ; Options:
TkbmMemTableCompareOptions; IndexType: TkbmIndexType);
; ; ;
```

Create a new index with the specified *Name* on the given memory table *DataSet*.
Fields should contain the field names. Field modifiers may be used.
Options contain flags for how the index should operate.
Indextype specifies if the index should be maintained as a sorted or non sorted index.
Internal is true if the index is an internal index without any fieldnames associated.
The index will not hold physical records, but only references to the physical records in TkbmCommon .

CreateByIndexDef

Delphi declaration

```
constructorCreateByIndexDef(IndexDef: TIndexDef;
DataSet: TkbmCustomMemtable);
```

Creates a new index based on an TIndexDef object.

Destroy

Delphi declaration

```
Destroy; ;
```

Destructor of an index.

IsDependingOn

Delphi declaration

```
IsDependingOn(AIndex: TkbmIndex): boolean;
```

Load

Delphi declaration

```
Load;
```

LoadAll

Delphi declaration

```
LoadAll;
```

Method which enters all references to physical records into the index, except records which are filtered out if the index is a filtered index.
If any references are allready in the list of references, the list will be cleared.
After LoadAll, the index is not ordered, but rather in physical record order.
If the index is a sorted index, a ReSort should be issued.

Rebuild

Delphi declaration

```
Rebuild;
```

Rebuild the index from scratch.
It will clear the references from the index, reload them all (unless the index is marked with IsView=true) and automatically do a ReSort.

RebuildDepending

Delphi declaration

```
RebuildDepending;
```

ReSort

Delphi declaration

```
ReSort;
```

Will do a complete resort of the references in the index, and look for dupplicates if the index is described as being unique.
If any dupplicates are found in a unique index (can happen if indexes has been disabled during load operations), an error is raised.

Search

Delphi declaration

```
Search(FieldList: TkbmFieldList; KeyRecord:
PkbmRecord; SearchType: TkbmSearchType; PrimaryOnly,
RespectFilter: boolean; : TkbmNativeInt; Found:
boolean): TkbmNativeInt;
```

Internal support function which search for a record matching the KeyRecord.
Returns 0 if found exact match, <0 if key is smaller than found record or >0 if key is bigger than found record.
Index contain the absolute index of the record or -1 if no record is found.

SearchRecord

Delphi declaration

```
SearchRecord(KeyRecord: PkbmRecord; : TkbmNativeInt;
PrimaryOnly, RespectFilter: boolean): TkbmNativeInt;
```

Internal support function which searches for a specific record reference in the index.
Normally this would require a sequential scan, but if the index is ordered, it tries to narrow down the range of the sequential scan.
Its a very important function for updating the index on the fly.
Returns 0 if found exact match, <0 if key is smaller than found record or >0 if key is bigger than found record.
Index contain the absolute index of the record or -1 if no record is found.

SearchRecordID

Delphi declaration

```
SearchRecordID(RecordID: TkbmNativeInt; :
TkbmNativeInt): TkbmNativeInt;
```

Internal support function for searching for a specific unique record ID.
Its used extensively during update of the row order index, and resyncing on the same record after switch of index.

If the index is ordered, a binary search will be invoked, else a sequential scan.
Returns 0 if found exact match, <0 if key is smaller than found record or >0 if key is bigger than found record.
Index contain the absolute index of the record or -1 if no record is found.

Protected Methods/Functions

BinarySearch

Delphi declaration

```
BinarySearch(FieldList: TkbmFieldList; FirstNo,
LastNo: TkbmNativeInt; KeyRecord: PkbmRecord;
SearchType: TkbmSearchType; PrimaryOnly: boolean;
RespectFilter: boolean; : TkbmNativeInt; Found:
boolean): TkbmNativeInt;
```

Internal support function which will return an numeric index of a record matching the search criterias.
This function will perform a binary search, and thus the index has to be sorted for the search to operate correctly.
Usually do not use directly unless you know what you do.

BinarySearchRecordID

Delphi declaration

```
BinarySearchRecordID(FirstNo, LastNo: TkbmNativeInt;
RecordID: TkbmNativeInt; Desc: boolean; :
TkbmNativeInt): TkbmNativeInt;
```

Internal support function which will return an numeric index of a record matching the given unique record ID.
This function will perform a binary search.
Usually do not use directly unless you know what you do.

CompareRecords

Delphi declaration

```
CompareRecords(AFieldList: TkbmFieldList; AKeyRecord,
ARecord: PkbmRecord; APrimaryOnly: boolean;
ASortCompare, APartial, AUniqueConstraint: boolean):
Integer;
```

Internal support function which compares two records given with KeyRecord and ARecord on the fields given by the AFIeldList.
Its used for sorting and searching purposes.

Generally do not use unless you know what you are doing.

FastQuickSort

Delphi declaration

```
FastQuickSort(L, R: Integer);
```

Internal support method which make a very fast sort using a combination of a recursive Quicksort and an Insertion sort algorithm. It will sort thousands of record references in no time.

Filter

Delphi declaration

```
Filter(ARecord: PkbmRecord; ACheckRange: boolean):
boolean;
```

Internal support function which will return true if the specified record is accepted according to the optional filter put on the index. This function is a key element in filtered indexes .

FindRecordNumber

Delphi declaration

```
FindRecordNumber(ARecord: PkbmRecord): integer;
```

Internal support function which does a sequential scan (thus an expensive operation) to find the specified reference in the list of references in the index and return its relative number compared to the start of the index.
Its currently not used by either the index mechanisms or the memory table itself.

GetIsFullScope

Delphi declaration

```
GetIsFullScope: boolean;
```

GetIsOrdered

Delphi declaration

```
GetIsOrdered: boolean;
```

QuickSort

Delphi declaration

```
QuickSort(L, R: Integer);
```

Internal support method for quicksorting the references in the index.
This is normally not used by the index or the memory table.
To use this normal Quicksort method instead of the FastQuicksort, undefine the USE_FASTQUICKSORT flag in start of source of kbmmemtable.pas
There would normally be no reason to do so.

SequentialSearch

Delphi declaration

```
SequentialSearch(FieldList: TkbmFieldList; FirstNo,
LastNo: TkbmNativeInt; KeyRecord: PkbmRecord;
SearchType: TkbmSearchType; PrimaryOnly: boolean;
RespectFilter: boolean; SearchOrdered: boolean; :
TkbmNativeInt; Found: boolean): TkbmNativeInt;
```

Internal support function for performing a sequential scan for a record using the KeyRecord as search criteria.
It is used by SearchRecord in the case a binary search cannot be performed.

SequentialSearchRecordID

Delphi declaration

```
SequentialSearchRecordID(FirstNo, LastNo:
TkbmNativeInt; RecordID: TkbmNativeInt; :
TkbmNativeInt): TkbmNativeInt;
```

Internal support function for sequentially searching for a specific RecordID.
Its used by SearchRecordID in case a binary search is not possible, or as a combination with a binary search.

SetRangeEndKey

Delphi declaration

```
SetRangeEndKey(AKeyRecord: PkbmRecord);
```

SetRangeStartKey

Delphi declaration

```
SetRangeStartKey(AKeyRecord: PkbmRecord);
```

TkbmIndexes File: kbmMemTable

TkbmIndexes contains a collection of all the indexes that has been created on a specific memory table.
You can use the TkbmCustomMemTable .Indexes property to get to this instance.
Operations on this collection usually are reflected automatically to all participating TkbmIndex 'es.

Delphi declaration
```
TkbmIndexes=Class
```

Public Methods/Functions

Add

Delphi declaration
```
Add(IndexDef: TIndexDef);
```

This method will create a new TkbmIndex using the specified IndexDef and add the index to the collection of indexes.

See also:
AddIndex

AddIndex

Delphi declaration
```
AddIndex(: TkbmIndex);
```

Add an allready created index to the collection of indexes.

See also:
Add

CheckRecordUniqueness

Delphi declaration
```
CheckRecordUniqueness(ARecord, ActualRecord:
PkbmRecord);
```

Method for checking if *ARecord* is violating index uniqueness for any index.
ActualRecord is the current record of the table.
If *ARecord* violates any index uniqueness, an error will be raised.

Clear

Delphi declaration
```
Clear;
```

The Clear method will clear all defined indexes, and unallocate them, except for the row order index.
The row order index will be the current index of the memory table.

Count

Delphi declaration
```
Count: integer;
```

Returns the number of indexes in the index collection.

Create

Delphi declaration
```
Create(ADataSet: TkbmCustomMemTable); ; ;
```

Constructor of the index collection.
Creating an index collection also binds it to the memory table.

Delete

Delphi declaration
```
Delete(IndexName: );
```

Removes an index specified with the index name from the collection of indexes, but doesnt deallocate its memory.

See also:
DeleteIndex

DeleteIndex

Delphi declaration
```
DeleteIndex(: TkbmIndex);
```

Removes an index specified with the indexnumber from the collection of indexes, but doesnt deallocate its memory.

See also:
Delete

Destroy

Delphi declaration
```
Destroy; ;
```

Destructor of TkbmIndexes.

All participant indexes in the collection will also be cleared and deallocated.

Empty

Delphi declaration
```
Empty(IndexName: );
```

Empty the contents of the index specified by name.

See also:
EmptyAll

EmptyAll

Delphi declaration
```
EmptyAll;
```

Empty the contents of all indexes.

See also:
Empty

Get

Delphi declaration
```
Get(IndexName: ): TkbmIndex;
```

Returns an index from the index collection by searching the index name.
If no index is found, nil is returned.

See also:
GetByFieldNames

GetByFieldNames

Delphi declaration
```
GetByFieldNames(FieldNames: ): TkbmIndex;
```

Returns an index from the index collection by searching the fieldnames comprising the index.
If no index is found, nil is returned.

See also:
Get

GetIndex

Delphi declaration
```
GetIndex(Ordinal: integer): TkbmIndex;
```

Get a specific index by value from 0 to Count-1.

MarkAllDirty

Delphi declaration
```
MarkAllDirty;
```

Mark all indexes as dirty.
That means that next time UpdateIndexes will be called, the indexes will be rebuild.

See also:
TkbmIndex.Rebuild

ReBuild

Delphi declaration
```
ReBuild(IndexName: );
```

Rebuild the specific index.

ReBuildAll

Delphi declaration
```
ReBuildAll;
```

Rebuild all indexes in the collection.

ReflectToIndexes

Delphi declaration
```
ReflectToIndexes(How: TkbmIndexUpdateHow; OldRecord,
NewRecord: PkbmRecord; RecordPos: TkbmNativeInt;
DontVersion: boolean);
```

Internal support method to do on the fly update on all indexes according to How parameter.
Its called on every insert/modification and deletion of a physical record.

Search

Delphi declaration

```
Search(FieldList: TkbmFieldList; KeyRecord:
PkbmRecord; SearchType: TkbmSearchType; PrimaryOnly,
RespectFilter, AutoAddIdx: boolean; : TkbmNativeInt;
Found: boolean): TkbmNativeInt;
```

Search for a specific record using the specified fields as key fields and the contents of *KeyRecord* as the values to search for.
If the field list matches an index, that index will be used for the search.
Returns 0 if found exact match, <0 if key is smaller than found record or >0 if key is bigger than found record.
Index contain the absolute index of the record or -1 if no record is found.

Protected Methods/Functions

IndexFreed

Delphi declaration

```
IndexFreed(AIndex: TkbmIndex); ;
```

TkbmIndexType File: kbmMemTable

Value	Meaning
mtitNonSorted:	
mtitSorted:	

Delphi declaration

```
TkbmIndexType =(mtitNonSorted, mtitSorted);
```

TkbmIndexUpdateHow File: kbmMemTable

Value	Meaning
mtiuhInsert:	
mtiuhEdit:	
mtiuhDelete:	

Delphi declaration

```
TkbmIndexUpdateHow =(mtiuhInsert, mtiuhEdit, mtiuhDelete);
```

TkbmList File: kbmMemTable

A specialy optimized TList which performs very well when manipulating large lists.
Its available through the kbmMemTable Pro Additions package.

Delphi declaration

```
TkbmList=TList;
```

TkbmLocaleID File: kbmMemTable

LocaleID is a integer number calculated with the following algorithm:

LocaleID:= (SortID shl 16) or (word(SubLanguageID) shl 10) or word(LanguageID);

The LocaleID specifies how strings in a certain language should be sorted and is used in CompareRecords.

Delphi declaration

```
TkbmLocaleID=integer;
```

TkbmMasterDataLink File: kbmMemTable

Delphi declaration

```
TkbmMasterDataLink=Class(TMasterDataLink);
```

Protected Methods/Functions

RecordChanged

Delphi declaration

```
RecordChanged(Field: TField); ;
```

TkbmMemSQL File: kbmMemSQL

Delphi declaration
```
TkbmMemSQL=Class(TkbmCustomMemTable);
```

Public Properties/Variables

Context

Delphi declaration
```
Context: TObject FContext FContext;
```

Parser

Delphi declaration
```
Parser: TkbmSQLParser FParser;
```

Tables

Delphi declaration
```
Tables: TkbmSQLTables FTables;
```

Public Methods/Functions

Calculate

Delphi declaration
```
Calculate(AExpression: ; ASyntaxCheckOnly:
boolean=false; AOptions:
TkbmSQLExpressionOptions=[seoOnlyNumericExpressions])
: variant;
```

Create

Delphi declaration
```
Create(AOwner: TComponent); ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Evaluate

Delphi declaration

```
Evaluate(AExpression: ; ASyntaxCheckOnly:
boolean=false; AOptions:
TkbmSQLExpressionOptions=[seoOnlyNumericExpressions])
: variant;
```

ExecSQL

Delphi declaration

```
ExecSQL(ASQL: );
```

Protected Methods/Functions

DoOnError

Delphi declaration

```
DoOnError(Sender: TObject; AErrorType, AErrorCode,
ALine, ACol: integer; AMsg, AData: ); ;
```

FixupFields

Delphi declaration

```
FixupFields;
```

FixupRefSourceTables

Delphi declaration

```
FixupRefSourceTables; ;
```

FixupRefTables

Delphi declaration

```
FixupRefTables; ;
```

ResetAffectedRows

Delphi declaration

```
ResetAffectedRows; ;
```

OnError

Delphi declaration

```
OnError: TkbmSQLOnError FOnError FOnError;
```

OnGetFunction

Delphi declaration

```
OnGetFunction: TkbmSQLOnGetFunction FOnGetFunction
FOnGetFunction;
```

OnGetVariableMetaData

Delphi declaration

```
OnGetVariableMetaData: TkbmSQLOnGetVariableMetaData
FOnGetVariableMetaData FOnGetVariableMetaData;
```

OnGetVariableValue

Delphi declaration

```
OnGetVariableValue: TkbmSQLOnGetVariableValue
FOnGetVariableValue FOnGetVariableValue;
```

TkbmMemTable File: kbmMemTable

TkbmMemTable is the published version of TkbmCustomMemTable.
If you at any time would make your own descendant based on the memory table, you would normally inherit from TkbmCustomMemTable and not TkbmMemTable. This way you can yourself decide which properties should be published and which should not.
TkbmMemTable adds no new functionality to the memory table classes, but simply publishes several protected properties.

Delphi declaration
```
TkbmMemTable=Class(TkbmCustomMemTable);
```

See Also:
TkbmCustomMemTable

Public Properties/Variables

FormatSettings

Delphi declaration
```
FormatSettings: TFormatSettings FFormatSettings
FFormatSettings;
```

Public Events

TkbmMemTableCompareOption File: kbmMemTable

Value	Meaning
mtcoDescending:	
mtcoCaseInsensitive:	
mtcoPartialKey:	
mtcoIgnoreNullKey:	
mtcoIgnoreLocale:	
mtcoUnique:	
mtcoNonMaintained:	

Delphi declaration

```
TkbmMemTableCompareOption =(mtcoDescending, mtcoCaseInsensitive, mtcoPartialKey,
             mtcoIgnoreNullKey, mtcoIgnoreLocale, mtcoUnique, mtcoNonMaintained);
```

TkbmMemTableCompareOptions File: kbmMemTable

A set of compare options.

Delphi declaration

```
TkbmMemTableCompareOptions=TkbmMemTableCompareOption;
```

Delphi Example

```
var
  co:TkbmMemTableCompareOptions;
begin
  co:=[mtcoCaseInsensitive,mtcoPartial];
end;
```

TkbmMemTableCopyTableOption File: kbmMemTable

Value	Meaning
mtcpoStructure:	
mtcpoOnlyActiveFields:	
mtcpoProperties:	
mtcpoLookup:	
mtcpoCalculated:	
mtcpoAppend:	
mtcpoFieldIndex:	
mtcpoDontDisableIndexes:	
mtcpoIgnoreErrors:	
mtcpoLookupAsData:	
mtcpoCalculatedAsData:	
mtcpoStringAsWideString:	
mtcpoWideStringUTF8:	

Delphi declaration

```
TkbmMemTableCopyTableOption =(mtcpoStructure, mtcpoOnlyActiveFields, mtcpoProperties,
             mtcpoLookup, mtcpoCalculated, mtcpoAppend, mtcpoFieldIndex,
             mtcpoDontDisableIndexes, mtcpoIgnoreErrors, mtcpoLookupAsData,
             mtcpoCalculatedAsData, mtcpoStringAsWideString, mtcpoWideStringUTF8);
```

TkbmMemTableCopyTableOptions File: kbmMemTable

A set of copy options.

Delphi declaration

```
TkbmMemTableCopyTableOptions=TkbmMemTableCopyTableOption;
```

Delphi Example

```
var
  co:TkbmMemTableCopyTableOptions;
begin
  co:=[mtcpoStructure,mtcpoProperties];
end;
```

TkbmMemTablePersistentSaveFormat File: kbmMemTable

Value	Meaning
mtsfBinary	Use binary file format during save.
mtsfCSV	Use CSV text file format during save.

Delphi declaration

```
TkbmMemTablePersistentSaveFormat=(mtsfBinary, mtsfCSV);
```

TkbmMemTableSaveFlag File: kbmMemTable

Value	Meaning
mtfSaveData	Save the data of the table.
mtfSaveCalculated	Save the contents of calculated fields.
mtfSaveLookup	Save the contents of lookup fields.
mtfSaveNonVisible	Save the contents of non visible fields.
mtfSaveBlobs	Save the contents of blob fields.
mtfSaveDef	Save the definition/structure of the table.
mtfSaveIndexDef	Save the table's index definitions.
mtfSkipRest	Dont generate empty spots in the CSV file for nonsaved fields.
mtfSaveFiltered	Save records even if they are filtered out.
mtfSaveInLocalFormat	Save fields using user definable date/time and numeric formats. See ERROR: INVALID LINK: !Alink(Sysutils.ShortDateFormat,1) ShortDateFormat . If not specified the following CSV formats will be used: DateSeparator:='/'; TimeSeparator:=':'; ThousandSeparator:=','; DecimalSeparator:='.'; ShortDateFormat:='dd/mm/yyyy'; CurrencyString:=''; CurrencyFormat:=0; NegCurrFormat:=1;
mtfSaveIgnoreRange	Save records even if they are outside the currently defined range.
mtfSaveIgnoreMasterDetail	Save records even if a M/D relationship is defined and the records are not part of the detail filter.
mtfSaveDeltas	Save all versions of records. Important if you want to copy the deltas from a client to a server for later resolving.
mtfSaveDontFilterDeltas	Normally only the relevant deltas are saved. Thus if a record has been inserted and then again deleted since same checkpoint, it will not be saved unless this flag is specified. The reason is that its normally a waste of work for the deltahandler to insert the record and then remove it again.
mtfSaveAppend	If not set the file or stream will be overwritten.
mtfSaveFieldKind	If set, also the field kind will be saved. Its only interesting when specified with mtfSaveDef. Default behaviour is to mark all fields as data fields in the save stream/file.
mtfSaveNoHeader	Dont create a header in the CSV save stream/file. Normally a line consisting of the field names is saved before the data.

Delphi declaration

```
TkbmMemTableSaveFlag=(mtfSaveData, mtfSaveCalculated, mtfSaveLookup,
mtfSaveNonVisible, mtfSaveBlobs, mtfSaveDef, mtfSaveIndexDef, mtfSkipRest,
mtfSaveFiltered, mtfSaveInLocalFormat, mtfSaveIgnoreRange, mtfSaveIgnoreMasterDetail,
mtfSaveDeltas, mtfSaveDontFilterDeltas, mtfSaveAppend, mtfSaveFieldKind,
mtfSaveNoHeader);
```

TkbmMemTableSaveFlags File: kbmMemTable

A set of save flags.

Several standard sets of save flags has also been provided:
kbmSaveAllDeltas =
[mtfSaveNonVisible,mtfSaveBlobs,mtfSaveDef,mtfSaveFiltered,mtfSaveIgnoreRange,
mtfSaveIgnoreMasterDetail,mtfSaveDeltas]

kbmSaveAllData =
[mtfSaveNonVisible,mtfSaveBlobs,mtfSaveDef,mtfSaveFiltered,mtfSaveIgnoreRange,
mtfSaveIgnoreMasterDetail,mtfSaveData]

kbmSaveAllDataAndDelta =
[mtfSaveNonVisible,mtfSaveBlobs,mtfSaveDef,mtfSaveFiltered,mtfSaveIgnoreRange,
mtfSaveIgnoreMasterDetail,mtfSaveData,mtfSaveDeltas]

Delphi declaration
TkbmMemTableSaveFlags=set of TkbmMemTableSaveFlag;

TkbmMemTableStorageType File: kbmMemTable

Value	Meaning
mtstDataSet:	
mtstStream:	
mtstBinaryStream:	
mtstFile:	
mtstBinaryFile:	

Delphi declaration

```
TkbmMemTableStorageType =(mtstDataSet, mtstStream, mtstBinaryStream, mtstFile,
             mtstBinaryFile);
```

TkbmMemTableUpdateFlag File: kbmMemTable

Controls how records of another dataset is updated based on the current memtable.

Value	Meaning
mtufEdit:	
mtufAppend:	
mtufDontClear:	

Delphi declaration

```
TkbmMemTableUpdateFlag =(mtufEdit, mtufAppend, mtufDontClear);
```

TkbmMemTableUpdateFlags File: kbmMemTable

A set of update flags.

Delphi declaration

```
TkbmMemTableUpdateFlags=TkbmMemTableUpdateFlag;
```

TkbmNativeInt File: kbmMemTypes

Delphi declaration

```
TkbmNativeInt=integer;
```

TkbmOnCompareFields File: kbmMemTable

The OnCompareFields event is called during comparison of two fields.
If you have special requirements for a specific field, you can write an event handler which takes care of that situation, and use standard behaviour on the rest of the fields by calling CompareFields method.
DataSet is the memory table which called the event handler.
KeyField is the raw data of the field in the key record. Remember that 1st. byte in the raw data of a field is a null/not null byte. The layout of the rest of the field is dependent on the fieldtype.
AField is the the raw data of the field in the record to compare with.
FieldType is the type of the field.
Options are a bitmask of index field comparison options. See TkbmFieldList.Add .
Result should contain <0 if the KeyField<AField, >0 if KeyField>AField and 0 if KeyField=AField
If the search is on a descending index, the result should be negated.

Delphi declaration

```
TkbmOnCompareFields=(DataSet: TDataSet; AFld: TField; KeyField, AField:
pointer; FieldType: TFieldType; Options: TkbmifoOptions; FullCompare:
boolean; Result: integer) ;
```

Delphi Example

```
procedure TForm1.kbmMemTable1CompareFields(KeyField,AField:pointer; FieldType:
TFieldType; LocaleID:TkbmLocaleID; IndexFieldOptions:integer; var Result:integer);
begin
  Result:=CompareFields(KeyField,AField,FieldType,LocaleID,IndexFieldOptions);
  if FieldType=ftDate then Result:=-Result;       // Special treatment of a date field.
end;
```

TkbmOnCompress File: kbmMemTable

If an OnCompress event handler has been written, it will be called during save to a stream or file.
Its purpose is to allow compressing or encrypting the data before the save.
The structure is pretty simple:
- read from the UnCompressedStream
- compress what have been read
- write to the CompressedStream

Delphi declaration

```
TkbmOnCompress=(Dataset: TkbmCustomMemTable; UnCompressedStream,
CompressedStream: TStream) ;
```

Delphi Example

```
procedure TForm1.kbmMemTable1CompressSave(UnCompressedStream,
CompressedStream: TStream);
begin
      Screen.Cursor := crHourglass;
      Application.ProcessMessages;
      try
            LZHCompressSave(UnCompressedStream,CompressedStream)
      finally
            Screen.Cursor:=crDefault;
      end;
end;
```

LZHCompressSave is found in the unsupported file kbmCompress.pas.

See Also:
TkbmOnDeCompress

TkbmOnCompressField File: kbmMemTable

If this event is assigned, it will be called for every field which has been marked as a compressed field.
Your job will be to decide how to compress the field, and then do the actual compression on the delivered data in the *Buffer*. The original size of the data is given in the *Size* parameter.
You need to return the new size in bytes in the *Size* parameter and a pointer to a buffer where the compressed data can be found. Default *ResultBuffer* points to the same area as *Buffer*.
If you want to use the default compression mechanism, you can pass the call to it:ne

Common.CompressFieldBuffer(Field,Buffer,Size)

Delphi declaration

```
TkbmOnCompressField=(DataSet: TDataSet; Field: TField; Buffer: PAnsiChar; Size: longint;
ResultBuffer: PAnsiChar) ;
```

See Also:
TkbmCommon.CompressFieldBuffer TkbmOnDecompressField

TkbmOnDeCompress File: kbmMemTable

If an OnDeCompress event handler has been written, it will be called during load from a stream or file.
Its purpose is to allow decompressing or unencrypting the data before the load.
The structure is pretty simple:
- read from the CompressedStream
- uncompress what have been read
- write to the DeCompressedStream

Delphi declaration

```
TkbmOnDeCompress=(Dataset: TkbmCustomMemTable; CompressedStream,
DeCompressedStream: TStream) ;
```

Delphi Example

```
procedure TForm1.kbmMemTable1DecompressLoad(CompressedStream,
DecompressedStream: TStream);
begin
      Screen.Cursor := crHourglass;
      Application.ProcessMessages;
      try
            LZHDeCompressLoad(CompressedStream,DecompressedStream)
      finally
            Screen.Cursor:=crDefault;
      end;
end;
```

LZHDecompressLoad is found in the unsupported file kbmCompress.pas.

See Also:
TkbmOnCompress

TkbmOnDecompressField File: kbmMemTable

If this event is assigned, it will be called for every field which has been marked as a compressed field.
Your job will be to decide how to decompress the field, and then do the actual decompression on the delivered data in the *Buffer*. The size of the compressed data is given in the *Size* parameter.
You need to return the new size in bytes in the *Size* parameter and a pointer to a buffer where the decompressed data can be found. Default *ResultBuffer* points to the same area as *Buffer*.
If you want to use the default decompression mechanism, you can pass the call to it:ne

Common.DecompressFieldBuffer(Field,Buffer,Size)

Delphi declaration

```
TkbmOnDecompressField=(DataSet: TDataSet; Field: TField; Buffer: PAnsiChar; Size: longint;
ResultBuffer: PAnsiChar) ;
```

See Also:
TkbmCommon.DecompressFieldBuffer TkbmOnCompressField

TkbmOnFilterIndex File: kbmMemTable

This eventhandler works pretty much the same as the OnFilterRecord event.
The difference is that the purpose of this event is to physically filter out records from an index where the ERROR: INVALID LINK: !Alink(TDataset.OnFilterRecord,1) OnFilterRecord is merely a virtual filter in the sense that no records are removed from the index, but instead just not shown at the time.
During the OnFilterIndex event the current record you can obtain the fieldvalues from the current record of the dataset using the standard TField.As.... methods.
If you decide to keep a record in the index, set *Accept* to true, otherwise set it to false.
The *Index* parameter is the index which is currently in a need to know if the record is accepted or not.

Delphi declaration

```
TkbmOnFilterIndex=(DataSet: TDataSet; : TkbmIndex; Accept: boolean) ;
```

Delphi Example
Example 1 - Only allow records with Fld1 < 50 in the filtered index myIndex. Accept record for all other filtered indexes.

```
procedure TForm1.kbmMemTable1FilterIndex(DataSet: TDataSet;
  Index: TkbmIndex; var Accept: Boolean);
begin
      Accept:=(Index<>myIndex) or (DataSet.FieldByName('Fld1').AsInteger<50);
end;
```

See Also:
TkbmCustomMemTable.AddFilteredIndex

TkbmOnFormatLoadField File: kbmMemCSVStreamFormat

This event is called when loading a CSV file.
It can be used to change the format of a loaded field value before the value is entered into the memtable.

Delphi declaration

```
TkbmOnFormatLoadField=(Sender: TObject; Field: TField; Null: boolean;
Data: );
```

See Also:
TkbmCustomCSVStreamFormat

TkbmOnFormatSaveField File: kbmMemCSVStreamFormat

This event is called when saving the memtable to a CSV file.
It can be used to change the format of the data to write to the CSV file for the specified field.

Delphi declaration

```
TkbmOnFormatSaveField=(Sender: TObject; Field: TField; Null: boolean;
Data: );
```

See Also:
TkbmCustomCSVStreamFormat

TkbmOnGetLine File: kbmMemCSVStreamFormat

Delphi declaration

```
TkbmOnGetLine=(Sender: TObject; Line: ; EOF: boolean) ;
```

TkbmOnLoad File: kbmMemTable

This event is called before every data load operation including LoadFromStream, LoadFromFile, LoadFromBinaryStream, LoadFromBinaryFile and LoadFromDataSet.

In all cases except LoadFromDataSet, the *Stream* parameter is the stream from which the load is going to happen.
For LoadFromDataSet, the *Stream* parameter will be nil.
StorageType gives you a hint about which of the load methods invoked this event.

Delphi declaration

```
TkbmOnLoad=(DataSet: TDataSet; StorageType: TkbmMemTableStorageType;
Stream: TStream) ;
```

See Also:

TkbmCustomMemTable.LoadFromBinaryFile TkbmCustomMemTable.LoadFromBinaryStream TkbmCustomMemTable.LoadFromDataSet TkbmCustomMemTable.LoadFromFile TkbmCustomMemTable.LoadFromStream TkbmOnSave

TkbmOnLoadField File: kbmMemTable

This event is called just right after a field has been populated with a value during a load operation during any of the methods LoadFromFile, LoadFromBinaryFile, LoadFromDataset, LoadFromStream, LoadFromBinaryStream, CopyRecords and AssignRecord, and before the record in which the field is loaded, is posted.

It gives you the opertunity to change the value of the field before storing it in the table if you so desire.

Delphi declaration

```
TkbmOnLoadField=(DataSet: TDataSet; FieldNo: integer; Field: TField) ;
```

See Also:

TkbmOnSaveField

TkbmOnLoadRecord File: kbmMemTable

This event is called after all fields for a record has been loaded, but just before the record is posted.
By setting the *Accept* to either true or false, you can control which records should be loaded into the memory table from an external source.
You can check the methods TField.As.... for the values of the record during this event.

Delphi declaration

```
TkbmOnLoadRecord=(DataSet: TDataSet; Accept: boolean) ;
```

See Also:
TkbmOnSaveRecord

TkbmOnProgress File: kbmMemTable

This event is called during many potentially long running operations at relatively regular intervals to give you the posibility to f.ex. inform the user about the progress of the operation.
See **TkbmProgressCode** for the operations which it will be called for.
You can also yourself call the *Progress* method in your own long running operations, although make sure to do it outside this event handler, or you would risc having an recursive endless loop.
Percentage is an integer usually between 0 and 100 giving a hint about how far the operation has proceeded.
Code gives you the type of operation which is informing you.
TkbmCustomMemTable.ProgressFlags governs during which operations the event should be called.

Delphi declaration

```
TkbmOnProgress=(DataSet: TDataSet; Percentage: integer; Code:
TkbmProgressCode) ;
```

See Also:
TkbmCustomMemTable.Progress

TkbmOnSave File: kbmMemTable

This event is called right before every data save operation including SaveToStream, SaveToFile, SaveToBinaryStream, SaveToBinaryFile and SaveToDataSet.

In all cases except SaveToDataSet, the *Stream* parameter is the stream to which the save is going.
For SaveToDataSet, the *Stream* parameter is be nil.
StorageType gives you a hint about which of the save methods invoked this event.

Delphi declaration

```
TkbmOnSave=(DataSet: TDataSet; StorageType: TkbmMemTableStorageType;
Stream: TStream) ;
```

See Also:

TkbmCustomMemTable.SaveToBinaryFile TkbmCustomMemTable.SaveToBinaryStream TkbmCustomMemTable.SaveToDataSet TkbmCustomMemTable.SaveToFile TkbmCustomMemTable.SaveToStream TkbmOnLoad

TkbmOnSaveField File: kbmMemTable

This event is called just right before a field is being saved using any of the methods SaveToFile, SaveToBinaryFile, SaveToDataset, SaveToStream, SaveToBinaryStream, CopyRecords and AssignRecord.

It gives you the opertunity to f.ex. change the value of the field before saving it if you so desire.

Delphi declaration

```
TkbmOnSaveField=(DataSet: TDataSet; FieldNo: integer; Field: TField) ;
```

See Also:

TkbmOnLoadField

TkbmOnSaveRecord File: kbmMemTable

This event is called right before any fields of a record is going to be saved.
By setting the *Accept* to either true or false, you can control which records should be saved to the external source.
You can check the methods TField.As.... for the values of the record during this event.

Delphi declaration

```
TkbmOnSaveRecord=(DataSet: TDataSet; Accept: boolean) ;
```

See Also:
TkbmOnLoadRecord

TkbmOnSetupField File: kbmMemTable

This event is called during opening a memory table at the time when the physical record layout is being determined.
Writing an eventhandler here will allow you to control the otherwise automatic rules about which fields should be indirect fields and which should be compressed.
Set *FieldFlags* to determine how a field should be stored.
Please see OnSetupField .

Delphi declaration

```
TkbmOnSetupField=(DataSet: TDataSet; Field: TField; FieldFlags: byte) ;
```

TkbmOnSetupFieldProperties File: kbmMemTable

This event is called when the properties of a field is setup.
Properties are such as DisplayName, EditFormat etc.
CreateTableAs, AttachedTo and LoadFromDataset with the load option mtcpoProperties will trigger this event.
It can be used to setting properties to other values than those copied.

Delphi declaration

```
TkbmOnSetupFieldProperties=(DataSet: TDataSet; Field: TField) ;
```

See Also:

TkbmCustomMemTable.AttachedTo TkbmCustomMemTable.CopyFieldProperties
TkbmCustomMemTable.CreateTableAs TkbmCustomMemTable.LoadFromDataSet

TkbmOnUserAggregate File: kbmMemTable

Delphi declaration

```
TkbmOnUserAggregate=(DataSet: TDataSet; UserFunction: integer; Count:
integer; Value: double; Accumulator: double) ;
```

TkbmPerformance File: kbmMemTable

Value	Meaning
mtpfFast:	
mtpfBalanced:	
mtpfSmall:	

Delphi declaration

```
TkbmPerformance =(mtpfFast, mtpfBalanced, mtpfSmall);
```

TkbmProgressCode File: kbmMemTable

Value	Meaning
mtpcLoad:	
mtpcSave:	
mtpcEmpty:	
mtpcPack:	
mtpcCheckPoint:	
mtpcSearch:	
mtpcCopy:	
mtpcUpdate:	
mtpcSort:	

Delphi declaration

```
TkbmProgressCode =(mtpcLoad, mtpcSave, mtpcEmpty, mtpcPack, mtpcCheckPoint, mtpcSearch,
             mtpcCopy, mtpcUpdate, mtpcSort);
```

TkbmProgressCodes File: kbmMemTable

A set of progress codes.

Delphi declaration

```
TkbmProgressCodes=TkbmProgressCode;
```

TkbmRecord File: kbmMemTypes

This record type contains the complete layout of the static part of a record.
Every time a new record is allocated, it is returned as a pointer to a TkbmRecord.
By defining the flag **DO_CHECKRECORD** at the start of the *kbmmemtable.pas* file and doing a recompilation, all records will be checked before and often also after use to see no memory corruption has happened.
This check is made by checking 4 special bytes in the start and end of the record. If they dont contain the expected value, some kind of memory corruption has happened. Its not a completely failsafe check, and memory corruption can occur without the checking routines discover it.
Generally do not deliver a production system with this flag set since it will slow down operations slightly and make each record buffer take up 8 bytes more space.
The size of the structure allocated is not simply the size of TkbmRecord.
During table open, the actual size of all the fields of a record plus bookmark handling etc. is calculated, and added to the size of TkbmRecord. And that is what is allocated in one chunk.
The record entry Data will then be set to point at the field and bookmark data which is placed right after the EndIdent entry in memory.

Delphi declaration
```
TkbmRecord=
```

Public Properties/Variables

Data

Delphi declaration
```
Data: PAnsiChar;
```

The Data entity is a pointer to the actual field and bookmark part of a record.

EndIdent

Delphi declaration
```
EndIdent: longint;
```

This is an identifier which can be checked for a specific value to see if a record has been corrupted.

Flag

Delphi declaration
```
Flag: byte;
```

This flag can contain a combination of the following values:

```
kbmrfInTable       = $01;  // 0000 0001 Is record a work record or
actually from the table.ne kbmrfDontCheckPoint= $02;  // 0000 0010 Is
record marked for not to checkpoint.
```

PrevRecordVersion

Delphi declaration

```
PrevRecordVersion: PkbmRecord;
```

This is a pointer to the previous version of the record if any is available. PrevRecordVersion points to another TkbmRecord structure which again can point to another etc. if the record has been modified many times and Versioning is enabled.

Its extensively used by versioning, transactions and delta handlers.

RecordID

Delphi declaration

```
RecordID: TkbmNativeInt;
```

The record is labeled with a unique number within the table.
It is not guaranteed that the number has not been used before on a allready deleted record.
The RecordID can be used as an index directly into the physical table of records.(TkbmCommon.FRecords).
It has a value of -1 if the record has not been stored in the table, although a value different from -1 is no guarantee that the record has not been stored. See the ERROR: INVALID LINK: kbmMemTable.TkbmRecord.InTable InTable flag.

RecordNo

Delphi declaration

```
RecordNo: TkbmNativeInt;
```

This value contains a very volatile value which is updated on each record fetch.
It can be used to return information about the actual RecNo of the specified record according to the current index.

StartIdent

Delphi declaration

```
StartIdent: longint;
```

Tag

Delphi declaration

```
Tag: longint;
```

This contains any information of max. 4 bytes which you would like to attach to a record.
Notice that using it for a pointer to some allocated memory can be unsafe, unless you remember to free the memory on every delete of a record.
Generally only use it for tagging a record with a special reference number of yours which you dont want to create a field for.

TransactionLevel

Delphi declaration

```
TransactionLevel: integer;
```

This value is used to determine which transaction the record is belonging to.
Its extensively used by versioning, transactions and delta handlers.

UniqueRecordID

Delphi declaration

```
UniqueRecordID: TkbmNativeInt;
```

This contains a session wide unique counter which is 'never' repeated betweeo an open and close of a memory table.
Never is in quotation marks because if more than 2147483600 records are added to the table during its open life without any close, it will overflow and raise an error.

UpdateStatus

Delphi declaration

```
UpdateStatus: TUpdateStatus;
```

This contains the status for the current record. Was it modified, deleted, inserted or is it unmodified?
Its extensively used by versioning, transactions and delta handlers.

TkbmSearchType File: kbmMemTable

Value	Meaning
mtstFirst:	
mtstLast:	
mtstNearestBefore:	
mtstNearestAfter:	

Delphi declaration

```
TkbmSearchType =(mtstFirst, mtstLast, mtstNearestBefore, mtstNearestAfter);
```

TkbmSQLAggregateFunction File: kbmSQLElements

Value	Meaning
safUNKNOWN:	
safCOUNT:	
safMAX:	
safMIN:	
safAVG:	
safSUM:	

Delphi declaration

```
TkbmSQLAggregateFunction =(safUNKNOWN, safCOUNT, safMAX, safMIN, safAVG, safSUM);
```

TkbmSQLAggregateNode File: kbmSQLElements

Delphi declaration
```
TkbmSQLAggregateNode=Class(TkbmSQLCustomNode);
```

Public Properties/Variables

AggrFunction

Delphi declaration
```
AggrFunction: TkbmSQLAggregateFunction FAggrFunction
FAggrFunction;
```

Distinct

Delphi declaration
```
Distinct: boolean FDistinct FDistinct;
```

ExpressionNode

Delphi declaration
```
ExpressionNode: TkbmSQLCustomNode FExpressionNode
FExpressionNode;
```

FieldModifier

Delphi declaration
```
FieldModifier: GetFieldModifier;
```

FieldOptions

Delphi declaration
```
FieldOptions: TkbmifoOptions GetFieldOptions;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOperation: TkbmSQLCustomOperation); ;
```

Destroy

Delphi declaration

```
Destroy; ;
```

Execute

Delphi declaration

```
Execute: variant; ;
```

GetName

Delphi declaration

```
GetName: ; ;
```

GetUniqueName

Delphi declaration

```
GetUniqueName: ; ;
```

Protected Methods/Functions

GetDataType

Delphi declaration

```
GetDataType: TFieldType; ;
```

GetFieldModifier

Delphi declaration

```
GetFieldModifier: ; ;
```

GetFieldOptions

Delphi declaration

```
GetFieldOptions: TkbmifoOptions; ;
```

GetWidth

Delphi declaration

```
GetWidth: integer; ;
```

TkbmSQLAllFields File: kbmSQLElements

Delphi declaration
```
TkbmSQLAllFields=Class(TkbmSQLField);
```

Public Methods/Functions

Create

Delphi declaration
```
Create; ;
```

Protected Methods/Functions

SetAlias

Delphi declaration
```
SetAlias(AAlias: ); ;
```

SetName

Delphi declaration
```
SetName(AName: ); ;
```

TkbmSQLBetweenNode File: kbmSQLElements

Delphi declaration
```
TkbmSQLBetweenNode=Class(TkbmSQLCustomNode);
```

Public Properties/Variables

LeftNode

Delphi declaration
```
LeftNode: TkbmSQLCustomNode FLeftNode FLeftNode;
```

RangeHigh

Delphi declaration
```
RangeHigh: TkbmSQLCustomNode FRangeHigh FRangeHigh;
```

RangeLow

Delphi declaration
```
RangeLow: TkbmSQLCustomNode FRangeLow FRangeLow;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOperation: TkbmSQLCustomOperation); ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Execute

Delphi declaration
```
Execute: variant; ;
```

Protected Methods/Functions

GetDataType

Delphi declaration
```
GetDataType: TFieldType; ;
```

TkbmSQLBinaryNode File: kbmSQLElements

Delphi declaration
```
TkbmSQLBinaryNode=Class(TkbmSQLCustomNode);
```

Public Properties/Variables

LeftNode

Delphi declaration
```
LeftNode: TkbmSQLCustomNode FLeftNode FLeftNode;
```

Operator

Delphi declaration
```
Operator: TkbmSQLBinaryNodeOperator FOperator;
```

RightNode

Delphi declaration
```
RightNode: TkbmSQLCustomNode FRightNode FRightNode;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOperation: TkbmSQLCustomOperation; AOperator:
TkbmSQLBinaryNodeOperator); ; ;
Create(AOperation: TkbmSQLCustomOperation; AOperator:
TkbmSQLBinaryNodeOperator); ; ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Execute

Delphi declaration
```
Execute: variant; ;
```

GetDataType

Delphi declaration

```
GetDataType: TFieldType; ;
```

GetWidth

Delphi declaration

```
GetWidth: integer; ;
```

TkbmSQLBinaryNodeOperator File: kbmSQLElements

Value	Meaning
ebAdd:	
ebSub:	
ebMul:	
ebDiv:	
ebAnd:	
ebOr:	
ebXor:	
ebMod:	
ebIDiv:	
ebLike:	
ebEqual:	
ebNotEqual:	
ebLess:	
ebGreater:	
ebLessEqual:	
ebGreaterEqual:	

Delphi declaration

```
TkbmSQLBinaryNodeOperator =(ebAdd, ebSub, ebMul, ebDiv, ebAnd, ebOr, ebXor, ebMod, ebIDiv,
             ebLike, ebEqual, ebNotEqual, ebLess, ebGreater, ebLessEqual, ebGreaterEqual);
```

TkbmSQLConstNode File: kbmSQLElements

Delphi declaration
```
TkbmSQLConstNode=Class(TkbmSQLCustomValueNode);
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOperation: TkbmSQLCustomOperation; AValue:
variant); ;
```

Protected Methods/Functions

GetDataType

Delphi declaration
```
GetDataType: TFieldType; ;
```

GetValue

Delphi declaration
```
GetValue: variant; ;
```

GetWidth

Delphi declaration
```
GetWidth: integer; ;
```

SetValue

Delphi declaration
```
SetValue(AValue: variant); ;
```

TkbmSQLCustomDatasetAPI File: kbmSQLDBAPI

Delphi declaration

```
TkbmSQLCustomDatasetAPI=Class(TkbmSQLCustomDBAPI);
```

Public Methods/Functions

Delete

Delphi declaration

```
Delete(ADataset: TObject; AOperation:
TkbmSQLDeleteOperation): boolean; ;
```

Insert

Delphi declaration

```
Insert(ADataset: TObject; AOperation:
TkbmSQLInsertOperation): boolean; ;
```

Select

Delphi declaration

```
Select(ADataset: TObject; AResultDataset:
TkbmCustomMemTable; AOperation:
TkbmSQLCustomSelectOperation): boolean; ;
```

Update

Delphi declaration

```
Update(ADataset: TObject; AOperation:
TkbmSQLUpdateOperation): boolean; ;
```

TkbmSQLCustomDBAPI File: kbmSQLDBAPI

Delphi declaration

```
TkbmSQLCustomDBAPI=Class(TkbmSQLCustomDBAPI_);
```

Public Methods/Functions

Delete

Delphi declaration

```
Delete(ADataset: TObject; AOperation:
TkbmSQLDeleteOperation): boolean; ; ;
```

Insert

Delphi declaration

```
Insert(ADataset: TObject; AOperation:
TkbmSQLInsertOperation): boolean; ; ;
```

LocateFirst

Delphi declaration

```
LocateFirst(ATable: TkbmSQLTable; ACondition:
TkbmSQLCustomNode): boolean; ;
```

LocateNext

Delphi declaration

```
LocateNext(ATable: TkbmSQLTable; ACondition:
TkbmSQLCustomNode): boolean; ;
```

Select

Delphi declaration

```
Select(ADataset: TObject; AResultDataset:
TkbmCustomMemTable; AOperation:
TkbmSQLCustomSelectOperation): boolean; ; ;
```

Update

Delphi declaration

```
Update(ADataset: TObject; AOperation:
TkbmSQLUpdateOperation): boolean; ; ;
```

Protected Methods/Functions

AppendResultRecord

Delphi declaration

```
AppendResultRecord(AResultTable: TkbmCustomMemTable;
ASelection: TkbmSQLNodes; AIncludeGroupFields:
boolean); ; ;
AppendResultRecord(AResultTable: TkbmCustomMemTable;
ASelection: TkbmSQLNodes; AIncludeGroupFields:
boolean); ; ;
```

ApplyGroupBy

Delphi declaration

```
ApplyGroupBy(ASourceTable, AResultTable:
TkbmCustomMemTable; ASelection: TkbmSQLNodes; AGroup:
TkbmSQLNodes; AAggregates: TkbmSQLNodes); ;
```

ApplyOffsetAndLimit

Delphi declaration

```
ApplyOffsetAndLimit(AResultTable: TkbmCustomMemTable;
AOffset: TkbmSQLCustomNode; ALimit:
TkbmSQLCustomNode); ;
```

ApplyOrderBy

Delphi declaration

```
ApplyOrderBy(AResultTable: TkbmCustomMemTable; AOrder:
TkbmSQLNodes); ;
```

CheckNonAggregateFieldsInGroupClause

Delphi declaration

```
CheckNonAggregateFieldsInGroupClause(ASelection:
TkbmSQLNodes; AGroup: TkbmSQLNodes); ;
```

CreateDistinctIndex

Delphi declaration

```
CreateDistinctIndex(ATable: TkbmCustomMemTable;
ADistinct: TkbmSQLNodes); ;
```

CreateField

Delphi declaration

```
CreateField(AOwner: TComponent; ADataType:
TFieldType): TField; ;
```

CreateSelectionFields

Delphi declaration

```
CreateSelectionFields(ATable: TkbmCustomMemTable;
ASelection: TkbmSQLNodes; ACreateGroupFields: boolean;
AUseAggregateSources: boolean); ;
```

DetermineUsableIndexesForRefFields

Delphi declaration

```
DetermineUsableIndexesForRefFields(AOperation:
TkbmSQLCustomOperation); ;
```

FixupTemporaryResultFields

Delphi declaration

```
FixupTemporaryResultFields(ATempResultTable:
TkbmCustomMemTable; AOperation:
TkbmSQLCustomSelectOperation; AFields:
TkbmSQLFieldNodes); ;
```

GetIndexNameForField

Delphi declaration

```
GetIndexNameForField(AField: TkbmSQLFieldNode): ; ; ;
```

Search

Delphi declaration

```
Search(ASourceTable, AResultTable: TkbmCustomMemTable;
AOperation: TkbmSQLCustomSelectOperation; ASelection:
TkbmSQLNodes; ACondition: TkbmSQLCustomNode;
ACopyDirectlyFromSource: boolean); ;
```

SetupSourceField

Delphi declaration

```
SetupSourceField(ANode: TkbmSQLCustomNode; ADataset:
TDataset); ;
```

SetupSourceFields

Delphi declaration

```
SetupSourceFields(ANodes: TkbmSQLNodes; ADataset:
TDataset); ;
```

TkbmSQLCustomDBAPI_ File: kbmSQLElements

Delphi declaration

```
TkbmSQLCustomDBAPI_=Class
```

TkbmSQLCustomFunction File: kbmSQLFuncAPI

Delphi declaration

```
TkbmSQLCustomFunction=(ASender: TObject; AOperation:
TkbmSQLFunctionOperation; AArgs: TkbmSQLNodes; AResult: variant):
boolean;
```

TkbmSQLCustomNode File: kbmSQLElements

Delphi declaration
```
TkbmSQLCustomNode=Class
```

Public Properties/Variables

Alias

Delphi declaration
```
Alias: FAlias FAlias;
```

DataType

Delphi declaration
```
DataType: TFieldType GetDataType FDataType;
```

Description

Delphi declaration
```
Description: FDescription FDescription;
```

DestinationField

Delphi declaration
```
DestinationField: TField FDestinationField
FDestinationField;
```

ID

Delphi declaration
```
ID: integer FID FID;
```

Nodes

Delphi declaration
```
Nodes: TkbmSQLNodes FNodes FNodes;
```

NonScalar

Delphi declaration
```
NonScalar: boolean GetNonScalar;
```

Operation

Delphi declaration
```
Operation: TkbmSQLCustomOperation FOperation;
```

Owner

Delphi declaration
```
Owner: TkbmSQLNodes FOwner FOwner;
```

Parent

Delphi declaration
```
Parent: TObject FParent FParent;
```

SourceField

Delphi declaration
```
SourceField: TField FSourceField FSourceField;
```

UniqueName

Delphi declaration
```
UniqueName: FUniqueName FUniqueName;
```

Width

Delphi declaration
```
Width: integer GetWidth;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOperation: TkbmSQLCustomOperation); ; ;
Create(AOperation: TkbmSQLCustomOperation); ; ;
```

Execute

Delphi declaration

```
Execute: variant; ; ;
```

GetName

Delphi declaration

```
GetName: ; ;
```

GetUniqueName

Delphi declaration

```
GetUniqueName: ; ;
```

Protected Methods/Functions

GetDataType

Delphi declaration

```
GetDataType: TFieldType; ;
```

GetExpectedWidth

Delphi declaration

```
GetExpectedWidth: integer; ;
```

GetNonScalar

Delphi declaration

```
GetNonScalar: boolean; ;
```

GetWidth

Delphi declaration

```
GetWidth: integer; ;
```

TkbmSQLCustomOperation File: kbmSQLElements

Delphi declaration
```
TkbmSQLCustomOperation=Class
```

Public Properties/Variables

Context

Delphi declaration
```
Context: TObject FContext FContext;
```

Flags

Delphi declaration
```
Flags: TkbmSQLOperationFlags FFlags FFlags;
```

FormatSettings

Delphi declaration
```
FormatSettings: TFormatSettings;
```

OpType

Delphi declaration
```
OpType: TkbmSQLOperationType FType;
```

Owner

Delphi declaration
```
Owner: TObject FOwner;
```

ParseState

Delphi declaration
```
ParseState: TkbmSQLOperationParseState FParseState
FParseState;
```

RefFields

Delphi declaration
```
RefFields: TkbmSQLRefFields FRefFields;
```

RefHavingFields

Delphi declaration

```
RefHavingFields: TkbmSQLFieldNodes FRefHavingFields;
```

RefSearchFields

Delphi declaration

```
RefSearchFields: TkbmSQLFieldNodes FRefSearchFields;
```

RefSourceFields

Delphi declaration

```
RefSourceFields: TkbmSQLFieldNodes FRefSourceFields;
```

RefSourceTables

Delphi declaration

```
RefSourceTables: TkbmSQLTables FRefSourceTables;
```

RefTables

Delphi declaration

```
RefTables: TkbmSQLRefTables FRefTables;
```

RefWildcards

Delphi declaration

```
RefWildcards: TkbmSQLFieldNodes FRefWildcards;
```

SearchCondition

Delphi declaration

```
SearchCondition: boolean FSearchCondition
FSearchCondition;
```

Public Methods/Functions

Create

Delphi declaration

```
Create(AOwner: TObject); ;
```

Destroy

Delphi declaration

```
Destroy; ;
```

Execute

Delphi declaration

```
Execute; ;
```

FixupRefSourceFields

Delphi declaration

```
FixupRefSourceFields; ;
```

GetErrorColOffset

Delphi declaration

```
GetErrorColOffset: integer; ;
```

GetNextID

Delphi declaration

```
GetNextID: integer;
```

Optimize

Delphi declaration

```
Optimize; ;
```

OptimizeNodeTree

Delphi declaration

```
OptimizeNodeTree(ANode: TkbmSQLCustomNode); ;
```

Prepare

Delphi declaration

```
Prepare; ;
```

Protected Methods/Functions

GetFieldByAlias

Delphi declaration

```
GetFieldByAlias(AAlias: ): TkbmSQLCustomNode; ;
```

ReplaceParentRef

Delphi declaration

```
ReplaceParentRef(AParent: TObject; AOldNode, ANewNode:
TkbmSQLCustomNode);
```

Public Events

OnGetFunction

Delphi declaration

```
OnGetFunction: TkbmSQLOnGetFunction FOnGetFunction
FOnGetFunction;
```

OnGetVariableMetaData

Delphi declaration

```
OnGetVariableMetaData: TkbmSQLOnGetVariableMetaData
FOnGetVariableMetaData FOnGetVariableMetaData;
```

OnGetVariableValue

Delphi declaration

```
OnGetVariableValue: TkbmSQLOnGetVariableValue
FOnGetVariableValue FOnGetVariableValue;
```

TkbmSQLCustomSelectOperation File: kbmSQLElements

Delphi declaration
```
TkbmSQLCustomSelectOperation=Class(TkbmSQLCustomOperation);
```

Public Properties/Variables

Aggregates

Delphi declaration
```
Aggregates: TkbmSQLNodes FAggregates;
```

Condition

Delphi declaration
```
Condition: TkbmSQLCustomNode FCondition FCondition;
```

Distinct

Delphi declaration
```
Distinct: boolean FDistinct FDistinct;
```

Group

Delphi declaration
```
Group: TkbmSQLNodes FGroup;
```

HavingCondition

Delphi declaration
```
HavingCondition: TkbmSQLCustomNode FHavingCondition
FHavingCondition;
```

IsAggregate

Delphi declaration
```
IsAggregate: boolean GetIsAggregate;
```

IsDistinctAggregate

Delphi declaration
```
IsDistinctAggregate: boolean GetIsDistinctAggregate;
```

Limit

Delphi declaration
```
Limit: TkbmSQLCustomNode FLimit FLimit;
```

Offset

Delphi declaration
```
Offset: TkbmSQLCustomNode FOffset FOffset;
```

Order

Delphi declaration
```
Order: TkbmSQLNodes FOrder;
```

ResultTable

Delphi declaration
```
ResultTable: TkbmCustomMemTable FResultTable
FResultTable;
```

Selection

Delphi declaration
```
Selection: TkbmSQLNodes FSelection;
```

Tables

Delphi declaration
```
Tables: TkbmSQLTables FTables;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOwner: TObject); ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Execute

Delphi declaration

```
Execute; ;
```

FixupRefSourceFields

Delphi declaration

```
FixupRefSourceFields; ;
```

FixupSelectionWildcardNodes

Delphi declaration

```
FixupSelectionWildcardNodes; ;
```

FixupWildCardSelectionNodes

Delphi declaration

```
FixupWildCardSelectionNodes; ;
```

Optimize

Delphi declaration

```
Optimize; ;
```

Prepare

Delphi declaration

```
Prepare; ;
```

Protected Methods/Functions

GetFieldByAlias

Delphi declaration

```
GetFieldByAlias(AAlias: ): TkbmSQLCustomNode; ;
```

GetIsAggregate

Delphi declaration

```
GetIsAggregate: boolean;
```

GetIsDistinctAggregate

Delphi declaration

```
GetIsDistinctAggregate: boolean;
```

TkbmSQLCustomValueNode File: kbmSQLElements

Delphi declaration
```
TkbmSQLCustomValueNode=Class(TkbmSQLCustomNode);
```

Public Properties/Variables

Value

Delphi declaration
```
Value: variant GetValue SetValue;
```

Public Methods/Functions

Execute

Delphi declaration
```
Execute: variant; ;
```

Protected Methods/Functions

GetValue

Delphi declaration
```
GetValue: variant; ; ;
```

SetValue

Delphi declaration
```
SetValue(AValue: variant); ;
```

TkbmSQLDBAPIRegistration File: kbmSQLDBAPI

Delphi declaration
```
TkbmSQLDBAPIRegistration=Class
```

Public Properties/Variables

Dataset

Delphi declaration
```
Dataset: TClass FDataset;
```

DatasetAPI

Delphi declaration
```
DatasetAPI: TkbmSQLCustomDBAPI FDatasetAPI;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(ADataset: TClass; ADatasetAPIClass:
TkbmSQLCustomDBAPIClass); ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

TkbmSQLDBAPIRegistrations File: kbmSQLDBAPI

Delphi declaration
```
TkbmSQLDBAPIRegistrations=Class
```

Public Methods/Functions

Clear

Delphi declaration
```
Clear;
```

Create

Delphi declaration
```
Create; ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

GetAPI

Delphi declaration
```
GetAPI(ADataset: TClass): TkbmSQLCustomDBAPI;
```

RegisterAPI

Delphi declaration
```
RegisterAPI(ADatasetClass: TClass; ADatasetAPIClass:
TkbmSQLCustomDBAPIClass);
```

TkbmSQLDeleteOperation File: kbmSQLElements

Delphi declaration
```
TkbmSQLDeleteOperation=Class(TkbmSQLCustomOperation);
```

Public Properties/Variables

Condition

Delphi declaration
```
Condition: TkbmSQLCustomNode FCondition FCondition;
```

Table

Delphi declaration
```
Table: TkbmSQLTable FTable FTable;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOwner: TObject); ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Execute

Delphi declaration
```
Execute; ;
```

Optimize

Delphi declaration
```
Optimize; ;
```

Prepare

Delphi declaration
```
Prepare; ;
```

TkbmSQLEvaluationOperation File: kbmSQLElements

Delphi declaration
```
TkbmSQLEvaluationOperation=Class(TkbmSQLCustomOperation);
```

Public Properties/Variables

Evaluation

Delphi declaration
```
Evaluation: TkbmSQLCustomNode FEvaluation FEvaluation;
```

Fields

Delphi declaration
```
Fields: TkbmSQLFieldNodes FFields;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOwner: TObject); ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Evaluate

Delphi declaration
```
Evaluate: variant; ;
```

Execute

Delphi declaration
```
Execute; ;
```

GetErrorColOffset

Delphi declaration
```
GetErrorColOffset: integer; ;
```

Optimize

Delphi declaration

```
Optimize; ;
```

Prepare

Delphi declaration

```
Prepare; ;
```

TkbmSQLField File: kbmSQLElements

Delphi declaration
```
TkbmSQLField=Class
```

Public Properties/Variables

Alias

Delphi declaration
```
Alias: FAlias SetAlias;
```

Descending

Delphi declaration
```
Descending: boolean FDescending FDescending;
```

Description

Delphi declaration
```
Description: FDescription FDescription;
```

ID

Delphi declaration
```
ID: FID;
```

IndexName

Delphi declaration
```
IndexName: FIndexName FIndexName;
```

Name

Delphi declaration
```
Name: FName SetName;
```

NonScalar

Delphi declaration
```
NonScalar: boolean FNonScalar FNonScalar;
```

OriginatingField

Delphi declaration

```
OriginatingField: TField FOriginatingField
FOriginatingField;
```

Table

Delphi declaration

```
Table: TkbmSQLTable FTable FTable;
```

Tablename

Delphi declaration

```
Tablename: FTablename FTablename;
```

Public Methods/Functions

Create

Delphi declaration

```
Create; ;
```

Protected Methods/Functions

SetAlias

Delphi declaration

```
SetAlias(AAlias: ); ;
```

SetName

Delphi declaration

```
SetName(AName: ); ;
```

TkbmSQLFieldNode File: kbmSQLElements

Delphi declaration
```
TkbmSQLFieldNode=Class(TkbmSQLCustomNode);
```

Public Properties/Variables

Descending

Delphi declaration
```
Descending: boolean FDescending FDescending;
```

Field

Delphi declaration
```
Field: TkbmSQLField FField FField;
```

FieldName

Delphi declaration
```
FieldName: FFieldName FFieldname;
```

FieldType

Delphi declaration
```
FieldType: TkbmSQLFieldNodeType FFieldType FFieldType;
```

GroupBy

Delphi declaration
```
GroupBy: boolean FGroupBy FGroupBy;
```

IndexName

Delphi declaration
```
IndexName: FIndexName FIndexName;
```

OrderBy

Delphi declaration
```
OrderBy: boolean FOrderBy FOrderBy;
```

OriginatingField

Delphi declaration
```
OriginatingField: TField FOriginatingField
FOriginatingField;
```

Table

Delphi declaration
```
Table: TkbmSQLTable FTable FTable;
```

TableName

Delphi declaration
```
TableName: FTableName FTableName;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOperation: TkbmSQLCustomOperation); ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Execute

Delphi declaration
```
Execute: variant; ;
```

GetName

Delphi declaration
```
GetName: ; ;
```

Protected Methods/Functions

GetDataType

Delphi declaration
```
GetDataType: TFieldType; ;
```

GetNonScalar

Delphi declaration

```
GetNonScalar: boolean; ;
```

GetWidth

Delphi declaration

```
GetWidth: integer; ;
```

TkbmSQLFieldNodes File: kbmSQLElements

Delphi declaration

```
TkbmSQLFieldNodes=Class(TkbmSQLNodes);
```

Public Properties/Variables

FieldNodes

Delphi declaration

```
FieldNodes[AIndex: integer]: TkbmSQLFieldNode
GetFieldNode SetFieldNode;
```

Protected Methods/Functions

GetFieldNode

Delphi declaration

```
GetFieldNode(AIndex: integer): TkbmSQLFieldNode; ;
```

SetFieldNode

Delphi declaration

```
SetFieldNode(AIndex: integer; ANode:
TkbmSQLFieldNode); ;
```

TkbmSQLFields File: kbmSQLElements

Delphi declaration
```
TkbmSQLFields=Class
```

Public Properties/Variables

Count

Delphi declaration
```
Count: integer GetCount;
```

FieldByID

Delphi declaration
```
FieldByID[AID: ]: TkbmSQLField GetFieldByID;
```

Fields

Delphi declaration
```
Fields[AIndex: integer]: TkbmSQLField GetField;
```

Public Methods/Functions

Add

Delphi declaration
```
Add(AField: TkbmSQLField);
```

Clear

Delphi declaration
```
Clear;
```

Create

Delphi declaration
```
Create; ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

GetCount

Delphi declaration

```
GetCount: integer; ;
```

GetField

Delphi declaration

```
GetField(AIndex: integer): TkbmSQLField; ;
```

GetFieldByID

Delphi declaration

```
GetFieldByID(AID: ): TkbmSQLField; ;
```

TkbmSQLFunctionNode File: kbmSQLElements

Delphi declaration
```
TkbmSQLFunctionNode=Class(TkbmSQLCustomNode);
```

Public Properties/Variables

Args

Delphi declaration
```
Args: TkbmSQLNodes FArgs;
```

FunctionName

Delphi declaration
```
FunctionName: FFunctionName;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOperation: TkbmSQLCustomOperation;
AFunctionName: ; AArgs: TkbmSQLNodes); ; ;
Create(AOperation: TkbmSQLCustomOperation;
AFunctionName: ; AArgs: TkbmSQLNodes); ; ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Execute

Delphi declaration
```
Execute: variant; ;
```

IsFunctionValid

Delphi declaration
```
IsFunctionValid: boolean; ;
```

GetDataType

Delphi declaration

```
GetDataType: TFieldType; ;
```

GetWidth

Delphi declaration

```
GetWidth: integer; ;
```

TkbmSQLFunctionOperation File: kbmSQLFuncAPI

Value	Meaning
foWidth:	
foExecute:	
foDataType:	

Delphi declaration

```
TkbmSQLFunctionOperation =(foWidth, foExecute, foDataType);
```

TkbmSQLFunctionRegistration File: kbmSQLFuncAPI

Delphi declaration
```
TkbmSQLFunctionRegistration=Class
```

Public Properties/Variables

_Function

Delphi declaration
```
_Function: TkbmSQLCustomFunction FFunction;
```

Enabled

Delphi declaration
```
Enabled: boolean FEnabled FEnabled;
```

FunctionName

Delphi declaration
```
FunctionName: FFunctionName;
```

Group

Delphi declaration
```
Group: FGroup;
```

ReturnDataType

Delphi declaration
```
ReturnDataType: TFieldType FReturnDataType;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AGroup: ; AFunctionName: ; AFunction:
TkbmSQLCustomFunction; AEnabled: boolean=true); ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

TkbmSQLFunctionRegistrations File: kbmSQLFuncAPI

Delphi declaration
```
TkbmSQLFunctionRegistrations=Class
```

Public Methods/Functions

Clear

Delphi declaration
```
Clear;
```

Create

Delphi declaration
```
Create; ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

DisableFunction

Delphi declaration
```
DisableFunction(AFunctionName: );
```

DisableGroup

Delphi declaration
```
DisableGroup(AGroup: );
```

EnableFunction

Delphi declaration
```
EnableFunction(AFunctionName: );
```

EnableGroup

Delphi declaration
```
EnableGroup(AGroup: );
```

GetFunctionReg

Delphi declaration

```
GetFunctionReg(AFunctionName: ):
TkbmSQLFunctionRegistration;
```

RegisterFunction

Delphi declaration

```
RegisterFunction(AGroup: ; AFunctionName: ; AFunction:
TkbmSQLCustomFunction);
```

TkbmSQLGroupFieldNode File: kbmSQLElements

Delphi declaration

```
TkbmSQLGroupFieldNode=Class(TkbmSQLFieldNode);;
```

TkbmSQLInNode File: kbmSQLElements

Delphi declaration
```
TkbmSQLInNode=Class(TkbmSQLCustomNode);
```

Public Properties/Variables

LeftNode

Delphi declaration
```
LeftNode: TkbmSQLCustomNode FLeftNode FLeftNode;
```

Nodes

Delphi declaration
```
Nodes: TkbmSQLNodes FNodes;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOperation: TkbmSQLCustomOperation); ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Execute

Delphi declaration
```
Execute: variant; ;
```

Protected Methods/Functions

GetDataType

Delphi declaration
```
GetDataType: TFieldType; ;
```

TkbmSQLInsertField File: kbmSQLElements

Delphi declaration
```
TkbmSQLInsertField=Class
```

Public Properties/Variables

ID

Delphi declaration
```
ID: FName;
```

Name

Delphi declaration
```
Name: FName SetName;
```

Protected Methods/Functions

SetName

Delphi declaration
```
SetName(AName: ); ;
```

TkbmSQLInsertFields File: kbmSQLElements

Delphi declaration
```
TkbmSQLInsertFields=Class
```

Public Properties/Variables

Count

Delphi declaration
```
Count: integer GetCount;
```

InsertFieldByID

Delphi declaration
```
InsertFieldByID[AID: ]: TkbmSQLInsertField
GetInsertFieldByID;
```

InsertFields

Delphi declaration
```
InsertFields[AIndex: integer]: TkbmSQLInsertField
GetInsertField;
```

Public Methods/Functions

Add

Delphi declaration
```
Add(AField: TkbmSQLInsertField);
```

Clear

Delphi declaration
```
Clear;
```

Create

Delphi declaration
```
Create; ;
```

Destroy

Delphi declaration

```
Destroy; ;
```

Protected Methods/Functions

GetCount

Delphi declaration

```
GetCount: integer; ;
```

GetInsertField

Delphi declaration

```
GetInsertField(AIndex: integer): TkbmSQLInsertField; ;
```

GetInsertFieldByID

Delphi declaration

```
GetInsertFieldByID(AID: ): TkbmSQLInsertField; ;
```

TkbmSQLInsertOperation File: kbmSQLElements

Delphi declaration
```
TkbmSQLInsertOperation=Class(TkbmSQLCustomOperation);
```

Public Properties/Variables

Fields

Delphi declaration
```
Fields: TkbmSQLFieldNodes FFields;
```

Table

Delphi declaration
```
Table: TkbmSQLTable FTable FTable;
```

Values

Delphi declaration
```
Values: TkbmSQLNodes FValues;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOwner: TObject); ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Execute

Delphi declaration
```
Execute; ;
```

Optimize

Delphi declaration
```
Optimize; ;
```

Prepare

Delphi declaration

```
Prepare; ;
```

TkbmSQLMemTableAPI File: kbmSQLMemTableAPI

Delphi declaration

```
TkbmSQLMemTableAPI=Class(TkbmSQLCustomDatasetAPI);
```

Public Methods/Functions

LocateFirst

Delphi declaration

```
LocateFirst(ATable: TkbmSQLTable; ACondition:
TkbmSQLCustomNode): boolean; ;
```

LocateNext

Delphi declaration

```
LocateNext(ATable: TkbmSQLTable; ACondition:
TkbmSQLCustomNode): boolean; ;
```

Protected Methods/Functions

GetIndexNameForField

Delphi declaration

```
GetIndexNameForField(AField: TkbmSQLFieldNode): ; ;
```

TkbmSQLNodes File: kbmSQLElements

Delphi declaration
```
TkbmSQLNodes=Class
```

Public Properties/Variables

Node

Delphi declaration
```
Node[AIndex: integer]: TkbmSQLCustomNode GetNode
SetNode;
```

Parent

Delphi declaration
```
Parent: TkbmSQLCustomNode FParent FParent;
```

Public Methods/Functions

Add

Delphi declaration
```
Add(ANodes: TkbmSQLNodes; AIgnoreIfExists:
boolean=false); ;
Add(ANode: TkbmSQLCustomNode; AIgnoreIfExists:
boolean=false); ;

Add(ANodes: TkbmSQLNodes; AIgnoreIfExists:
boolean=false); ;
Add(ANode: TkbmSQLCustomNode; AIgnoreIfExists:
boolean=false); ;
```

Clear

Delphi declaration
```
Clear;
```

Count

Delphi declaration
```
Count: integer;
```

Create

Delphi declaration

```
Create(AOwnsNodes: boolean); ;
```

Delete

Delphi declaration

```
Delete(ANode: TkbmSQLCustomNode); ;
Delete(APos: integer); ;

Delete(ANode: TkbmSQLCustomNode); ;
Delete(APos: integer); ;
```

Destroy

Delphi declaration

```
Destroy; ;
```

GetByAlias

Delphi declaration

```
GetByAlias(AAlias: ): TkbmSQLCustomNode;
```

GetByFieldName

Delphi declaration

```
GetByFieldName(AName: ): TkbmSQLFieldNode;
```

GetByName

Delphi declaration

```
GetByName(AName: ): TkbmSQLCustomNode;
```

GetByUniqueName

Delphi declaration

```
GetByUniqueName(AUniqueName: ): TkbmSQLCustomNode;
```

IndexOf

Delphi declaration

```
IndexOf(ANode: TkbmSQLCustomNode): integer;
```

Insert

Delphi declaration

```
Insert(APos: integer; ANode: TkbmSQLCustomNode);
```

Replace

Delphi declaration

```
Replace(AOldNode, ANewNode: TkbmSQLCustomNode;
AFreeOldNode: boolean=false);
```

Protected Methods/Functions

GetNode

Delphi declaration

```
GetNode(AIndex: integer): TkbmSQLCustomNode;
```

SetNode

Delphi declaration

```
SetNode(AIndex: integer; ANode: TkbmSQLCustomNode);
```

TkbmSQLNodeStack File: kbmSQLElements

Delphi declaration
```
TkbmSQLNodeStack=Class
```

Public Properties/Variables

Count

Delphi declaration
```
Count: integer GetCount;
```

Stack

Delphi declaration
```
Stack[AIndex: integer]: TkbmSQLCustomNode GetNode;
```

Public Methods/Functions

Clear

Delphi declaration
```
Clear;
```

Create

Delphi declaration
```
Create; ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Pop

Delphi declaration
```
Pop: TkbmSQLCustomNode;
```

Push

Delphi declaration
```
Push(AExpr: TkbmSQLCustomNode);
```

GetCount

Delphi declaration

```
GetCount: integer; ;
```

GetNode

Delphi declaration

```
GetNode(AIndex: integer): TkbmSQLCustomNode; ;
```

TkbmSQLOperationType File: kbmSQLElements

Value	Meaning
sotSELECT:	
sotUPDATE:	
sotINSERT:	
sotDELETE:	

Delphi declaration

```
TkbmSQLOperationType =(sotSELECT, sotUPDATE, sotINSERT, sotDELETE);
```

TkbmSQLRefFields File: kbmSQLElements

Delphi declaration
```
TkbmSQLRefFields=Class
```

Public Properties/Variables

Count

Delphi declaration
```
Count: integer GetCount;
```

FieldByID

Delphi declaration
```
FieldByID[AID: ]: TkbmSQLField GetFieldByID;
```

Fields

Delphi declaration
```
Fields[AIndex: integer]: TkbmSQLField GetField;
```

Public Methods/Functions

Add

Delphi declaration
```
Add(AField: TkbmSQLField);
```

Clear

Delphi declaration
```
Clear;
```

Create

Delphi declaration
```
Create; ;
```

Delete

Delphi declaration
```
Delete(APos: integer);
```

Destroy

Delphi declaration

```
Destroy; ;
```

GetIndexOfField

Delphi declaration

```
GetIndexOfField(AField: TkbmSQLField): integer; ;
```

Insert

Delphi declaration

```
Insert(APos: integer; AField: TkbmSQLField);
```

Protected Methods/Functions

GetCount

Delphi declaration

```
GetCount: integer; ;
```

GetField

Delphi declaration

```
GetField(AIndex: integer): TkbmSQLField; ;
```

GetFieldByID

Delphi declaration

```
GetFieldByID(AID: ): TkbmSQLField; ;
```

TkbmSQLRefTables File: kbmSQLElements

Delphi declaration
```
TkbmSQLRefTables=Class
```

Public Properties/Variables

Count

Delphi declaration
```
Count: integer GetCount;
```

TableByID

Delphi declaration
```
TableByID[AID: ]: TkbmSQLTable GetTableByID;
```

Tables

Delphi declaration
```
Tables[AIndex: integer]: TkbmSQLTable GetTable;
```

Public Methods/Functions

Add

Delphi declaration
```
Add(ATable: TkbmSQLTable);
```

Clear

Delphi declaration
```
Clear;
```

Create

Delphi declaration
```
Create; ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Protected Methods/Functions

GetCount

Delphi declaration

```
GetCount: integer; ;
```

GetTable

Delphi declaration

```
GetTable(AIndex: integer): TkbmSQLTable; ;
```

GetTableByID

Delphi declaration

```
GetTableByID(AID: ): TkbmSQLTable;
```

TkbmSQLSelectOperation File: kbmSQLElements

Delphi declaration

```
TkbmSQLSelectOperation=Class(TkbmSQLCustomSelectOperation);;
```

TkbmSQLTable File: kbmSQLElements

Delphi declaration
```
TkbmSQLTable=Class
```

Public Properties/Variables

AffectedRows

Delphi declaration
```
AffectedRows: integer FAffectedRows FAffectedRows;
```

Alias

Delphi declaration
```
Alias: FAlias SetAlias;
```

AliasOrName

Delphi declaration
```
AliasOrName: GetAliasOrName;
```

API

Delphi declaration
```
API: TkbmSQLCustomDBAPI_ FAPI FAPI;
```

BaseTable

Delphi declaration
```
BaseTable: TkbmSQLTable FBaseTable FBaseTable;
```

Dataset

Delphi declaration
```
Dataset: TkbmCustomMemTable FDataset FDataset;
```

ID

Delphi declaration
```
ID: FID;
```

Name

Delphi declaration

```
Name: FName SetName;
```

Public Methods/Functions

Create

Delphi declaration

```
Create; ;
```

GetCurrentRecNo

Delphi declaration

```
GetCurrentRecNo: integer;
```

GetCurrentRowID

Delphi declaration

```
GetCurrentRowID: int64;
```

Protected Methods/Functions

GetAliasOrName

Delphi declaration

```
GetAliasOrName: ;
```

SetAlias

Delphi declaration

```
SetAlias(AAlias: ); ;
```

SetName

Delphi declaration

```
SetName(AName: ); ;
```

TkbmSQLTables File: kbmSQLElements

Delphi declaration
```
TkbmSQLTables=Class
```

Public Properties/Variables

Count

Delphi declaration
```
Count: integer GetCount;
```

OwnsTables

Delphi declaration
```
OwnsTables: boolean FOwnsTables;
```

TableByAlias

Delphi declaration
```
TableByAlias[AAlias: ]: TkbmSQLTable GetTableByAlias;
```

TableByID

Delphi declaration
```
TableByID[AID: ]: TkbmSQLTable GetTableByID;
```

TableByName

Delphi declaration
```
TableByName[AName: ]: TkbmSQLTable GetTableByName;
```

Tables

Delphi declaration
```
Tables[AIndex: integer]: TkbmSQLTable GetTable;
```

Public Methods/Functions

Add

Delphi declaration

```
Add(AName: ; ADataset: TkbmCustomMemTable); ;
Add(ATable: TkbmSQLTable); ;

Add(AName: ; ADataset: TkbmCustomMemTable); ;
Add(ATable: TkbmSQLTable); ;
```

AddUnique

Delphi declaration

```
AddUnique(ATable: TkbmSQLTable);
```

Clear

Delphi declaration

```
Clear;
```

Create

Delphi declaration

```
Create(AOwnTables: boolean); ;
```

Delete

Delphi declaration

```
Delete(AName: ); ;
Delete(ATable: TkbmSQLTable); ;

Delete(AName: ); ;
Delete(ATable: TkbmSQLTable); ;
```

Destroy

Delphi declaration

```
Destroy; ;
```

GetTablesContainingFieldName

Delphi declaration

```
GetTablesContainingFieldName(AFieldName: ):
TkbmSQLTables;
```

Protected Methods/Functions

GetCount

Delphi declaration

```
GetCount: integer; ;
```

GetTable

Delphi declaration

```
GetTable(AIndex: integer): TkbmSQLTable; ;
```

GetTableByAlias

Delphi declaration

```
GetTableByAlias(AAlias: ): TkbmSQLTable;
```

GetTableByID

Delphi declaration

```
GetTableByID(AID: ): TkbmSQLTable;
```

GetTableByName

Delphi declaration

```
GetTableByName(AName: ): TkbmSQLTable;
```

TkbmSQLUnaryNode File: kbmSQLElements

Delphi declaration
```
TkbmSQLUnaryNode=Class(TkbmSQLCustomNode);
```

Public Properties/Variables

Operator

Delphi declaration
```
Operator: TkbmSQLUnaryNodeOperator FOperator;
```

RightNode

Delphi declaration
```
RightNode: TkbmSQLCustomNode FRightNode FRightNode;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOperation: TkbmSQLCustomOperation; AOperator:
TkbmSQLUnaryNodeOperator); ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Execute

Delphi declaration
```
Execute: variant; ;
```

Protected Methods/Functions

GetDataType

Delphi declaration
```
GetDataType: TFieldType; ;
```

GetWidth

Delphi declaration

```
GetWidth: integer; ;
```

TkbmSQLUnaryNodeOperator File: kbmSQLElements

Value	Meaning
euNot:	
euNegate:	

Delphi declaration

```
TkbmSQLUnaryNodeOperator =(euNot, euNegate);
```

TkbmSQLUpdateField File: kbmSQLElements

Delphi declaration
```
TkbmSQLUpdateField=Class
```

Public Properties/Variables

ID

Delphi declaration
```
ID: FName;
```

Name

Delphi declaration
```
Name: FName FName;
```

TkbmSQLUpdateOperation File: kbmSQLElements

Delphi declaration
```
TkbmSQLUpdateOperation=Class(TkbmSQLCustomOperation);
```

Public Properties/Variables

Condition

Delphi declaration
```
Condition: TkbmSQLCustomNode FCondition FCondition;
```

Fields

Delphi declaration
```
Fields: TkbmSQLFieldNodes FFields;
```

Table

Delphi declaration
```
Table: TkbmSQLTable FTable FTable;
```

Values

Delphi declaration
```
Values: TkbmSQLNodes FValues;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOwner: TObject); ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Execute

Delphi declaration
```
Execute; ;
```

Optimize

Delphi declaration

```
Optimize; ;
```

Prepare

Delphi declaration

```
Prepare; ;
```

TkbmSQLValueNode File: kbmSQLElements

Delphi declaration
```
TkbmSQLValueNode=Class(TkbmSQLCustomValueNode);
```

Public Properties/Variables

ValueType

Delphi declaration
```
ValueType: TkbmSQLValueType FValueType;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOperation: TkbmSQLCustomOperation; AValueType:
TkbmSQLValueType); ;
```

Protected Methods/Functions

GetDataType

Delphi declaration
```
GetDataType: TFieldType; ;
```

GetValue

Delphi declaration
```
GetValue: variant; ;
```

GetWidth

Delphi declaration
```
GetWidth: integer; ;
```

TkbmSQLValueType File: kbmSQLElements

Value	Meaning
evtTimeStamp:	
evtNull:	

Delphi declaration

```
TkbmSQLValueType =(evtTimeStamp, evtNull);
```

TkbmSQLVariableNode File: kbmSQLElements

Delphi declaration
```
TkbmSQLVariableNode=Class(TkbmSQLCustomNode);
```

Public Properties/Variables

VariableName

Delphi declaration
```
VariableName: FVariableName FVariableName;
```

Public Methods/Functions

Create

Delphi declaration
```
Create(AOperation: TkbmSQLCustomOperation); ;
```

Destroy

Delphi declaration
```
Destroy; ;
```

Execute

Delphi declaration
```
Execute: variant; ;
```

IsVariableValid

Delphi declaration
```
IsVariableValid: boolean; ;
```

Protected Methods/Functions

GetDataType

Delphi declaration
```
GetDataType: TFieldType; ;
```

GetWidth

Delphi declaration
```
GetWidth: integer; ;
```

TkbmState File: kbmMemTable

Value	Meaning
mtstBrowse:	
mtstLoad:	
mtstSave:	
mtstEmpty:	
mtstPack:	
mtstCheckPoint:	
mtstSearch:	
mtstUpdate:	
mtstSort:	

Delphi declaration

```
TkbmState =(mtstBrowse, mtstLoad, mtstSave, mtstEmpty, mtstPack, mtstCheckPoint, mtstSearch,
            mtstUpdate, mtstSort);
```

TkbmStreamFlagAppend File: kbmMemTable

Value	Meaning
sfSaveAppend:	
sfSaveInsert:	

Delphi declaration

```
TkbmStreamFlagAppend =(sfSaveAppend, sfSaveInsert);
```

TkbmStreamFlagBlobs File: kbmMemTable

Value	Meaning
sfSaveBlobs:	
sfLoadBlobs:	

Delphi declaration

```
TkbmStreamFlagBlobs =(sfSaveBlobs, sfLoadBlobs);
```

TkbmStreamFlagCalculated File: kbmMemTable

Value	Meaning
sfSaveCalculated:	
sfLoadCalculated:	

Delphi declaration

```
TkbmStreamFlagCalculated =(sfSaveCalculated, sfLoadCalculated);
```

TkbmStreamFlagData File: kbmMemTable

Value	Meaning
sfSaveData:	
sfLoadData:	

Delphi declaration

```
TkbmStreamFlagData =(sfSaveData, sfLoadData);
```

TkbmStreamFlagDataTypeHeader File: kbmMemBinaryStreamFormat

Value	Meaning
sfSaveDataTypeHeader:	
sfLoadDataTypeHeader:	

Delphi declaration

```
TkbmStreamFlagDataTypeHeader =(sfSaveDataTypeHeader, sfLoadDataTypeHeader);
```

TkbmStreamFlagDataTypeHeaders File: kbmMemBinaryStreamFormat

Set of TkbmMWStreamFlagDataTypeHeader

Delphi declaration

```
TkbmStreamFlagDataTypeHeaders=TkbmStreamFlagDataTypeHeader;
```

TkbmStreamFlagDef File: kbmMemTable

Value	Meaning
sfSaveDef:	
sfLoadDef:	
sfAutoLayoutDef:	
sfLoadByFieldNo:	

Delphi declaration

```
TkbmStreamFlagDef =(sfSaveDef, sfLoadDef, sfAutoLayoutDef, sfLoadByFieldNo);
```

TkbmStreamFlagDeltas File: kbmMemTable

Value	Meaning
sfSaveDeltas:	
sfLoadDeltas:	

Delphi declaration

```
TkbmStreamFlagDeltas =(sfSaveDeltas, sfLoadDeltas);
```

TkbmStreamFlagDontFilterDeltas File: kbmMemTable

A streamformatter setting which controls if deltas (older record versions) should be filtered out while saving.
This is currently implemented for the binary streamformatter only as the CSV streamformatter doesnt save deltas at all.

Delphi declaration

```
TkbmStreamFlagDontFilterDeltas=(sfSaveDontFilterDeltas);
```

TkbmStreamFlagFieldKind File: kbmMemTable

Value	Meaning
sfSaveFieldKind:	
sfLoadFieldKind:	

Delphi declaration

```
TkbmStreamFlagFieldKind =(sfSaveFieldKind, sfLoadFieldKind);
```

TkbmStreamFlagFiltered File: kbmMemTable

A streamformatter flag that controls if the currently defined filter on the table should be ignored while saving records.

Delphi declaration

```
TkbmStreamFlagFiltered=(sfSaveFiltered);
```

TkbmStreamFlagFromStart File: kbmMemTable

A streamformatter flag that controls if data loading should happen from the current spot in the stream or from the start.

Delphi declaration

```
TkbmStreamFlagFromStart=(sfLoadFromStart);
```

TkbmStreamFlagIgnoreMasterDetail File: kbmMemTable

A streamformatter flag that controls if master/detail settings should be ignored while saving data.

Delphi declaration

```
TkbmStreamFlagIgnoreMasterDetail=(sfSaveIgnoreMasterDetail);
```

TkbmStreamFlagIgnoreRange File: kbmMemTable

A streamformatter flag that controls if range settings should be ignored while saving.

Delphi declaration

```
TkbmStreamFlagIgnoreRange=(sfSaveIgnoreRange);
```

TkbmStreamFlagIndexDef File: kbmMemTable

Value	Meaning
sfSaveIndexDef:	
sfLoadIndexDef:	

Delphi declaration

```
TkbmStreamFlagIndexDef =(sfSaveIndexDef, sfLoadIndexDef);
```

TkbmStreamFlagLocalFormat File: kbmMemCSVStreamFormat

Value	Meaning
sfSaveLocalFormat:	
sfLoadLocalFormat:	
sfLoadAsASCII:	

Delphi declaration

```
TkbmStreamFlagLocalFormat =(sfSaveLocalFormat, sfLoadLocalFormat, sfLoadAsASCII);
```

TkbmStreamFlagLookup File: kbmMemTable

Value	Meaning
sfSaveLookup:	
sfLoadLookup:	

Delphi declaration

```
TkbmStreamFlagLookup =(sfSaveLookup, sfLoadLookup);
```

TkbmStreamFlagNoHeader File: kbmMemCSVStreamFormat

Value	Meaning
sfSaveNoHeader:	
sfLoadNoHeader:	

Delphi declaration

```
TkbmStreamFlagNoHeader =(sfSaveNoHeader, sfLoadNoHeader);
```

TkbmStreamFlagNonVisible File: kbmMemTable

Value	Meaning
sfSaveNonVisible:	
sfLoadNonVisible:	

Delphi declaration

```
TkbmStreamFlagNonVisible =(sfSaveNonVisible, sfLoadNonVisible);
```

TkbmStreamFlagPlaceholders File: kbmMemCSVStreamFormat

A streamformatter flag that controls if place holders will be provided for fields that arent supposed to be saved due to other settings. CSV streamformatter only.

Delphi declaration

```
TkbmStreamFlagPlaceholders=(sfSavePlaceholders);
```

See Also:

TkbmCustomCSVStreamFormat

TkbmStreamFlagQuoteOnlyStrings File: kbmMemCSVStreamFormat

A streamformatter flag that controls if only string values will be quoted while saving in CSV format.

Delphi declaration
```
TkbmStreamFlagQuoteOnlyStrings=(sfSaveQuoteOnlyStrings);
```

See Also:
TkbmCustomCSVStreamFormat

TkbmStreamFlagsAppend File: kbmMemTable

A streamformatter flag that controls if the data saved should be appended to the given stream/file or if it should be overwritten.

Delphi declaration

```
TkbmStreamFlagsAppend=TkbmStreamFlagAppend;
```

TkbmStreamFlagsAutoInc File: kbmMemCSVStreamFormat

Delphi declaration

```
TkbmStreamFlagsAutoInc=TkbmStreamFlagAutoInc;
```

TkbmStreamFlagsBlobs File: kbmMemTable

A streamformatter flag that controls if blob fields are to be saved/loaded when streaming data.

Delphi declaration

```
TkbmStreamFlagsBlobs=TkbmStreamFlagBlobs;
```

TkbmStreamFlagsCalculated File: kbmMemTable

A streamformatter flag that controls if the value of calculated fields should be saved/loaded while streaming.

Delphi declaration

```
TkbmStreamFlagsCalculated=TkbmStreamFlagCalculated;
```

TkbmStreamFlagsData File: kbmMemTable

A streamformatter flag that controls if regular data or internal calc fields should be streamed.

Delphi declaration

```
TkbmStreamFlagsData=TkbmStreamFlagData;
```

TkbmStreamFlagsDef File: kbmMemTable

A streamformatter flag that controls if the memtables metadata/definitions should be saved or loaded.

Delphi declaration

```
TkbmStreamFlagsDef=TkbmStreamFlagDef;
```

TkbmStreamFlagsDeltas File: kbmMemTable

A set of TkbmMWStreamFlagDeltas

Delphi declaration

```
TkbmStreamFlagsDeltas=TkbmStreamFlagDeltas;
```

TkbmStreamFlagsDisplayWidth File: kbmMemTable

Delphi declaration

```
TkbmStreamFlagsDisplayWidth=TkbmStreamFlagDisplayWidth;
```

TkbmStreamFlagsDontFilterDeltas File: kbmMemTable

Set of TkbmMWStreamFlagDontFilterDeltas

Delphi declaration

```
TkbmStreamFlagsDontFilterDeltas=TkbmStreamFlagDontFilterDeltas;
```

TkbmStreamFlagsFieldKind File: kbmMemTable

Set of TkbmStreamFlagFieldKind

Delphi declaration

```
TkbmStreamFlagsFieldKind=TkbmStreamFlagFieldKind;
```

TkbmStreamFlagsFiltered File: kbmMemTable

Set of TkbmStreamFlagFiltered

Delphi declaration

```
TkbmStreamFlagsFiltered=TkbmStreamFlagFiltered;
```

TkbmStreamFlagsFromStart File: kbmMemTable

Set of TkbmStreamFlagFromStart

Delphi declaration

```
TkbmStreamFlagsFromStart=TkbmStreamFlagFromStart;
```

TkbmStreamFlagsIgnoreMasterDetail File: kbmMemTable

Set of TkbmStreamFlagIgnoreMasterDetail

Delphi declaration

```
TkbmStreamFlagsIgnoreMasterDetail=TkbmStreamFlagIgnoreMasterDetail;
```

TkbmStreamFlagsIgnoreRange File: kbmMemTable

Set of TkbmStreamFlagIgnoreRange

Delphi declaration

```
TkbmStreamFlagsIgnoreRange=TkbmStreamFlagIgnoreRange;
```

TkbmStreamFlagsIndexDef File: kbmMemTable

Set of TkbmStreamFlagIndefDef

Delphi declaration

```
TkbmStreamFlagsIndexDef=TkbmStreamFlagIndexDef;
```

TkbmStreamFlagsLocalFormat File: kbmMemCSVStreamFormat

Set of TkbmStreamFlagLocalFormat

Delphi declaration

```
TkbmStreamFlagsLocalFormat=TkbmStreamFlagLocalFormat;
```

TkbmStreamFlagsLookup File: kbmMemTable

Set of TkbmStreamFlagLookup

Delphi declaration

```
TkbmStreamFlagsLookup=TkbmStreamFlagLookup;
```

TkbmStreamFlagsNoHeader File: kbmMemCSVStreamFormat

Set of TkbmStreamFlagNoHeader

Delphi declaration

```
TkbmStreamFlagsNoHeader=TkbmStreamFlagNoHeader;
```

TkbmStreamFlagsNonVisible File: kbmMemTable

Set of TkbmStreamFlagNonVisible

Delphi declaration

```
TkbmStreamFlagsNonVisible=TkbmStreamFlagNonVisible;
```

TkbmStreamFlagsPlaceHolders File: kbmMemCSVStreamFormat

Set of TkbmStreamFlagPlaceHolders

Delphi declaration

```
TkbmStreamFlagsPlaceHolders=TkbmStreamFlagPlaceHolders;
```

TkbmStreamFlagsQuoteOnlyStrings File: kbmMemCSVStreamFormat

Set of TkbmStreamFlagQuoteOnlyStrings

Delphi declaration

```
TkbmStreamFlagsQuoteOnlyStrings=TkbmStreamFlagQuoteOnlyStrings;
```

TkbmStreamFlagUsingIndex File: kbmMemBinaryStreamFormat

Streamflag controlling if saving is happening in current index order, or in the internal record list order.

Delphi declaration

```
TkbmStreamFlagUsingIndex=(sfSaveUsingIndex);
```

TkbmStreamFlagUsingIndexs File: kbmMemBinaryStreamFormat

Set of TkbmStreamFlagUsingIndex

Delphi declaration

```
TkbmStreamFlagUsingIndexs=TkbmStreamFlagUsingIndex;
```

TkbmStreamFormat File: kbmMemTable

A simple child class of TkbmCustomStreamFormat which publishes several events and properties.
It can be used as base for other specialized streaming classes.

Delphi declaration

```
TkbmStreamFormat=Class(TkbmCustomStreamFormat);
```

Public Events

TkbmThreadDataSet File: kbmMemTable

TkbmThreadDataSet is a pretty simple component which can be used to make other non thread safe tables, thread-safe.
It is not very usefull for linking dataaware controls to a table, since the table which protected by TkbmThreadDataSet will have no direct connection to any dataaware controls.

TkbmMemTable is itself thread-safe and doesnt need the protection of TkbmThreadDataSet.

Delphi Example
Example 1 - Typical operation of a TkbmThreadDataSet protected dataset.

Put a TDataset descendant (TTable f.ex) on a form/datamodule.
Put a TkbmThreadDataset on the form. For the sake of the example name it dsprotect.
Link dsprotect.Dataset to the TTable.
Set the TTable.Active property to true.

When you need the TTable in your code you should then write:

```
var
    t:TDataset;
begin
    t:=dsprotect.Lock;
    try
        // Do the operations you need to do here.
        t.Append;
        t.FieldByName('xxxx').AsInteger:=yyy;
        t.Post;
     finally
        dsprotect.Unlock;
     end;
end;
```

Public Properties/Variables

Dataset

Delphi declaration
```
Dataset: TDataset FDataset SetDataset;
```

Set this to the dataset which you want to protect.

IsLocked

Delphi declaration
```
IsLocked: boolean GetIsLocked;
```

Returns true if the dataset is allready exclusively locked by some other thread.

Public Methods/Functions

Create

Delphi declaration
```
Create(AOwner: TComponent); ;
```

Normal component contructor.

Destroy

Delphi declaration

```
Destroy; ;
```

Normal component destructor.

Lock

Delphi declaration

```
Lock: TDataset;
```

When you need to get exclusive access to the dataset protected by the TkbmThreadDataset, call Lock and the dataset will be returned to you for direct use.
Remember to call Unlock to release the dataset to other threads.

Delphi Example

```
 var
   t:TDataset;
begin
   l:=dsprotect.Lock;
   try
      ...
   finally
     dsprotect.Unlock;
   end;
end;
```

TryLock

Delphi declaration

```
TryLock(TimeOut: DWORD): TDataset;
```

Call this to try to get exclusive access to the dataset.
If the lock hasnt come thru before TimeOut millisecs (1/1000th sec), the function will return with the value nil.
You can use a TimeOut value of INFINITE ($FFFFFFFF) to make it wait forever. It will give same result as Lock.

Delphi Example

Example 1 - Try to get an exclusive lock. If not succeeded within 30 secs, give up.

t:=dsprotect.TryLock(30000);

Unlock

Delphi declaration

```
Unlock;
```

Releases the exclusive lock so other threads can try to lock the dataset for their use.

Protected Methods/Functions

Notification

Delphi declaration

```
Notification(AComponent: TComponent; Operation:
TOperation); ;
```

Internal support method for keeping the Dataset parameter valid.

TkbmUserBookmark File: kbmMemTable

A record structure which holds information about a bookmark and what table version it belongs to.

Delphi declaration
```
TkbmUserBookmark=
```

Public Properties/Variables

Bookmark

Delphi declaration
```
Bookmark: PkbmRecord;
```

DataID

Delphi declaration
```
DataID: longint;
```

RecordID

Delphi declaration
```
RecordID: TkbmNativeInt;
```

TkbmVersioningMode File: kbmMemTable

Value	Meaning
mtvm1SinceCheckPoint:	
mtvmAllSinceCheckPoint:	

Delphi declaration

```
TkbmVersioningMode =(mtvm1SinceCheckPoint, mtvmAllSinceCheckPoint);
```

TSQLTimeStamp File: kbmMemTable

Delphi declaration
```
TSQLTimeStamp=
```

Public Properties/Variables

Day

Delphi declaration
```
Day: Word;
```

Fractions

Delphi declaration
```
Fractions: LongWord;
```

Hour

Delphi declaration
```
Hour: Word;
```

Minute

Delphi declaration
```
Minute: Word;
```

Month

Delphi declaration
```
Month: Word;
```

Second

Delphi declaration
```
Second: Word;
```

Year

Delphi declaration
```
Year: SmallInt;
```

TUpdateStatusSet File: kbmMemTable

Set of updatestatus flags.

Delphi declaration

```
TUpdateStatusSet=TUpdateStatus;
```