Introduction

Rubicon is a powerful database search engine that benefits the end user, the developer, and the network manager. Rubicon provides native Delphi support as well as 16 and 32 bit DLL versions.

The Rubicon technology allows the end user of an application to perform searches for words or phrases using wildcards, to apply And, Or, Not, Near, and Like logic to the search, and the ability to iteratively narrow or widen the search without regard to the underlying database or field structure. Best of all, the search results are returned just as fast as a keyed search regardless of field type or the location of the word or string in the field.

From a developers standpoint, Rubicon encapsulates this robust search technology in a set of three straightforward native Delphi VCL components that build indexes, update indexes, and execute searches, respectively. Rubicon dramatically reduces the complexity associated with searching highly normalized tables and tables containing blob data. All the components are entirely written in Delphi and are compatible with Delphi 1.0 and 2.0. Paradox for Windows can access the Rubicon DLL's via the RUBICON.LSL interface.

Rubicon performs all searches at keyed index-like speeds by building a single Rubicon table that indexes all the words in the source table(s) and their locations. This means that most Rubicon searches never read the source table(s), an important consideration in secure environments. Reads and writes against this table are further minimized by built-in compression technology. For a stand alone users perspective, this speeds up the entire search process. For network managers, this means that the search minimizes the use of network bandwidth and both client and server CPU cycles. And by building Rubicon indexes during off peak hours, the network manager can further free up precious peak period network resources.

Delphi Installation

To install Rubicon into the component palette:

1. Create a new directory (e.g. d:\rubicon)
2. Unzip the files in the new directory
3. Delphi 1.0 trial run users unzip TARB16TR.ZIP and ST16TR.ZIP while Delphi 2.0 trial run users should unzip TARB32TR.ZIP and ST32TR.ZIP Registered users should unzip ST16.ZIP or ST32.ZIP for Delphi 1.0 or Delphi 2.0, respectively.
4. (Optional - registered users only) Using a text editor, modify TARUBICN.INC per the instructions contained within TARUBICN.INC
5. (Recommended) Backup COMPLIB.DCL or CMPLIB32.DCL
6. Start Delphi, select Options|Install Components (Delphi 1.0) or Components|Install (Delphi 2.0)
7. Click the Add button, then the Browse button and locate RUBICON.PAS in your new directory
8. Select it
9. Press OK in the Install Components dialog and wait for the Library to rebuild
10. (Trial run users) Programs can now use Rubicon while Delphi is running

Rubicon is now available in the Data Controls palette. See Using the Delphi Components to see how a simple application is put together.

To install the on-line help:

1. RUBICON.HLP and RUBICON.KWF should be in the same directory as TARUBICN.DCU
2. If Delphi is running, shut it down
3. (Recommended) Backup \DELPHI\BIN\DELPHI.HDX
4. Run \DELPHI\HELP\HELPINST
5. File|Open \DELPHI\BIN\DELPHI.HDX
6. If any existing KWF files are "not found", then add the appropriate search paths by selecting Options|Search Path
7. Select Keywords|Add File menu choice and select d:\RUBICON\RUBICON.KWF
8. File|Save
9. Exit the program
10. Check the WINHELP.INI file in the Windows directory and be sure that this entry is included: rubicon`.hlp=<fullpath>` where <fullpath> indicates the location of the help file

The Rubicon help file is now installed.

Paradox Installation

1. Create a new directory (e.g. d:\rubicon)
2. Unzip the files in the new directory
3. Unzip FILES16.ZIP or FILES32.ZIP depending on your platform (to use both, create separate directories for each)
4. Copy RBCNB16.DLL and/or RBCNB32.DLL to your \windows\system directory

Using the Delphi Components

A simple application using TMakeDictionary and TSearchDictionary requires the following steps:

1. Open a new application
2. Add a TTable (Table1) , connect it to BIOLIFE.DB in DBDEMOS, and set the Active property to True
3. Add a TDataSource (DataSource1) and set the DataSet property to Table1
4. Add a second TTable (Table2), set the DatabaseName to DBDEMOS, and TableName to 'Match'. Do not try to set Active to True! This table will later be created by SearchDictionary1.
5. Repeat step 4 (creating Table3), but set TableName to 'Words'. This table will later be created by MakeDictionary1.
6. Add another TDataSource (DataSource2) and set the DataSet property to Table2
7. Add a TDBGrid (DBGrid1) and set the DataSource to DataSource2 (the grid will be empty until we run the application and press the 'Search' button)
8. Add a TMakeDictionary (MakeDictionary1) and a TMakeProgress to the form, set DataSource to DataSource1 and WordsTable to Table3
9. Add a TSearchDictionary (SearchDictionary1) to the form, set the Builder property to MakeDictionary1, and set MatchTable to Table2. Be aware that whenever the Builder property is used, many of the properties in the Object Inspector become read only. See the Builder property in the reference section for a complete discussion of this behavior.
10. Add a TButton (Button1) to the form, set the caption to 'Make', double click on the button and add the following code: `MakeDictionary1.Execute`
11. Add a TEdit (Edit1) and clear the Text property
12. Add a second TButton (Button2), set the Caption to 'Search', double click on the button and add the following code: `SearchDictionary1.Search(Edit1.Text)`
13. Run the application and press the 'Make' button.
14. Move to Edit1 and enter 'sea'
15. Press the 'Search' button and six records appear in the grid. For most of the records, the word 'sea' is in the memo field (you may want to add a TDBMemo control to the form to make this Notes field visible).
16. Close the application, click on the MakeDictionary1 icon, double click on the DataTypes property, and set dtMemo to False
17. Repeat steps 13 - 15, but this time only one record is selected because the Notes field is no longer part of the dictionary

This example uses all the essential properties of these two components (TUpdateDictionary is very similar to TMakeDictionary). Properties next in importance would include IndexMode (common to all three components), SearchLogic (TSearchDictionary), and RankMode (TSearchDictionary).

While there are many other properties, the default values should suffice in the majority of situations.

Using the DLL (Paradox for Windows)

Rubicon for Paradox includes several examples of how to use the DLLs in a Paradox form:

- EXMAKE.FSL demonstrates how to use RBCNMAKE.LSL (32 bit only) and RUBICON.LSL to build a dictionary using library methods and direct API calls
- EXSEARCH.FSL demonstrates how to use RBCNSRCH.LSL (32 bit only) and RUBICON.LSL to search a dictionary using library methods and direct API calls
- EXNAV.FSL demonstrates how to perform a search and then navigate from matching record to matching record.
- EXUPDATE.FSL demonstrates how to perform updates
- RBCNDEMO.FSL is a comprehensive demo (Version 7 16/32 only)
- TEMPLATE.FSL is a convenient way for 16 bit Paradox users to access the DLL as it contains all the constants, types, and uses statements needed (see discussion below)

The example forms are discussed in more detail below. RBCNDEMO.FSL is discussed in the Demo Program section. You may wish to use the CodeView utility to browse the source code of the examples. See Utility Programs from more information on CodeView.

The example forms (those starting with EX) are designed to be used with Paradox tables, while the comprehensive demo may be used with any table type.

Paradox 7 32 bit users need only include the following code in a form's uses section to gain full access to all RUBICON.LSL constants, types, methods, and uses statements (this feature is referred to as extended uses syntax):

```
uses ObjectPAL
 "Rubicon.lsl"
 "RbcnMake.lsl"  ;// optional
 "RbcnSrch.lsl"  ;// optional
 "RbcnUpdt.lsl"  ;// optional
endUses
```

The optional libraries above provide a simpler interface for 32 bit developers by eliminating the Handle argument in the calling convention:

```
          libRubicon.setProperty(hMake, rblMinWordLen, 3)
```

becomes

```
          libRbcnMake.setProperty(rblMinWordLen, 3)
```

The only limitation is that the form can only work with one instance of the object at a time.

Because 16 bit developers do not have access to the extended uses syntax, all copies of all the needed constants and methods must be placed in the target form.

TEMPLATE.FSL simplifies this by providing a blank form that already includes all the constants, methods, etc., needed to develop a Rubicon application. Edit the form's open method to enable one or more of the three predefined handles, hMake, hSearch, and/or hUpdate.

If you are adding Rubicon to an existing 16 bit form, just copy the type, const, and uses sections from TEMPLATE.FSL to your form. You may also want to use the code in the open and close methods to allocate and deallocate handles.

The only differences between the 16 and 32 bit versions of the examples is that the 32 bit versions rely on the extended uses syntax. The 32 bit version references DLL routines with the STDCALL calling convention.

The table designated as the WordsTable may not be part of the form's data model. If your application is getting a 'table is busy' error, make sure your data model does not contain a conflicting table. The routine

```
      rbiOpenTest(TableName String) LongInt
```

may be useful in determining which table is causing the conflict. If the routine returns zero, it was unable to open the table. If it returns 1, it was successful.

EXMAKE.FSL: Like all the other examples, this form is designed to work with Paradox tables. Simply enter a source table using either a path name or alias and press one of the buttons to build the Rubicon dictionary. All the fields in the table will be included in the dictionary. Except for the Make Using RBI Calls button, the form will enumerate and display all the component's properties and will also use MAKEPROG.FSL to display its progress. Examine the code in each button's pushButton event to see how the component is initialized and executed.

EXSEARCH.FSL: After having run EXMAKE.FSL, use the same table (remember to include a path or alias) in this form to search for specific words. You may enter more than one word to search for. This example uses slAnd search logic (explained in more detail later) and creates and displays a match table.

EXNAV.FSL: This form demonstrates how to navigate through a table from one matching record to another. Use it the same way as EXSEARCH.FSL. When using the trial run DLL, the number of times you can move from record to record is limited during a given search.

EXUPDATE.FSL: This form shows how to use the update component and uses the messages.db table. First press the Make button to build the dictionary (this may overwrite the tables uses by the other examples). Only the fields Forum, Topic, and Message are included in the dictionary (see the form's Open event to see how the components are initialized). Next you may perform some searches (the search logic is set to slExpression, so you may use AND, OR, NOT, etc.). Then you may add or change records and check that the search engine recognizes the changes (If you delete a record, the dictionary will be invalidated since the index mode is imSeqNo). Use the Actions button to see recent action events and the Info button is review all the component's properties. The updating process is performed in the TableFrame's action event.

TMakeDictionary Component

Properties Methods Events Tasks See also

The TMakeDictionary component is used to scan the records of the DataSet specified by the DataSource property. The component compiles a dictionary of all the words used in the selected fields and their record locations in the table.

Use the Execute method to build the dictionary. This is a two phase process. During the first phase, the records are read and an in-memory dictionary of words and record positions is built. This is a memory intensive operation and there are several properties described below to help control resource consumption. The second phase writes the dictionary to the table specified by the WordsTable property. If the WordsTable table does not exist, it is created. If it does exist, it is deleted and recreated. During phase two, memory used in phase one is released.

You can control which fields are included in the dictionary several ways: use the field editor to select fields, use DataTypes/FieldTypes to filter which fields are processed, and by using the FieldNames property. Note, that these methods of restricting field processing use AND logic: the field must be in DataSource.DataSet.Fields, it must pass the DataTypes/FieldTypes filter, and if the FieldNames list is not empty, the FieldName must be in FieldNames.

TMakeDictionary provides build-in support for the following field types: ftString, ftSmallInt, ftInteger, ftWord, ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime, ftAutoInc (Delphi 2.0), and ftMemo. The OnProcessField event can be used to provide custom handling for all other field types.

Numeric fields generally are not good candidates for inclusion in the dictionary unless there are a fairly limited number of values and/or the user is likely to search for specific values.

String and memo fields are most commonly used to build a dictionary. Unlike numeric and date/time fields, TMakeDictionary has to break these fields in to separate words and uses the WordsDelim property to identify word separators. Short words can be excluded by using the MinWordLen property.

You may explicitly identify words to omit from the dictionary by using the OmitList property. After the build, you may also delete records from the WordsTable.

By default, all words are converted to upper case using the SysUtils.UpperCase function. This behavior may be overridden by setting the UpperCase property.

There are several methods that TMakeDictionary can use to keep track of where the word appears in the DataSource. This is controlled by the IndexMode property.

Use the OnPhaseOne and OnPhaseTwo events to monitor the progress of the process. The properties BlobBytesWritten, CacheCount, DiskInserts, MaxMemUsed, MemCompression, MemoryUsage, RecordNo, and State are useful indicators of execution. These events may also be used to abort the process.

Resource usage can be controlled by using the FileCompression, MemoryLimit, and RecordLimt properties. WordFieldSize, LikeFieldSize, and BlobFieldSize can be used to define the Word, Likeness, and BlobData fields in the WordsTable. KeyViolName specifies the key violation table name. Key violations usually result because of a too small WordFieldSize.

The OnProcessField event can be used to customize the filtering of fields, words, and parsing of strings to words and handle field types not supported by TMakeDictionary (ftUnknown, ftBytes, ftVarBytes, ftBlob, ftGraphic, ftFmtMemo, ftParadoxOle, ftDBaseOle, ftTypedBinary). Within this event, you may use ProcessField, ProcessList, ProcessPChar, and ProcessWord methods.

TSearchDictionary Component

Properties Methods Events Tasks See also

The TSearchDictionary component is used to search the dictionary specified by the WordsTable property by using the Search, NarrowSearch, and WidenSearch methods. These methods take a string argument, S, that may be a single or multiple word(s) separated by delimiters defined by the WordDelims property. Alternatively, the SearchMode and SearchFor properties and the Execute method may be used to conduct a search.

The type of search is controlled via the SearchLogic property which specifies slAnd, slOr, slNot, slNear, slLike, slPhrase ,and slExpression (see Expression Evaluation) logic. Wildcards may be used and are defined by the AnyChar and OneChar properties.

The MatchCount and RecordCount properties provide a numerical indicator on the success of the search. MatchCount reports how many words were located in the WordsTable , while RecordCount reports how many records met the search criteria.

When using wildcards, it is often useful to see the words that actually matched the search pattern. To generate this list, use the MatchingWords method (this method may also be used when wildcards are not used).

The MatchTable property can be used to create a result set of matching records. For many searches the only time that the DataSource is ever read by TSearchDictionary is during the creation of the MatchTable. The order of the records in the MatchTable is controlled by the RankMode property. The number of records added to MatchTable may be limited with the RecordLimit property.

The FindFirst, FindNext, FindPrior, FindLast, and Matches method can be used to locate matching records in the DataSource and manage filters.

There is no need for TSearchDictionary to be paired with TMakeDictionary in the same application. If they are paired together in the same application, use the Builder property to ensure that all the common properties are synchronized. If they are not paired together, then the following properties must be set to the same values used to build the dictionary: DataSource, DataTypes/FieldTypes, FieldNames, IndexMode, Likeness, MinOrdIndex, OnProcessField, StrictChecking, UpperCase, WordDelims, and WordsTable.

The WordsTable does not contain any information regarding which field(s) a word originated from, only that a word is associated with a record. This is usually not a problem unless you want to exclude a field(s) from a search that is already part of the WordsTable. To do this, use the SubFieldNames property.

TSearchDictionary ignores searches on words that have a length less than MinWordLen (unless it includes an AnyChar wildcard) or is in the OmitList.

Unlike TMakeDictionary, TSearchDictionary is not memory intensive. In fact, because of its architecture and compression options, it makes relatively small demand on resources (CPU and network) at the time the search is conducted.

You may elect to cache search results so that searches for the same word do not result in any disk or network activity.   Simply set the MemoryLimit property to the amount of memory you wish to devote to caching.   If the WordsTable is being updated while searches are being conducted, then search caching should not be enabled.

Indicators of resource consumption include BlobBytesRead, CacheReads, DiskReads, MaxMemUsed, MemoryUsage, and State.   Search progress may be monitored with the OnSearch event.

Searches against the dictionary may not be performed during a build or update.

TUpdateDictionary Component

Properties Methods Events Tasks See also

The TUpdateDictionary component may be used to keep the WordsTable synchronized with its DataSource. Alternatively, the WordsTable may by kept synchronized by simply rebuilding the table with TMakeDictionary. The tradeoffs of these two approaches is discussed in the next section.

For TUpdateDictionary to work, it needs to be notified when DataSource has a record deleted, inserted/appended, and edited. This is accomplished by adding the TUpdateDictionary routines AfterDelete, AfterPost, BeforeDelete, BeforeEdit, and BeforeInsert into the DataSource.DataSet events of the same name.

The choice of index modes plays a large part in determining the applicability of TUpdateDictionary to a DataSource. The best choice of IndexMode is imOrdinalIndex because it has the fewest restrictions, while imRecordNo and imSeqNo are limited to edits and appends (no insertions or deletions).

Updating a dictionary that has a low WordFieldSize property can lead to false additions to and deletions from the dictionary. For instance, if the word 'conglomerate' is added to a dictionary with WordFieldSize set to 11, it is entered as 'conglomerat'. An attempt to add 'conglomeration' would incorrectly associate it with 'conglomerat'.

Performance can be improved by allocating memory for caching. This is controlled by the MemoryLimit property. The cache memory stores records that have already been read. By setting the DelayedWrites property to True, writing records to disk is postponed until a call to FlushCache, Free, or WriteCache. Searches against the tables should not be conducted while DelayedWrites are enabled unless steps are taken to write the cache immediately before a search.

Resource usage and performance may be monitored with the following properties: BlobBytesRead, BlobBytesWritten, CacheEdits, CacheInserts, CacheReads, DiskDeletes, DiskEdits, DiskInserts, DiskReads, MaxMemUsed, MemCompression, MemoryUsage, and State. Statistics may be reset with the ResetStats method.

As with TSearchDictionary, TUpdateDictionary must use the same values for DataSource, IndexMode, Likeness, MinOrdIndex, StrictChecking, OmitList, UpperCase, WordDelims, and WordsTable as were used with TMakeDictionary to originally build the WordsTable. Additionally, TUpdateDictionary must have the same values for DataTypes/FieldTypes, FileCompression, FieldNames, and OnProcessField as TMakeDictionary. It is your responsibility to ensure this. If TUpdateDictionary is in the same application as TMakeDictionary, use the Builder property to automatically synchronize these properties.

TMakeDictionary versus TUpdateDictionary

Appearing below is a table summarizing the relative merits of TMakeDictionary versus TUpdateDictionary.

	TMakeDictionary	TUpdateDictionary
Purpose	Processing large numbers of records	Process one update at a time
Memory Usage	High	Low Caching increases usage
Speed Per Record	Fast memory based	Slow disk based Caching improves performance
Speed To Update One Record	Slow must process all records	Fast only processes one record
Speed To Update Many Records	May be faster than update depending on the scope of changes	May be faster than make depending on the scope of changes
Speed To Update All Records	Fastest	Much slower if caching disabled
IndexMode Limits	None	Best suited for imOrdinalIndex imRecordNo & imSeqNo limited to appends and edits

If the database is updated in a batch mode (e.g. overnight updates), then TMakeDictionary is probably the appropriate choice if the number of changes is substantial.

If the database is continually updated and the end user needs to be able to locate even the most recent changes, then TUpdateDictionary must be used.

Keep in mind that careful database design can minimize or eliminate the need for updating the dictionary. For instance, a parts database may consist of descriptions, inventories, and orders. While the inventories and orders portions of the database are going to be subject to frequent updates, the parts descriptions are probably relatively static. Thus, if the dictionary is created just on part descriptions, then the need to dynamically update the dictionary is minimized.

Index Modes

The index mode describes how the link is maintained between the words in the dictionary and the locations of the words in the search table. There are four index modes described below. See also Search Strategies for information on how the index mode affects global searches.

imOrdinalIndex

- Best choice when a primary or unique secondary key consisting of a single ordinal field is available
- When the StrictChecking property is False, this mode may be used with floating point fields as long as there are no fractional values (this is not checked)
- Available for all table types
- Scales easily to SQL
- Most compatible with dynamic updating
- Works well with filters
- Table must be open on the index containing the ordinal index field unless IndexFieldName is set

For best results, the index value should be sequential or nearly sequential (e.g. CustomerNo should be 1001, 1002, 1003 rather than 1010, 1020, 1030). Gaps created by record deletions are not a problem. It makes no difference whether the first (or lowest) index value is 1 or 100000. New records must not have an index value lower the first record when using TUpdateDictionary unless MinOrdIndex was used during the building of the dictionary.

imRecordNo

- Available only for dBase tables
- Use this option if you cannot use an imOrdinalIndex
- No range limits may be in place
- Only compatible with dynamic updating under certain restrictive conditions (only appends and edits allowed, no insertions permitted)
- Works poorly with filters
- Table may not be packed after the creation of the dictionary (dBase only)

imSeqNo

- Available only for Paradox files
- Same restrictions as imRecordNo
- No deletions are permitted
- During searches and updates, the table must use the same index used when the dictionary was created

imNone

No index available. None of the components will work with tables that have this IndexMode. You must either switch to another index and reset the IndexMode or create a compatible index.

Search Strategies

Searches are conducted against the words in the dictionary and are by default not case sensitive. The dictionary does not keep track of which field(s) the word was obtained from. This means that a search for the word 'green' could find records that contain 'Mr. Green' in the Name field, '125 Green Street' in the Address field, and 'Green Acres' in the City field.

While not requiring the user to specify individual fields to search is generally a plus, there may be instances when the search should be restricted to a subset of fields. In these cases, there are two options: one is to construct a second dictionary that is limited to the subset of fields. The limitation here is that the subset would need to be known ahead of time so that the dictionary could be pre-built (for small databases, this may not be an issue).

The second option would be to use SubFieldNames property to limit a search to a subset of the fields represented in the WordsTable. Using SubFieldNames forces the search to read the DataSource during each search. The only records read are those that match the search criteria before applying SubFieldNames. During the reading process, the SubFieldNames are checked to see whether they match the search criteria too.

The choice of IndexMode can have an impact on some global searches. To illustrate the problem, consider that a search for '*' using slOr logic should find every record. Conversely, a search for '*' using slNot logic should return no records. However, the slNot search will return a positive RecordCount when an imOrdinalIndex is used and there are gaps between index values. The records that are 'found' are really just the gaps in the index values. Since these 'records' do not really exist, a call to CreateMatchTable will return an empty table. To avoid this problem, slNot logic should not be used to begin a new search, only to narrow an existing search (try a slOr Search on '*', followed by a slNot NarrowSearch this will result in zero records found).
The UpperCase property can be used to override the default case conversion function.

Expression Evaluation

Version 1.10 of Rubicon introduced an new SearchLogic type, slExpression. Using slExpression, searches may take the form of:

```
windows
like windows
windows and driver and not video
windows near driver or "sound card"
(window* and driver) or (sound and card?)
```

slExpression allows the use of these familiar operators that are evaluated in the following precedence (highest appear first)

```
like, near
not
and, or
```

The syntax for these operators is

```
like <string>
<string> near <string>
not <expression>
<expression> or <expression>
<expression> and <expression>
```

where

`<string>` is a string or wildcard (e.g. windows, window*)

`<expression>` is a `<string>`, another operator, or parentheses enclosing an expression

Appearing below are some common mistakes:

Mistake	Solution
`(windows or driver) near video`	`windows near video or driver near video`
`like (problem or corruption)`	`like problem or like corruption`
`like 'delphi'`	`like delphi`
`like "borland delphi"`	`none`
`windows or driver not video`	`windows or driver and not video`
`windows driver`	`windows and driver`

Other common errors include not matching quotes (which may be paired single or double quotes) or parenthesis. When there is a syntax error, ErrorPos contains the approximate location of the error.

The following expressions are equivalent:

`windows and driver near video`	`windows and (driver near video)`
`windows or driver and not video`	`(windows or driver) and not video`
`windows and driver and video`	`((windows and driver) and video)`

The expression evaluator does not attempt to optimize the expression. This only becomes significant with searches using NEAR or phrases because these searches require reading the DataSource.

Working with Link, Lookup, or Normalized Tables

Performing a text search on a set of linked tables generally requires searching a field in a lookup table, grabbing its index value, returning to the master table, changing the index, finding the index value, etc., etc.   Now try performing a complex multi-field search!

Rubicon for Delphi eliminates this complexity by allowing you to build the dictionary with a DataSource that contains all the lookups.   Just use the Delphi field editor to define the relationships and process the table with TMakeDictionary, TSearchDictionary, and/or TUpdateDictionary.   Now you can search for any word in any field regardless of whether the field is in the master table or in a detail table.

Rubicon for Paradox can create linked tables by using the addLookupField method described in the Paradox Interface section.

Working with SQL Tables

When working with SQL tables, the IndexMode must be imOrdinalIndex. In order to perform all operations, Rubicon needs to calculate the difference between the minimum and maximum index values. This requires that a call be made to TTable.First and TTable.Last. On local tables, these operations are fast, but can be very slow on large SQL tables. There are two ways to improve performance.

In order to avoid moving to the first record, set MinOrdIndex to a value equal to or lower than the minimum index value when the WordsTable is built with TMakeDictionary. The same value for MinOrdIndex must be used to perform all subsequent updates and/or searches.

Moving to the last record can be avoided for searches by setting the SourceRange property to a value equal to or greater than the difference between the maximum and minimum index values. If MinOrdIndex is being used, MinOrdIndex plus SourceRange must be greater than or equal to the maximum index value.

Using MinOrdIndex and SourceRange with SQL tables turns off some internal integrity checks (thus avoiding the calls to First and Last), so these values must be set carefully.

Working with Huge Tables

When working with huge tables, you should

- Perform a test build by setting RecordLimit to about 4000 records, then inspect the WordsTable to see if there are any obvious characters and/or words which should be excluded from the build
- Shut down all other applications
- If possible, use a 32 bit version of the application
- Set MemoryLimit equal to the amount of physical RAM minus four megabytes (later, you may wish to experiment with this setting   it is not a hard and fast rule!)
- If using the 32 bit version, be sure AltMemMgr is set to True (Delphi only)
- See TARUBICN.INC for other performance options (Delphi only)

16 vs. 32 Bit Memory Issues

Memory is only an important factor for TMakeDictionary because all of it's basic operations occur in memory. TSearchDictionary and TUpdateDictionary require relatively little memory (to search a one million record search table would require less than 300kb of memory), however performance will benefit if additional memory is made available to cache indexes.

While Rubicon is compatible with both the 16 bit and 32 bit memory models, large tables builds are better suited to 32 bit environment. 16 bit applications requiring large amounts of memory are at a disadvantage because of the 16 bit memory suballocator (discussed in more detail below) and the 8192 global memory block limit (this is shared among all running applications).

If you must build a large dictionary with a 16 bit application, you should:

- Shut down all other applications
- Use an efficient index mode
- Minimize the number of fields included in the dictionary
- For Delphi applications, set memory suballocator variables HeapLimit and HeapBlock to 16384 and 65535, respectively
- Consider building the WordsTable on a 32 bit system and allow users to search the table from 16 bit applications.

For both 16 and 32 bit applications, the amount of virtual memory required to build a dictionary is approximately:

```
# of unique words * (IndexRange + 1) * (1 - compression rate) / 8
```

The IndexRange is the difference between the lowest index (or MinOrdIndex) and highest index values. When the IndexMode is imRecordNo or imSeqNo, the IndexRange is the same as the DataSource.DataSet.RecordCount - 1. For imOrdinalIndex, an inefficient index (one with gaps between index values) will result in inefficient use of memory.

Applying the formula to a table composed of one million records and 5,000 unique words using a imRecordNo or imSeqNo (or a very efficient imOrdinalIndex) IndexMode and a 97% compression rate would require 18.75mb of virtual memory.

Since the amount of virtual memory available is not always clear, TMakeDictionary will keep consuming memory as it needs it until it runs out. If you want to set an absolute limit on the amount of memory available to the component, add an OnPhaseOne event handler to monitor MemoryUsage and abort the process once the memory threshold has been exceeded.

In addition, 16 bit applications are limited to blob sizes of 64k unless you write an OnProcessField handler.

Delphi 2.0 Memory Fragmentation

Rubicon for Delphi caches and compresses indexes in memory in order to minimize disk/network activity. In doing so, it is frequently disposing large blocks of memory for small ones, or visa versa. Unfortunately, this pattern of behavior is the Achilles heel of the 32 bit memory suballocator and eventually leads to massive memory fragmentation which will grind the application (but not the system) to a halt.

Fragmentation usually does not become a problem unless the DataSource has more than 50,000 records and 15,000 words. This is an approximate threshold, and will vary with the amount RAM devoted to caching. The problem is most likely to affect TMakeDictionary since it goes through the most compress and decompress cycles during execution. TUpdateDictionary may be affected if a very large number of records are updated during execution and caching is enabled. TSearchDictionary should not be affected even if caching is enabled since the number or records cached is likely to be very small.

Tools such as MemorySleuth 1.0 do not catch this bug. The Windows 95 System Monitor will. You may use this tool to determine whether your application is being affected by fragmentation. If SM shows memory use increasing even after the component has reached its MemoryLimit and the performance of the application is degrading, then fragmentation is the likely cause.

Fragmentation does not lead to a memory leak. All memory used by the components are returned to the system when they are freed or done processing.

There are suggestions that Borland will release an upgraded System unit that will include a fix for this problem.

Rubicon for Delphi 1.10 provides an alternative memory manager which works around this bug. It uses an algorithm that is optimized to work with the TMakeDictionary pattern of memory use. To use this option, the `AltMemMgr` compiler directive in TARUBICN.INC must be enabled and the AltMemMgr property must be set to True. This option does not replace the existing memory manager (i.e. it does not call SetMemoryManager), but rather supplements GetMem and FreeMem.

Unlike TMakeDictionary, TUpdateDictionary has a much more unpredictable pattern of memory use so it is more difficult to assure that the alternative memory manager will not also defragment memory. If you are processing a large number of changes to a table and are using caching, then you may wish to call FlushCache periodically.

The alternative memory manager eliminates the fragmentation problem by creating a list of pointers available for reuse (a memory pool). When execution begins, this list is empty and requests for memory are passed to GetMem. As execution proceeds, any memory that is released is saved in the memory pool. Subsequent requests for memory first check the memory pool to see if there is a pointer available of the appropriate size. If one exists, it is used, otherwise GetMem is called.

When the alternative memory manager is used, MemoryUsage may exceed MemoryLimit by a large amount. MemoryUsage is largely made up of the memory used to hold data structures and memory pool. The MemoryLimit is compared to only the portion of MemoryUsage that is actually holding data, and thus the memory pool portion is excluded.

Performance Optimization

Performance can be optimized by:

- Use a 32 bit version of the application
- Narrowing the list of fields selected for inclusion in the index
- Use an efficient index (see Index Modes)
- Shut down all other applications
- Add more memory
- If you own SysTools, turn off the ThreadSafe compiler option (Delphi only)
- Enable the dbiWrite option in TARUBICN.INC if you are using Paradox, Local InterBase, InterBase 4.0, or other servers that supports 32 bit integers (Delphi only)
- For server based tables, build the WordsTable locally, then move it to a server
- Examine the underlying structure of the database to determine whether Rubicon needs to be applied against the entire database or a just subset of the database
- See TARUBICN.INC for other performance options (Delphi only)

Utility Programs

Two utility programs are included with Rubicon for Delphi: Verify and Compare.   Use Verify to check the integrity of a WordsTable.   If Verify reports any errors, the WordsTable should be rebuilt.

Use the Compare utility to compare two WordsTables.   Generally, two WordsTables will only pass the Compare tests when they are exactly the same.   WordsTables that have different table types may pass the test if all the words consist of standard characters (international characters may be treated differently by the table types and therefore cause differences).

These utilities are available for DLL users.   Contact Tamarack Associates.

Rubicon for Paradox includes a utility, CodeView.fsl, that displays the source code of any form or library in a convenient manner.   Enter the form or library's name in the FSL/LSL edit box.   Libraries must include the LSL extension.   Press enter and the source code will be displayed.   Double click on the Object or MethodName titles to change the sort order.   Resize the form to increase the viewing area.

Demo Program

Delphi: To compile the demo program, open DEMO.DPR, load the BOLTS.ICO icon (in Delphi 1 select Options|Project|Application|Load Icon, in Delphi 2 Project|Options|Application|Load Icon), and turn break on exceptions off (Delphi 1 select Options|Environment|Preferences, Delphi 2 Tools|Options|Preferences). Then press F9.

Paradox for Windows 7: Run the form RBCNDEMO.FSL, which closely matches the Delphi demo (a compiled version is available on our web site which may be helpful to users who do not have Paradox 7). There are some differences, but the instructions below should be a very good guide. The sample search references the table BIOLIFE.DB which you will not have unless you also own Delphi, so you will need to substitute your own table.

Tables Tab

Alias

Select from the available aliases. For SQL aliases, you will be prompted for a password.

Table

Select a table to search. Generally, you should select a table from the drop down list. You may also enter the path + table name. This table is ReadOnly, so the demo program only reads from the table.

Index

Select the index for the table. Usually the primary index is the appropriate choice. See Index Mode discussion on the Configure Tab below.

Available Fields

Lists the available fields in the table.

Selected Fields

Lists the fields selected for inclusion in the search dictionary. Generally, you will only want to include fields that are string (or char) and memo types and exclude numeric fields.

Add Link

Create a link to another table (also called lookups or calculated fields).

Edit Link

Edit an existing link.

Build Tab

Statistics

Build statistics include elapsed time, word count (number of words in the dictionary), memory usage, the table size, the blob field size, and compression rates.

Memory usage includes the vast majority of memory used by the application to build the dictionary, but excludes some data structures.

The table size represents the amount of data written to the table and may not necessarily correspond to the table size reported by File Manage or Explorer (after packing a Paradox table, they are close, however). The table size excludes the blob data which is reported separately.

The blob data size figure also may not correspond to the physical table size as it does not take into account the physical structure of the table.

Tables

Specify the table names for tables created by the demo. The **Words** table holds the words contained in the Search table. The table will be created in the alias specified in the Table tab. The **Key Viol.** table is used to store any key violations. This will always be a Paradox table and will be located in the path or alias specified in the edit box.

The **File Compression** checkbox compresses information inside the dictionary. This option should be left on unless the table resides on a compressed disk drive that offers superior compression to the build-in compression routines (this would be determined by comparing the file size with and without the file compression option enabled). The amount of blob data compression will vary with the frequency that a word appears in multiple records (the more often it shows up, the less compression achieved), but compression rates of 95% or more are not unusual.

Word Delimiters

These characters define the beginning and end of a word. Control characters can be entered as ^M and ^J. To enter a ^, use ^^.

The most common delimiters are spaces, commas, and periods. You will probably want to include other punctuation (colon, semi-colon, double quotes, single quotes, question marks, exclamation marks), parentheses, braces, brackets, and mathematical symbols.

Whether you want to include any of these @#$%&\~ depends on your database.

In some instances, you may consider using numbers as word delimiters. This will effectively eliminate all numbers from the dictionary.

Be aware that the period delimiter causes havoc with numerical values embedded in string or memo fields: a number like '19.95' would become two words: '19' and '95'.

Word delimiters are not applied to any numerical, boolean, date, or time fields, only to string and memo fields.

Index Mode

There are four available Index Modes:

- imOrdinalIndex: Best choice when a primary or unique secondary key consisting of a single ordinal field is available. When Strict Checking is disabled, this mode may be used with floating point fields as long as there are no fractional values (this is not checked). This option is available for all table types, scales easily to SQL, and is most compatible with dynamic updating.

For best results, the index value should be sequential or nearly sequential (e.g. CustomerNo should be 1001, 1002, 1003 rather than 1010, 1020, 1030).

- imRecordNo: Available only for dBase files. Use this option if you cannot use an imOrdinalIndex. Only compatible with dynamic updating under certain restrictive conditions (only appends allowed, no deletes or insertions permitted).
- imSeqNo: Available only for Paradox files. Same restrictions as imRecordNo. During searches, the table must use the same index used when the dictionary was created.

If your table does not have an ordinal index or you have an ordinal index but there are large increments between index values, it is recommended that you add an additional field and create a unique secondary index and populate the field with sequential index values (gaps due to deleted records are not a problem).

When checked, **Strict Checking** only allows the imOrdinalIndex option to be used when there is a single ordinal field primary key or unique secondary index. When unchecked, the restriction on the ordinal field is relaxed to include numeric or floating point fields as well. Floating point fields will only work if there are no fractional values in the index (this is not checked).

Other Build Options

Words with a length less than the **Minimum Word Length** will not be included in the dictionary. A minimum word length of three would therefore exclude words like 'a', 'of', 'we', etc., which are generally not very useful in a search. Note that it may exclude some useful state (CA, NY), company (3M, TI), and other (PC) abbreviations and words like 'go' and 'hi'.

Record Limit may be used for limiting the build to the first N records for testing purposes.

Memory Limit is used during the build process up to this limit, at which point the least frequently used items are compressed. If all indexes in memory have been compressed, the build process will continue to consume memory beyond this limit.

When checked, the **Alternative Memory Manager** check box enables the use of a memory allocation algorithm designed to work around a Delphi 2.0 memory fragmentation bug. This bug primarily affects the processing of tables with a large number of records. This option is disabled in the 16 bit demo.

Search Tab

Search For

Enter the word(s) to search for. Words can be separated by any of the word delimiters defined in the configuration tab. The default wildcards are '*' and '?'. Searches are not case sensitive. See Search Logic below for examples.

Search Button

Begins a new search.

Narrow Button

Scope of the search is limited to the records already selected during prior searches.

Search Results

Shows the results of the search in terms of the number of words that matched the search criteria and the number of matching records (a function of the words found and the search logic).   Keep in mind that when using an slAnd search, the more words that are found, the less likely there will be individual records that contain all the words.   The elapsed time of the search is also displayed.   This figure does not include the time needed to update the match table.

Press the **Words** button to switch to the Matching Words panel.   Press the **Records** button to view the match table.

Search Logic

There are seven search types: slAnd, slPhrase, slLike, slNear, slOr, slNot, and slExpression.   The three most common are: slAnd which searches for records that contain all instances of the words in the Search For combo box; slOr which searches for records that contain at least one instance of the words in the Search For combo box; and slNot which selects all records that do not contain instances of the words in the Search For combo box.

slLike searches for words that evaluate as the same using the Likeness function (the results of which appear in the Words table in the Likeness field).

slNear searches for two words that are within NearWord (see the Near Word option below) words of one another in a field(s).   If the number of words in the search is not two, an error is raised.

slPhrase searches for words in a specific order of appearance in the field(s).

slExpression enables expression evaluation using AND, OR, NOT, LIKE, NEAR, "quoted phrase searches", and parentheses.

Search Examples

Logic	Example	Comment	slExpression Equivalent
slAnd	`delphi paradox`	ANDs each word	`delphi and paradox`
slOr	`delphi paradox`	ORs each word	`delphi or paradox`
slNot	`access`	NOTs each word	`not access`
slLike	`computer`	LIKEs each word Wildcards ignored	`like computer`
slNear	`delphi paradox`	Two word limit	`delphi near`

		Must read table	`paradox`

slPhrase	`database engine`	Must read table	`"database engine"`
			`'database engine'`

Search Mode

Use smSearch to begin a new search. smNarrow ANDs the results of the current search against the prior search, thereby narrowing the search results. smWiden is like smNarrow, but uses OR logic to widen or expand the search results.

Rank Mode

Determines how records are ordered in the Match Table. There are three Rank Modes: rmNone, rmCount, and rmPercent. rmNone leaves the records in index order. rmCount adds a Rank field to the table that contains a count of the matching words. rmPercent is like rmCount, except it uses a 100 scale.

Fields

By default, all searches are conducted against all the Selected Fields. However, this option allows the search to be applied against a subset of fields (searches may not be conducted on excluded fields without rebuilding the Words Table).

Caching

Memory Limit specifies how much memory is made available for caching. **Reset Stats** button resets the caching statistics. **Flush Cache** clears the cache.

Matched Words

A list of words found that match the search criteria. This list is especially useful when wildcards are being used. This is a cumulative list of matching words.

Other Search Options

The **Match Table** contains the records located during searches. The table will be created in the alias specified in the Table tab. **Record Limit** sets a maximum number of records allowed in the Match Table.

Use **Any Char** and **One Char** to set the characters used as wildcards.

Near Word specifies how close two words must be to qualify as near.

Links Dialog Box

The **Data Field** identifies the field from which the link is created.

The **Link Table** identifies the table containing the Link Field.

The **Link Field** identifies the field to which the link is created.

The **Link Display** identifies the field to be displayed in the table. Multiple fields may be entered by separating the field names with semicolons (e.g. Name;Company;Address).

A Sample Search

1. Select the Tables tab
2. Select the DBDEMOS alias
3. Select the BIOLIFE.DB table
4. Use the menu to view the search table (View|Search Table)
5. Close the BIOLIFE.DB grid
6. In the Available list box double click on Common_Name, Notes, and Species Name to move them to the Selected list box
7. Using the menu, select View|Show All Fields
8. Press Ctrl+T to reopen the BIOLIFE.DB grid and note that only the selected fields are now displayed
9. Scroll the grid so that the Notes column is visible
10. Double click on one of the Notes fields and the memo contents are displayed
11. Close the Notes memo and the BIOLIFE.DB grid
12. Press the Next button
13. Press the Build button (compression will be negative for very small tables)
14. Using the menu, select View|Words Table to view the words in the dictionary
15. Close the WORDS.DB grid
16. Press the Next button
17. Enter 'edibility night' in the Search For combo box and press enter
18. The search results in two words being found and four matching records
19. Press Ctrl+M (or press the Records button) to view the records that match the criteria (the matches are all in the Notes field)
20. Return to the Rubicon Demo form (you may leave the matched grid visible)
21. Press Alt+O to change the Search Logic to Or
22. Press the Search button
23. Now 24 records have been selected
24. Clear the contents of the Search For combo box and enter 'areas'
25. Press Alt+T to change the Search Logic to Not
26. Select the Search Mode option and change the Search Mode to smNarrow.
27. Press the Narrow button
28. The word 'area' was found once and narrowed the number of selected records to 13
29. Clear the contents of the Search For combo box and enter 'edibility or night and not areas'

30. Change the Search Mode to smSearch

31. Select the Search Logic option and press Alt+X to change the Search Logic to slExpression

32. Press the Search button and again 13 records are found

33. Press File|Save to save the settings and results

AfterDelete method

Applies to

TUpdateDictionary

Declaration

```
procedure AfterDelete;
```

Insert this method into the DataSource.DataSet event of the same name so that TUpdateDictionary can be notified of changes and thereby keep the WordsTable synchronized with the DataSource.

The purpose of this method is to check for invalid deletions, and if one is identified, to raise an error.

When IndexMode is imRecordNo, no errors are raised as the WordsTable remains valid unless the DataSource is packed.

When IndexMode is imOrdinalIndex, any record may be deleted except the first record (in index order) unless MinOrdIndex has been set.

Example

```
procedure TForm1.Table1AfterDelete(DataSet: TDataSet);
begin
 UpdateDictionary1.AfterDelete
end;
```

See also

AfterPost, BeforeDelete, BeforeEdit, BeforeInsert , TUpdateTable

AfterPost method

Applies to

TUpdateDictionary

Declaration

```
procedure AfterPost;
```

Insert this method into the DataSource.DataSet event of the same name so that TUpdateDictionary can be notified of changes and thereby keep the WordsTable synchronized with the DataSource.

Example

```
procedure TForm1.Table1AfterPost(DataSet: TDataSet);
begin
 UpdateDictionary1.AfterPost
end;
```

See also

AfterDelete, BeforeDelete, BeforeEdit, BeforeInsert , TUpdateTable

AltMemMgr property

Applies to

TMakeDictionary, TUpdateDictionary

Declaration

```
{$IFDEF AltMemMgr}
property AltMemMgr: Boolean;
{$ENDIF}
```

Setting AltMemMgr to True enables the alternative memory management code to be used for cache memory. This option must be used when processing a DataSource that contain a large number of records in a 32 bit application. Some performance gain (up to 3%) may be realized by turning this option off for small to medium tables. This option does not replace the existing memory manager. See Delphi 2.0 Memory Fragmentation for more details.

The compiler directives in TARUBICN.INC determine whether this property is available. By default, the property is available under Delphi 2.0 and not available under Delphi 1.0.

Default is True.

This property may be removed in future releases if Borland provides a fix to the problem.

Example

```
MakeDictionary1.AltMemMgr := True;
```

See also

MemoryLimit

AnyChar property

Applies to

TSearchDictionary

Declaration

```
property AnyChar: Char;
```

The wildcard that matches any series of characters.   Must be different from OneChar. Default is '*'.

Example

```
SearchDictionary1.AnyChar := '%';
```

See also

NarrowSearch, OneChar, Search

AutoClose property

Applies to

TMakeProgress

Declaration

```
property AutoClose: Boolean;
```

When True, the progress form will automatically close upon completion of the dictionary build.   If False, it will remain open until the user closes the form.

Default is True.

Example

```
MakeProgress1.AutoClose := False;
```

BatchAdd method

Applies to

TUpdateDictionary

Declaration

```
procedure BatchAdd;
```

Whenever multiple records are added to the DataSource using TBatchMove, copy, or equivalent command, the WordsTable can be updated by moving the cursor to each added record and calling BatchAdd.

Example

```
with Table1 do
 begin
  Last;
  Bookmark := GetBookmark;
  { add the records }
  GotoBookmark(Bookmark);
  FreeBookmark(Bookmark);
  Next;
  while not Eof do
   begin
    UpdateDictionary1.BatchAdd;
    Next
   end
 end;
```

See also

BatchDelete, BeforeEdit, BeforeInsert

BatchDelete method

Applies to

TUpdateDictionary

Declaration

```
procedure BatchDelete;
```

Whenever multiple records are deleted from the DataSource using TBatchMove, subtract, or equivalent command, the WordsTable can be updated by moving the cursor to each record to be deleted and calling BatchDelete, and then performing the delete.

Example

```
with Table1 do
 begin
  Bookmark := GetBookmark;
  while not Eof do
   begin
    UpdateDictionary1.BatchDelete;
    Next
   end;
  GotoBookmark(Bookmark);
  FreeBookmark(Bookmark);
  { delete records from current location to Eof }
 end;
```

See also

BatchAdd, BeforeDelete

BeforeDelete method

Applies to

TUpdateDictionary

Declaration

```
procedure BeforeDelete;
```

Insert this method into the DataSource.DataSet event of the same name so that TUpdateDictionary can be notified of changes and thereby keep the WordsTable synchronized with the DataSource.

Example

```
procedure TForm1.Table1BeforeDelete(DataSet: TDataSet);
begin
 UpdateDictionary1.BeforeDelete
end;
```

See also

AfterDelete, AfterPost, BeforeEdit, BeforeInsert , #LJTUpdateTable#TUpdateTable

BeforeEdit method

Applies to

TUpdateDictionary

Declaration

```
procedure BeforeEdit;
```

Insert this method into the DataSource.DataSet event of the same name so that TUpdateDictionary can be notified of changes and thereby keep the WordsTable synchronized with the DataSource.

Example

```
procedure TForm1.Table1BeforeEdit(DataSet: TDataSet);
begin
 UpdateDictionary1.BeforeEdit
end;
```

See also

AfterDelete, AfterPost, BeforeDelete, BeforeInsert , #LJTUpdateTable#TUpdateTable

BeforeInsert method

Applies to

TUpdateDictionary

Declaration

```
procedure BeforeInsert;
```

Insert this procedure into the DataSource.DataSet event of the same name so that TUpdateDictionary can be notified of changes and thereby keep the WordsTable synchronized with the DataSource.

Example

```
procedure TForm1.Table1BeforeInsert(DataSet: TDataSet);
begin
 UpdateDictionary1.BeforeInsert
end;
```

See also

AfterDelete, AfterPost, BeforeDelete, BeforeEdit, #LJTUpdateTable#TUpdateTable

BlobBytesRead property

Applies to

TSearchDictionary, TUpdateDictionary

Declaration

```
property BlobBytesRead: LongInt;
```

Returns the number of Blob bytes read from the WordsTable. Does not include any Blob field related overhead (which varies by table type) or any Blob bytes read from the DataSource.

Run-time and read only.

Example

```
BlobBytesReadLabel.Caption :=
IntToStr(SearchDictionary1.BlobBytesRead);
```

See also

BlobBytesWritten, CacheEdits, CacheInserts, CacheReads, DiskDeletes, DiskEdits, DiskInserts, DiskReads, ResetStats

BlobBytesWritten property

Applies to

TMakeDictionary, TUpdateDictionary

Declaration

```
property BlobBytesWritten: LongInt;
```

Returns the number of Blob bytes written to the WordsTable.   Does not include any Blob field related overhead (which varies by table type).

Run-time and read only.

Example

```
BlobBytesWrittenLabel.Caption :=
                 IntToStr(MakeDictionary1.BlobBytesWritten);
```

See also

BlobBytesRead, CacheEdits, CacheInserts, CacheReads, DiskDeletes, DiskEdits, DiskInserts, DiskReads, ResetStats

BlobFieldSize property

Applies to

TMakeDictionary

Declaration

```
property BlobFieldSize: Integer;
```

This property defines the size of the BlobData field in the WordsTable. For Paradox tables, higher values increase the size of the DB file and reduce the size of the MB file. Setting the value too high results in excessive wasted space in the DB file. Setting it too low causes more data to be saved to the MB file and, because the MB file has to be accessed more frequently, degrades build performance.

Not available for all table types. Default is 32.

Example

```
MakeDictionary1.BlobFieldSize := 40;
```

See also

WordFieldSize

Builder property

Applies to

TSearchDictionary, TUpdateDictionary

Declaration

```
property Builder: TBuildDictionary;
```

Setting Builder to the TBuildDictionary (which is the ancestor to TMakeDictionary and TUpdateDictionary) that made the WordsTable ensures that all the values in common are set correctly.

When a value is assigned to Builder, the following properties become read only: DataSource, DataTypes, FieldNames, FieldTypes, FileCompression, IndexMode, Likeness, MinOrdIndex, MinWordLen, OnProcessField, StrictChecking, UpperCase, WordDelims, WordsTables.

Example

```
SearchDictionary1.Builder := MakeDictionary1;
```

See also

DataSource, DataTypes, FieldNames, FieldTypes, FileCompression, IndexMode, Likeness, MinOrdIndex, MinWordLen, OnProcessField, StrictChecking, UpperCase, WordDelims, WordsTables.

CacheCount property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property CacheCount: LongInt;
```

The number of unique words in the cache.

During a TMakeDictionary.Execute, the number of words in the cache is the same as the number of unique words processed plus the number of omitted words (i.e. OmitList.Count).

Run-time and read only.

Example

```
WordCountLabel.Caption := IntToStr(MakeDictionary1.CacheCount);
```

See also

MemoryLimit

CacheEdits property

Applies to

TUpdateDictionary

Declaration

```
property CacheEdits: LongInt;
```

Returns the number of edits to WordsTable that were not immediately written to disk but saved to cache, therefore saving disk write time.   DelayedWrites must be True in order for there to be any CacheEdits.

Run-time and read only.

Example

```
with UpdateDictionary1 do
 if CacheEdits + CacheInserts > 0 then
  begin
   WriteCache;
   ResetStats
  end;
```

See also

BlobBytesRead, BlobBytesWritten, CacheInserts, CacheReads, DiskDeletes, DiskEdits, DiskInserts, DiskReads, ResetStats

CacheInserts property

Applies to

TUpdateDictionary

Declaration

```
property CacheInserts: LongInt;
```

Returns the number of inserts to WordsTable that were not immediately written to disk but saved to cache, therefore saving disk write time. DelayedWrites must be True in order for there to be any CacheInserts.

Run-time and read only.

Example

```
with UpdateDictionary1 do
 if CacheEdits + CacheInserts > 0 then
  begin
   WriteCache;
   ResetStats
  end;
```

See also

BlobBytesRead, BlobBytesWritten, CacheEdits, CacheReads, DiskDeletes, DiskEdits, DiskInserts, DiskReads, ResetStats

CacheReads property

Applies to

TSearchDictionary, TUpdateDictionary

Declaration

```
property CacheReads: LongInt;
```

Returns the number of cache reads or hits.   This is the count of WordsTable records read from memory, and therefore represent records not read from disk.

Run-time and read only.

Example

```
CacheReadsLabel.Caption :=
IntToStr(UpdateDictionary1.CacheReads);
```

See also

BlobBytesRead, BlobBytesWritten, CacheEdits, CacheInserts, DiskDeletes, DiskEdits, DiskInserts, DiskReads, ResetStats

CreateMatchTable method

Applies to

TSearchDictionary

Declaration

```
procedure CreateMatchTable(Table: TTable);
```

Call this procedure after a search to fill the Table with records from the DataSource matching the search criteria. The number of records copied into Table is the lesser of RecordCount or RecordLimit. The RankMode property control the order of the records in the Table. If Table exists before the call to CreateMatchTable, it is deleted.

If RecordCount is greater than RecordLimit, then matching records up to RecordLimit are added to the Table. These records are added in index order irrespective of any RankMode that may be in place (ranking occurs after the match table has been created).

Only the fields that are in DataSource.DataSet.Fields are included in Table. Fields that have a DataType of ftAutoInc are translated to ftInteger.

Example

```
with SearchDictionary1 do CreateMatchTable(Table1);
```

See also

MatchTable, RankMode, RecordLimit

DataSource property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property DataSource: TDataSource;
```

The DataSource property determines where the component obtains the data to be searched.   DataSource.DataSet may be set to ReadOnly.

The fields included in the dictionary are the fields accessible in the DataSource.DataSet.Fields property.   If you do not specify any fields in the Delphi IDE field editor, then all fields are included.   The list of fields included in the dictionary can be further limited by the DataTypes/FieldTypes and FieldNames properties.

To include link or lookup fields from other tables, simply add calculated fields to the DataSource.DataSet.

Example

```
MakeDictionary1.DataSource := DataSource1;
```

See also

DataTypes, FieldNames, FieldTypes, WordsTable, *Working with Link, Lookup, or Normalized Tables*

DataTypes property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property DataTypes: TDataTypes;
```

Because FieldTypes is too large of a set to be handled by the property editor, a property editor compatible set, DataTypes, is provided to access the most common values of FieldTypes. With DataTypes you can set all FieldTypes except ftUnknown, ftGraphic, ftFmtMemo, ftParadoxOle, ftDBaseOle, and ftTypedBinary (to set these types, use FieldTypes in your code).

While you can read and write to DataTypes at run time, you should use FieldTypes instead.

Before the contents of a field are added to the dictionary or are accessed as part of a search, three checks are made: first, the field must be a member of DataSource.DataSet.Fields; second, the DataType of the field is checked to see if it is in FieldTypes; and, third, if there are entries in the FieldNames list, the FieldName is checked against a list of FieldNames.

Default is [dtString, dtMemo] for Delphi, all fields for DLL applications.

Example

```
DataTypes := [dtString..dtDateTime,dtMemo];
```

See also

FieldNames

DelayedWrites property

Applies to

TUpdateDictionary

Declaration

```
property DelayedWrites: Boolean;
```

Enables delayed writes. This means that changes to records in the WordsTable are not written to disk until the cache is full, WriteCache, FlushCache, or Free is called. Turning this property off causes any unwritten records to be written.

Use this property to speed up the updating process. When the cache is full, the least recently used records are written to disk. This process may slow down your application.

No searches should be conducted against the table while using DelayedWrites.

Default is False.

Example

```
UpdateDictionary1.DelayedWrites := True;
```

See also

CacheEdits, CacheInserts, FlushCache, WriteCache

DiskDeletes property

Applies to

TUpdateDictionary

Declaration

```
property DiskDeletes: LongInt;
```

Returns the number of records (and therefore words) deleted from the WordsTable during the update process.

Note that there is no corresponding cache property for DiskDeletes since all deletes are immediately written to disk.

Run-time and read only.

Example

```
DiskDeletesLabel.Caption :=
IntToStr(UpdateDictionary1.DiskDeletes);
```

See also

BlobBytesRead, BlobBytesWritten, CacheEdits, CacheInserts, CacheReads, DiskEdits, DiskInserts, DiskReads, ResetStats

DiskEdits property

Applies to

TUpdateDictionary

Declaration

```
property DiskEdits: LongInt;
```

Returns the number of records that have been edited from the WordsTable during the update process.

Run-time and read only.

Example

```
DiskEditLabel.Caption := IntToStr(UpdateDictionary1.DiskEdits);
```

See also

BlobBytesRead, BlobBytesWritten, CacheEdits, CacheInserts, CacheReads, DiskDeletes, DiskInserts, DiskReads, ResetStats

DiskInserts property

Applies to

TMakeDictionary, TUpdateDictionary

Declaration

```
property DiskInserts: LongInt;
```

Returns the number of records added to the WordsTable.

Run-time and read only.

Example

```
DiskInsertsLabel.Caption :=
IntToStr(MakeDictionary1.DiskInserts);
```

See also

BlobBytesRead, BlobBytesWritten, CacheEdits, CacheInserts, CacheReads, DiskDeletes, DiskEdits, DiskReads, ResetStats

DiskReads property

Applies to

TSearchDictionary, TUpdateDictionary

Declaration

```
property DiskReads: LongInt;
```

Returns the number of WordsTable records read from disk.

Run-time and read only.

Example

```
BytesReadLabel.Caption := IntToStr(SearchDictionary1.DiskReads);
```

See also

BlobBytesRead, BlobBytesWritten, CacheEdits, CacheInserts, CacheReads, DiskDeletes, DiskEdits, DiskInserts, ResetStats

EDictionary object

Declaration

```
EDictionary = class(EStException);

EStException = class(Exception)
               protected
              FErrorCode: LongInt;
               public
                property ErrorCode: LongInt
                 read FErrorCode write FErrorCode;
               end;
```

Most errors returned by Rubicon are EDictionary errors.   However, the routines in the taTools unit return EStContainerError errors which are also derived from EStException.

ErrorPos property

Applies to

TSearchDictionary

Declaration

```
property ErrorPos: Integer;
```

When SearchLogic is slExpression and there is an error in the syntax, ErrorPos returns the approximate position of the error in SearchFor.   If there is no error, ErrorPos is 0.

Run-time and read only.

Example

```
if SearchDictionary1.ErrorPos > 0 then { do something } ;
```

See also

SearchFor, SearchLogic

Execute method

Applies to

TMakeDictionary

Declaration

```
procedure Execute;
```

Builds a new WordsTable based on the contents of DataSource. DataSource.DataSet should not be updated during the build process (it can be a ReadOnly table).

The fields added to the dictionary are determined by the DataSource.DataSet.Fields, DataTypes/FieldTypes and FieldNames properties.

For string and memo fields, words are parsed using WordDelims.

Numeric, boolean, date, and time fields are added directly to the dictionary.

For Blob fields other than memo fields, an OnProcessField event handler must exist. This event can also be used to customize the handling of standard field types. In 16 bit applications, Blob fields are limited to 64k (an OnProcessField event may be used to work around this).

The execution process consists of two phases: Phase one reads each record in the DataSource.DataSet and builds a list of words and their locations in memory. During this phase, memory consumption grows. Phase two writes the in memory data to the WordsTable and releases memory used in phase one.

Resource usage can be controlled by setting FileCompression and MemoryLimit.

Use the OnPhaseOne and OnPhaseTwo events to monitor the progress of the process. The properties BlobBytesWritten, CacheCount, DiskInserts, MaxMemUsed, MemCompression, MemoryUsage, RecordNo, State are useful indicators of execution. These events may also be used to abort the process.

The number of records processed can be limited by using RecordLimit.

Example

```
Screen.Cursor := crHourGlass;
with MakeDictionary1 do
 try
  MemoryLimit := 8 * 1048575 {2^20 - 1};
  MinWordLen  := 3;
  IndexMode   := imOrdinalIndex;
  FieldTypes  := [ftString..ftDateTime,ftMemo];
  Execute
 finally
  Screen.Cursor := crDefault;
 end;
```

See also

CacheCount, DataTypes, FileCompression, MaxMemUsed, MemCompression, MemoryUsage, MemoryLimit, OnProcessField, OnPhaseOne, OnPhaseTwo, RecordLimit, DataSource, RecordNo, State, WordDelims, WordsTable

Execute method

Applies to

TSearchDictionary

Declaration

```
procedure Execute;
```

Performs the search based on the values of the SearchFor, SearchLogic, and SearchMode properties.

Example

```
with SearchDictionary1 do
 begin
  SearchFor := 'Borland';
  SearchLogic := slAnd;
  SearchMode := smSearch;
  Execute
 end;
```

See also

NarrowSearch, Search, Search Examples, SearchFor, SearchLogic, SearchMode, WidenSearch

Expanded property

Applies to

TMakeProgress, TUpdateStats

Declaration

```
property Expanded: Boolean;
```

Determines whether the form is opened with an expanded view.

Default is False.

Example

```
MakeProgress1.Expanded := True;
```

See also

Panels

FieldNames property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property FieldNames: TStrings;
```

Before the contents of a field are added to the dictionary or are accessed as part of a search, three checks are made: first, the field must be a member of DataSource.DataSet.Fields; second, the DataType of the field is checked to see if it is in FieldTypes; and, third, if there are entries in the FieldNames list, the FieldName of the field is checked against a list of FieldNames.

For DLL applications, all fields and all field types are included in the DataSource.DataSet.Fields and FieldTypes properties, so use the FieldNames property to select the fields for inclusion in the WordsTable.

Example

```
MakeDictionary1.FieldNames := MyListOfFields;
```

See also

DataTypes, FieldTypes

FieldTypes property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property FieldTypes: TFieldTypes;
```

Before the contents of a field are added to the dictionary or are accessed as part of a search, three checks are made: first, the field must be a member of DataSource.DataSet.Fields; second, the DataType of the field is checked to see if it is in FieldTypes; and, third, if there are entries in the FieldNames list, the FieldName is checked against a list of FieldNames.

This property is not published because the size of the set is too large for the property editor to handle. Use DataTypes in the property editor to set FieldTypes of ftString..ftMemo. The DataTypes property cannot be used to set ftUnknown, ftGraphic, ftFmtMemo, ftParadoxOle, ftDBaseOle, and ftTypedBinary in the IDE or at run time (you must use FieldTypes instead).

Default is [ftString, ftMemo] for Delphi, all field types for DLL applications. Run-time.

Example

```
MakeDictionary1.FieldTypes := [ftString..ftDateTime,ftMemo];
```

See also

DataTypes, FieldNames

FileCompression property

Applies to

TMakeDictionary, TUpdateDictionary

Declaration

```
property FileCompression: Boolean;
```

This option compresses information inside the dictionary.   This option should be left on unless the table resides on a compressed disk drive that offers superior compression to the build-in compression routines (this would be determined by comparing the file size with and without the file compression option enabled).   The amount of blob data compression will vary with the frequency that a word appears in multiple records (the more often it shows up, the less compression achieved), but compression rates of 95% or more are not unusual.

Do not change the value of FileCompression during a dictionary build.

Default is True.

Example

```
MakeDictionary1.FileCompression := True;
```

See also

MemCompression

FindXxxx methods

Applies to

TSearchDictionary

Declaration

```
function FindFirst: Boolean;
function FindLast: Boolean;
function FindNext: Boolean;
function FindPrior: Boolean;
```

A set of routines that may be called after a search has been conducted to move the cursor in the DataSource to the first, last, next, or prior location of a record that matched the search criteria.

First, last, next, and prior are all relative to the IndexMode used to make the WordsTable. Thus if an imSeqNo IndexMode was used to make the WordsTable, but the table is open on a secondary index, calls to FindNext will find the next SeqNo, but this may not be the 'next' record in the secondary index.

Example

```
with SearchDictionary1 do
 if FindFirst then
  repeat
   { do something }
  until not FindNext;
```

See also

MatchBits

FlushCache method

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
procedure FlushCache;
```

Forces any unwritten records in the memory cache to be written to disk and releases the associated memory. Most useful for applications that are performing searches or updates. Should not be used by TMakeDictionary since it manages its own cache.

Example

```
UpdateDictionary1.FlushCache;
```

See also

CacheEdits, CacheInserts, DelayedWrites, WriteCache

Form property

Applies to

TMakeProgress, TUpdateStats

Declaration

```
property TMakeProgress.Form: TMakeProgressForm;
property TUpdateStats.Form: TUpdateStatsForm;
```

Use this property to access the form.

Run-time.

Example

```
with MakeProgress1 do
 if (Form <> nil) and
    (Form.WindowState = wsMinimized) then
  Form.WindowState := wsNormal;
```

See also

Expanded, Panels

IndexFieldName property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property IndexFieldName: string;
```

This property is useful when using an imOrdinalIndex IndexMode and the DataSource is open on an index another index.   If imOrdinalIndex is not being used, this property is ignored.

If no IndexFieldName is specified, all the components assume the DataSource is open on the index which contains the imOrdinalIndex field.   If the table is not open on this index, use the IndexFieldName to specify the correct field.

If StrictChecking is True, checks are performed to confirm that the IndexFieldName is a single field unique index and that it's DataType is ftSmallInt, ftWord, or ftInteger.   When StrictChecking is False, the only check performed is that the IndexFieldName is a defined field in the table.   In addition, the value of StrictChecking affects how many field(s) are displayed in the Delphi Object Inspector.

For TSearchDictionary FindXxxx routines and for searches that require reading the source table (SubFieldNames, slNear and slPhrase SearchLogic), the use of IndexFieldName will require that the Delphi component (but not the DLL) temporarily switch indexes.   There is some overhead associated with switching indexes, so the use of IndexFieldName may not be appropriate, particularly for SQL tables.   You may find it faster to simple devote another TTable open on the ordinal index to the TSearchDictionary.

In DLL applications, IndexFieldName must be set after setting the DataSource.   The DLL always opens DataSource on the IndexFieldName index.

Example

```
UpdateDictionary1.IndexFieldName = 'CustNo';
```

See also

IndexMode, StrictChecking

IndexMode property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property IndexMode: TIndexMode;
```

There are four available Index Modes:

imOrdinalIndex: Best choice when a primary or unique secondary key consisting of a single ordinal field is available. The table must be open on this index unless IndexFieldName is specified. When StrictChecking is disabled, this mode may be used with floating point fields as long as there are no fractional values (this is not checked). This option is available for all table types, scales easily to SQL, and is most compatible with dynamic updating. For best results, the index value should be sequential or nearly sequential (e.g. CustomerNo should be 1001, 1002, 1003 rather than 1010, 1020, 1030).

imRecordNo: Available only for dBase files. Use this option if you cannot use an imOrdinalIndex. The DataSource.DataSet must not have any range limitations on it. Only compatible with dynamic updating under certain restrictive conditions (only appends and edits allowed, no deletions or insertions permitted).

imSeqNo: Available only for Paradox files. Same restrictions as imRecordNo. During searches, the DataSource.DataSet must use the same index used when the dictionary was created.

imNone: Table does not have a compatible IndexMode.

If your table does not have an ordinal index or you have an ordinal index but there are large increments between index values, it is recommended that you add an additional field and create a unique secondary index and populate the field with sequential index values (gaps due to deleted records are not a problem).

For a given DataSource and resulting WordsTable, the IndexMode used in TSearchDictionary must be the same as the one used in TMakeDictionary (this is not checked).

Example

```
MakeDictionary1.IndexMode := imOrdinalIndex;
```

See also

Index Modes, IndexFieldName, MinOrdIndex, TIndexMode

IndexRange property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property IndexRange: LongInt;
```

Difference between the minimum index value and the maximum index value.   If MinOrdIndex is set, it replaces the minimum index value.   If SourceRange is set, IndexRange is equal to the SourceRange.

Run-time and read only.

See also

IndexMode, MinOrdIndex

KeyViolName property

Applies to

TMakeDictionary

Declaration

```
property KeyViolName: string;
```

During phase one of a dictionary build, the words are stored in memory (without being truncated if the length of the word exceeds WordFieldSize). During phase two, these words are written to the WordsTable. If the WordFieldSize property is set too low, it is possible that key violations will result due to the truncation of trailing characters. If a key violation results, the word is written to a Paradox table with the name KeyViolName. The only valid extension is '.db', and this is optional.

Examples

```
MakeDictionary1.KeyViolName := 'd:\project\keyviol';
MakeDictionary1.KeyViolName := ':rubicon:keyviol.db';
```

See also

DataSource, MatchTable, WordsTable

LikeFieldSize property

Applies to

TMakeDictionary

Declaration

```
property LikeFieldSize: Integer;
```

Determines the size (or length) of the Likeness field in the WordsTable. Setting LikeFieldSize to zero removes the Likeness field from the WordsTable after the next Execute and disables slLike SearchLogic.

Default is 5.

Example

```
MakeDictionary1.LikeFieldSize := 8;
```

See also

Likeness

Likeness property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property Likeness: TStringFunc;
```

Function used to convert words to a 'like' equivalent string.   Two or more words that have the same 'like' string will evaluate as equivalent when using slLike SearchLogic. The Likeness function must return a string of less than or equal to LikeFieldSize characters.

Default is Soundex.   Run-time.

Example

```
MakeDictionary1.Likeness := Metaphone;
```

See also

Builder, LikeFieldSize, SearchLogic, Soundex

LoadOmitsFromTable method

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
procedure LoadOmitsFromTable(Table: TTable; FieldName: string);
```

Use this method to fill the OmitList with words from the FieldName field from Table. The   OmitList is limited to MaxOmits words.

Example

```
MakeDictionary1.LoadOmitsFromTable(OmitsTable, 'OmitWord');
```

See also

OmitList

Maker property

Applies to

TMakeProgress

Declaration

```
property Maker: TMakeDictionary;
```

The Maker is the TMakeDictionary whose progress is displayed in the form.

Example

```
MakeProgress1.Maker := MakeDictionary2;
```

MakeWordDelims function

Declaration

```
function MakeWordDelims(WordCharSet: TCharSet) : string;
```

Given the characters that make up a word, WordCharSet, returns a string of WordDelims. The example sets WordDelims to all characters except 'A'..'Z' and 'a'..'z'.

Example

```
MakeDictionary1.WordDelims := MakeWordDelims(['A..'Z','a'..'z']);
```

See also

WordDelims

MatchBits property

Applies to

TSearchDictionary

Declaration

```
property MatchBits: TtaBits;
```

MatchBits is a bit set class derived from TurboPower's TStBits, part of SysTools. Because other routines encapsulate the most common TtaBits methods (see See Also), an application will rarely have to rely on this property directly.

Example

```
with SearchDictionary1 do
 begin
  Location := MatchBits.FirstSet;
  while Location <> -1 do
   begin
    case IndexMode of
     imOrdinalIndex : TTable(DataSource.DataSet).FindKey(
                        [Location + MinIndex]);
     imRecordNo     : SetToRecordNo(DataSource.DataSet,Location +
1);
     imSeqNo        : SetToSeqNo(DataSource.DataSet,Location + 1)
    end;
    { do something with the record }
    Location := MatchBits.NextSet(Location)
   end
 end;
```

See also

FindFirst, FindLast, FindNext, FindPrior, Matches, MinIndex

MatchCount property

Applies to

TSearchDictionary

Declaration

```
property MatchCount: LongInt;
```

Number of words that match the search criteria of the latest Search or NarrowSearch. Is not cumulative over successive calls to NarrowSearch.

Run-time and read only.

Example

```
if SearchDictionary1.MatchCount >= 20 then
 MessageDlg('Please narrow search further',mtInformation,
[mbOk],0);
```

See also

MatchingWords, NarrowSearch, RecordCount, Search

Matches method

Applies to

TSearchDictionary

Declaration

```
function Matches: Boolean;
```

Returns True if the current record of the DataSource matches the search criteria. Returns False the current record does not match the search criteria, the DataSource is nil, the DataSource.DataSet is closed, or a search has not been conducted. It may be necessary to call UpdateCursorPos before calling Matches.

Example

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet;
                              var Accept: Boolean);
begin
 { Assumes IndexMode = imOrdinalIndex }
 Accept := SearchDictionary1.Matches
end;
```

See also

NarrowSearch, Search

MatchingWords method

Applies to

TSearchDictionary

Declaration

```
procedure MatchingWords(List: TStrings);
```

Returns a list of words matched during the search. If only slAnd, slNear, and slPhrase SearchLogic are used, than all the words in the list will be in each matching record. If other types of SearchLogic are used, then only a subset of words in the list will be in each matching record (e.g. words matched using slNot will never appear in the matching records)

Example

```
with SearchDictionary1 do
 begin
  SearchLogic := slOr;
  Search('Win*');
  MatchingWords(ListBox1.Items);  { Filled with Win95, Windows,
WinNT }
 end;
```

See also

SearchFor, SearchMode, SearchLogic

MatchTable property

Applies to

TSearchDictionary

Declaration

```
property MatchTable: TTable;
```

After a search, records matching the search criteria are copied from the DataSource into this table. The number of records copied into the MatchTable is the lesser of RecordCount or RecordLimit. The order of the records in the MatchTable is controlled by RankMode.

MatchTable is optional, it may be left unassigned.

Example

```
SearchDirectory1.MatchTable := Table1;
```

See also

CreateMatchTable, RankMode, RecordCount, RecordLimit

MaxMemUsed property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property MaxMemUsed: LongInt;
```

The maximum value of MemoryUsage.

Run-time and read only.

Example

```
MaxMemUsedLabel.Caption := IntToStr(MakeDictionary1.MaxMemUsed);
```

See also

MemoryLimit, MemoryUsage, ResetStats

MemCompression property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property MemCompression: Boolean;
```

Indicates whether memory compression is being used.

Run-time and read only.

See also

FileCompression

MemoryLimit property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property MemoryLimit: LongInt;
```

For TMakeDictionary, if MemoryUsage exceeds MemoryLimit, then MemCompression is turned on and the least frequently used memory data structures are compressed. If all data structures in memory have been compressed, the build process will continue to consume memory beyond this limit.

For TSearchDictionary and TUpdateDictionary, behavior is the same except that when compression is on and MemoryLimit is exceeded, indexes are removed from memory.

If the alternative memory manager is being used, the portion of MemoryUsage that represents memory waiting to be reused is not included in the test against MemoryLimit. See Delphi 2.0 Memory Fragmentation for more details.

Default for TMakeDictionary is 4,000,000 with a minimum of 1,000,000.

Default for TSearchDictionary and TUpdateDictionary is 0. Note that even with MemoryLimit set to zero, there still will be several thousand of bytes reported by MemoryUsage. This represents memory used by non-cache data structures. Therefore, if you want to set aside 50k for cache usage, set MemoryLimit to 55k (approximate, will vary with the IndexRange of the DataSource).

Example

```
MakeDictionary1.MemoryLimit := 20000000;
```

See also

MemCompression, MemoryUsage

MemoryUsage property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property MemoryUsage: LongInt;
```

The current amount of memory used by the component. The value excludes some data structures, so the actual memory usage is somewhat higher.

Run-time and read only.

Example

```
MemoryUsageLabel.Caption :=
IntToStr(MakeDictionary1.MemoryUsage);
```

See also

MaxMemUsed, MemoryLimit

MinIndex property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property MinIndex: LongInt;
```

When working with the MatchBits methods FirstSet, NextSet, PrevSet, LastSet (and the equivalent XxxxClear routines), MinIndex should be added to the return value in order to identify the correct location in the table.

Run-time and read only.

Example

```
with SearchDirectory1 do
 begin
  Location := MatchBits.FirstSet;
  if (Location <> -1) and (IndexMode = imOrdinalIndex) then
   TTable(DataSource.DataSet).FindKey([Location + MinIndex])
 end;
```

See also

MatchBits

MinOrdIndex property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property MinOrdIndex: LongInt;
```

A non-zero value causes MinIndex to be set to MinOrdIndex when IndexMode is imOrdinalIndex. This property is useful when updating imOrdinalIndex tables and you wish to be able to add records before the first record in the table.

This property must be set when the dictionary is first built and the same value must be used for all subsequent updates and searches.

Changing MinOrdIndex after the build and then updating the WordsTable will corrupt the locations within the dictionary. An incorrect MinOrdIndex used in searches will simply result in meaningless search results (the underlying WordsTable are not corrupted).

For SQL tables, using MinOrdIndex eliminates a call to DataSource.DataSet.First and may speed up some operations. When used in conjunction with SourceRange, MinOrdIndex plus SourceRange must be greater than or equal to the maximum value of the index. This is not checked.

Default is 0.

Example

```
MakeDictionary1.MinOrdIndex := 100;
```

See also

IndexMode

MinWordLen property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property MinWordLen: Integer;
```

Words with a length less than the minimum word length will not be included in the dictionary or a search (unless one of the characters is AnyChar).   A minimum word length of three would therefore exclude words like 'a', 'of', 'we', etc., which are generally not very useful in a search.   Note that it may exclude some useful state (CA, NY), company (3M, TI), and other (PC) abbreviations and words like 'go' and 'hi'.

Valid values are 1..10.   Default is 1.

Example

```
MakeDictionary1.MinWordLen := 3;
```

See also

DataTypes, FieldNames, FieldTypes, OmitList

NarrowSearch method

Applies to

TSearchDictionary

Declaration

```
procedure NarrowSearch(S: string);
```

NarrowSearch is a shorthand equivalent to:

```
     SearchDictionary1.SearchFor := S;
     SearchDictionary1.SearchMode := smNarrow;
     SearchDictionary1.Execute;
```

Example

```
with SearchDirectory1 do
 begin
  SearchLogic := slAnd;
  Search('delphi paradox');
  SearchLogic := slNot;
  NarrowSearch('access')
 end;
```

See also

Execute, Search, SearchFor, SearchLogic, SearchMode, WidenSearch

NearWord property

Applies to

TSearchDictionary

Declaration

```
property NearWord: Integer;
```

The parameter used by slNear SearchLogic to determine whether two words are near each other.

Default is 8.

Example

```
with SearchDirectory1 do
 begin
  NearWord := 10;
  SearchLogic := slNear;
  Search('delphi component');
 end;
```

See also

SearchLogic

OmitList property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
procedure OmitList: TStrings;
```

Assign the list of words to be omitted from the WordsTable. These words need not be in upper case. One word per line. No duplicate words permitted.

Example

```
TUpdateDictionary.OmitList := MyOmitList;
```

See also

LoadOmitsFromTable, MinWordLen, WordsTable

OneChar property

Applies to

TSearchDictionary

Declaration

```
property OneChar: Char;
```

The wildcard that matches any series of characters. Must be different from AnyChar.

Default is '?'.

Example

```
SearchDictionary1.OneChar := '#';
```

See also

AnyChar

OnPhaseOne property

Applies to

TMakeDictionary

Declaration

```
property OnPhaseOne: TNotifyEvent;
```

Provides notification during phase one of the execute process.

Before processing of phase one begins, OnPhaseOne is called and State = [dsPhaseOne, dsStart]. When phase one processing is completed, OnPhaseOne is called with State = [dsPhaseOne, dsDone] (may also include dsAbort).

During phase one, RecordNo refers to the number of records processed. Phase one processes all the records (or up to RecordLimit) in DataSource, so OnPhaseOne will be called approximately DataSource.DataSet.RecordCount + 2 times (RecordCount is approximate for some table types).

Before indexes are compressed, the event is called with State = [dsPhaseOne, dsCompress, dsStart]. When compression is completed, the event is called again with dsStart replaced with dsDone.

To abort processing, set State := State + [dsAbort].

Example

```
procedure TForm1.MakeDictionary1PhaseOne(Sender: TObject);
begin
 with TMakeDictionary(Sender),PhaseForm do
  begin
   if State = [dsPhaseOne, dsStart] then
    begin
     Gauge.MinValue := 0;
     { RecordCount is approximate for some table types! }
     Gauge.MaxValue := DataSource1.DataSet.RecordCount;
     Gauge.Progress := 0;
     Caption := 'Phase One';
     DBSizeLabel.Caption := '';
     MBSizeLabel.Caption := '';
     CompressionLabel.Caption := '';
    end;
   Gauge.Progress := RecordNo;
   MemUsedLabel.Caption :=
     Format('%10.0n',[MemoryUsage + 0.001]);
   WordCountLabel.Caption :=
     Format('%10.0n',[CacheCount - OmitList.Count + 0.001]);
   Application.ProcessMessages;
   if not PhaseForm.Visible then State := State + [dsAbort]
  end
```

```
end;
```

See also

OnPhaseTwo

OnPhaseTwo property

Applies to

TMakeDictionary

Declaration

```
property OnPhaseTwo: TNotifyEvent;
```

Provides notification during phase two of the execute process.

Before processing of phase two begins, OnPhaseTwo is called and State = [dsPhaseTwo, dsStart]. When phase two processing is completed, OnPhaseTwo is called with State = [dsPhaseTwo, dsDone] (may also include dsAbort).

During phase two, DiskInserts refers to the number of records written to the WordsTable. Phase two will process CacheCount records, so OnPhaseTwo will be called CacheCount + 2 times. (Note: the value of CacheCount cited here is its value at the end of phase one or at the start of phase two. During the processing of phase one CacheCount is increasing, while in phase two it is decreasing).

To abort processing, set State := State + [dsAbort].

Example

```
procedure TForm1.MakeDictionary1PhaseTwo(Sender: TObject);
begin
 with TMakeDictionary(Sender),PhaseForm do
  begin
   if dsStart in State then
    begin
     Caption := 'Phase Two';
     Gauge.MinValue := 0;
     Gauge.MaxValue := CacheCount;
     Gauge.Progress := 0;
    end;
   Gauge.Progress := DiskInserts;
   MemUsedLabel.Caption :=
     Format('%10.0n',[MemoryUsage + 0.001]);
   Application.ProcessMessages;
   if not PhaseForm.Visible then State := State + [dsAbort]
  end
end;
```

See also

OnPhaseOne

OnProcessField property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property OnProcessField: TProcessFieldEvent;
```

Allows for custom field handling.

Example

```
UpdateDictionary1.OnProcessField :=
UpdateDictionary1ProcessField;
```

See also

ProcessField, TProcessFieldEvent

OnSearch property

Applies to

TSearchDictionary

Declaration

```
property OnSearch: TNotifyEvent;
```

Most wildcard searches will require that all or part of WordsTable to be scanned (e.g. '*fish' would require the entire table to be scanned, whereas 'tuna*' would require only those words beginning with 'tuna' to be scanned).   This event is called periodically during the scan to give the application an opportunity to abort the scan by setting adding [dsAbort] to State.

Example

```
procedure TMainForm.SearchDictionary1Search(Sender: TObject);
begin
 Application.ProcessMessages;
 if not FContinue then
  with SearchDictionary1 do
   State := State + [dsAbort]
end;
```

See also

NarrowSearch, Search

OnWrite property

Applies to

TUpdateDictionary

Declaration

```
property OnWrite: TNotifyEvent;
```

When DelayedWrites is True, TUpdateDictionary holds as many words as possible in memory. When the MemoryLimit is reached, least recently used words are compressed. When words can no longer be compressed, least recently used words are written to disk and removed from the cache. This process can stall the application, but the OnWrite event can be used to continue processing other tasks.

OnWrite is also called during a WriteCache or FlushCache. In these cases, the process may be aborted if your application has directly initiated the process and you have included dsMayAbort in State. OnWrite will not abort if dsMayAbort is not in State. You would then resume the process later.

FlushCache is also called during Free, when the DataSource or Builder properties are set, and when the DataSource.DataSet becomes active. These processes must not be aborted.

Use the CacheCount property to determine the number of words in the cache. Use the UnwrittenWords property to determine the number of unwritten words in the cached. OnWrite will be called UnwrittenWords times. You should not call UnwrittenWords during each OnWrite event since UnwrittenWords has to iterate through the entire cache each time. Instead, save the value of UnwrittenWords before a call to WriteCache or FlushCache and decrement it during each call to OnWrite.

Example

```
. . .
with UpdateDictionary1 do
 begin
  State := State + [dsMayAbort];
  FlushCache
 end;
. . .
procedure TMainForm.UpdateDictionary1Write(Sender: TObject);
begin
 Application.ProcessMessages;
 { do other processing }
 if dsCompress in UpdateDictionary1.State then
  { May not abort }
 else
  { FlushCache or WriteCache is being processed }
  if condition then
   UpdateDictionary1.State := [dsAbort] + UpdateDictionary1.State
```

```
end;
```

See also

CacheCount, DelayedWrites, FlushCache, MemoryLimit, UnwrittenWords, WriteCache

Panels property

Applies to

TUpdateStats

Declaration

```
property Panels: TStatPanels;
```

Determines which panels are visible.

Default is [spCache, spMemory].

Example

```
UpdateStats.Panels := [spCache..spLRU];
```

ProcessField method

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
procedure ProcessField(Field: TField; Location: LongInt);
```

ProcessField may only be called from within an OnProcessField event handler.

Example

```
procedure TForm1.MakeDictionary1ProcessField(Sender: TObject;
                       Field: TField; Location: LongInt);
begin
 with TMakeDictionary(Sender) do
  if (Field.FieldName = 'Company') and
     (Field.AsString = 'IBM') then
   begin
    ProcessWord('IBM',Location);
    ProcessWord('International',Location);
    ProcessWord('Business',Location);
    ProcessWord('Machine',Location);
   end
  else ProcessField(Field,Location)
end;
```

See also

ProcessList, ProcessRecord, ProcessWord, OnProcessField

ProcessList method

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
procedure ProcessList(List: TStrings; Location: LongInt;
                      Parse: Boolean);
```

The strings contained in List are added to the WordsTable or processed in a search. If Parse is True, then each string is broken down into single words using WordDelims, otherwise each entry in the List should be an individual word.

The strings or words contained in List should be in the same order as they appear in the field. Do not attempt to eliminate duplicate words.

ProcessList may only be called from within an OnProcessField event handler.

Example

```
procedure TForm1.MakeDictionary1ProcessField(Sender: TObject;
                       Field: TField; Location: LongInt);
var List: TStrings;
begin
 with TMakeDictionary(Sender) do
  if (Field.FieldName = 'Company') and
     (Field.AsString = 'IBM') then
   begin
    List := TStringList.Create;
    try
     List.Add('IBM International Business Machines');
     ProcessList(List,Location,True)
    finally
     List.Free
    end
   end
  else ProcessField(Field,Location)
end;
```

See also

ProcessField, ProcessPChar, ProcessRecord, ProcessWord, OnProcessField

ProcessPChar method

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
procedure ProcessPChar(S: PChar; Location: LongInt);
```

The words contained in S are added to the WordsTable or processed in a search.   May only be called from within an OnProcessField event handler.

See also

OnProcessField, ProcessField, ProcessList, ProcessWord

ProcessRecord method

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
procedure ProcessRecord(Location: LongInt);
```

A virtual procedure that may be overridden in order to provide record level control over a search or controlling additions to the dictionary.

See also

OnProcessField, ProcessField, ProcessList, ProcessWord

ProcessWord method

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
procedure ProcessWord(S: string; Location: LongInt);
```

S is added to the WordsTable or processed in a search. No parsing is performed. ProcessWord may only be called from within an OnProcessField event handler.

Call ProcessWord for each word contained in the field and in the same order as they appear in the field. Do not attempt to eliminate duplicate words.

Example

```
procedure TForm1.MakeDictionary1ProcessField(Sender: TObject;
                       Field: TField; Location: LongInt);
begin
 with TMakeDictionary(Sender) do
  if (Field.FieldName = 'Company') and
     (Field.AsString = 'IBM') then
   begin
   begin
    ProcessWord('IBM',Location);
    ProcessWord('International',Location);
    ProcessWord('Business',Location);
    ProcessWord('Machine',Location);
   end
  else ProcessField(Field,Location)
end;
```

See also

OnProcessField, ProcessField, ProcessList, ProcessRecord

RankMode property

Applies to

TSearchDictionary

Declaration

```
property RankMode: TRankMode;
```

Determines how records are ordered in the MatchTable or by CreateMatchTable. There are three RankModes: rmNone, rmCount, and rmPercent. rmNone leaves the records in index or natural order as determined by the IndexMode used to make the WordsTable. Since the DataSource may be open on another index, the ordering of records may not be consistent with other views of the table. rmCount adds a Rank field to the table that contains a count of the matching words. rmPercent is like rmCount, except it uses a 100 scale. Ranking, if any, occurs after a match table has been created.

For searches using slNear or slPhrase SearchLogic, the ranking process only scores the matching words in the record, not whether the words are near or in sequence of each other.

The Rank field will be in the rightmost column. If a column already exists with a name of Rank, it will alternatively be named Ranking, Score, or Scoring.

Be aware that if a search locates 100 records and a match table is created with a limit of 50 records, only those records (the first 50) are ranked. There may be higher ranking records in the remaining 50 records.

Default is rmNone.

Example

```
with SearchDictionary1 do
 begin
  RankMode := rmCount;
  CreateMatchTable(Table1);
 end;
```

See also

CreateMatchTable, MatchTable

RecordCount property

Applies to

TSearchDictionary

Declaration

```
property RecordCount: LongInt;
```

Number of records matching the search criteria.   Read only.

Example

```
with SearchDictionary1 do
 if RecordCount < 100 then CreateMatchTable(Table1);
```

See also

CacheCount, NarrowSearch, Search, SearchLogic, WordDelims

RecordLimit property

Applies to

TMakeDictionary, TSearchDictionary

Declaration

```
property RecordLimit: LongInt;
```

For TMakeDictionary, useful for limiting the build to the first RecordLimit records for testing purposes.   A zero value indicates no record limit.

For TSearchDictionary, if MatchTable is assigned and RecordLimit is less than or equal to RecordCount, then the MatchTable is automatically filled with records matching the search criteria up to RecordLimit records.

Default for TMakeDictionary is 0 and for TSearchDictionary is 25.

Example

```
MakeDictionary1.RecordLimit := 1000;
```

See also

Execute, MemoryLimit

RecordNo property

Applies to

TMakeDictionary

Declaration

```
property RecordNo: LongInt;
```

During phase one of execution, RecordNo is the record number being processed (e.g. the fifth record being processed).

Run-time and read only.

Example

```
procedure TMainForm.MakeDictionary1PhaseOne(Sender: TObject);
begin
 with TMakeDictionary(Sender),PhaseForm do
  begin
   if dsStart in State then
    begin
     Gauge.MinValue := 0;
     { RecordCount is approximate for some table types! }
     Gauge.MaxValue := DataSource1.DataSet.RecordCount;
     Gauge.Progress := 0
    end;
   Gauge.Progress := RecordNo
  end
end;
```

See also

Execute, OnPhaseOne, OnPhaseTwo

RefreshInterval property

Applies to

TMakeProgress, TUpdateStats

Declaration

```
property RefreshInterval: LongInt;
```

The data displayed on the form is updated every RefreshInterval ticks. Setting RefreshInterval to zero results in the data being updated every time the TMakeDictionary or TUpdateDictionary processes a record.

Default is 500 (0.5 seconds).

Example

```
UpdateStats1.RefreshInterval := 0;
```

ResetStats method

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
procedure ResetStats;
```

Resets the following properties to zero: BlobBytesRead, BlobBytesWritten, CacheEdits, CacheInserts, CacheReads, DiskDeletes, DiskEdits, DiskInserts, DiskReads, and MaxMemoryUsed.

Example

```
with UpdateDictionary1 do
 if CacheEdits + CacheInserts > 0 then
  begin
   WriteCache;
   ResetStats
  end;
```

See also

BlobBytesRead, BlobBytesWritten, CacheEdits, CacheInserts, CacheReads, DiskDeletes, DiskEdits, DiskInserts, DiskReads

Search method

Applies to

TSearchDictionary

Declaration

```
procedure Search(S: string);
```

Search is a shorthand equivalent to:

```
     SearchDictionary1.SearchFor := S;
     SearchDictionary1.SearchMode := smSearch;
     SearchDictionary1.Execute;
```

To narrow an existing search, use NarrowSearch.

Example

```
with SearchDictionary1 do
 begin
  SearchLogic := slOr;
  Search('borland delphi paradox')
 end;
```

See also

Execute, NarrowSearch, SearchFor, SearchMode, WidenSearch

SearchFor property

Applies to

TSearchDictionary

Declaration

```
property SearchFor: string;
```

If SearchLogic is slExpression, then SearchFor contains the expression to be evaluated, otherwise it contains the words to search for.

Example

```
with SearchDictionary1 do
 begin
  SearchFor := Edit1.Text;
  SearchLogic := slAnd;
  SearchMode := smSearch;
  Execute
 end;
```

See also

ErrorPos, NarrowSearch, Search, *Search Examples*, Search Strategies, TSearchLogic

SearchLogic property

Applies to

TSearchDictionary

Declaration

```
property SearchLogic: TSearchLogic;
```

There are seven search types: slAnd, slPhrase, slLike, slNear, slOr, slNot, and slExpression.

The three most common are: slAnd which searches for records that contain all instances of the words in the SearchFor property; slOr which searches for records that contain at least one instance of the words in the SearchFor property; and slNot which selects all records that do not contain instances of the words in the SearchFor property.

When IndexMode is imOrdinalIndex, slNot should only be used to narrow an existing search, not to start a new search (see discussion in Search Strategies).

slLike searches for words that evaluate as the same using the Likeness function. Wildcards are ignored.

slNear searches for two words that are within NearWord words of one another in a field(s). If the number of words in the search is not two, an error is raised.

slPhrase searches for words in a specific order of appearance in the field(s).

slExpression enables expression evaluation using AND, OR, NOT, LIKE, NEAR, "quoted phrase searches", and parentheses. For more information on this type of search, see Expression Evaluation.

Example

```
with SearchDictionary1 do
 begin
  SearchFor := Edit1.Text;
  SearchLogic := slOr;
  SearchMode := smSearch;
  Execute
 end;
```

See also

ErrorPos, NarrowSearch, Search, *Search Examples*, Search Strategies, TSearchLogic

SearchMode property

Applies to

TSearchDictionary

Declaration

```
property SearchMode: TSearchMode;
```

There are three search modes: smSearch, smNarrow, smWiden.

smSearch conducts a new search when Execute is called.

smNarrow ANDs the contents of the current search with the previous search when Execute is called.

smWiden ORs the contents of the current search with the previous search when Execute is called.

Example

```
with SearchDictionary1 do
 begin
  SearchFor   := 'Borland';
  SearchLogic := slAnd;
  SearchMode  := smNarrow;
  Execute
 end;
```

See also

Execute, NarrowSearch, Search, SearchFor, SearchLogic, WidenSearch

Soundex function

Declaration

```
function Soundex(const S: string): string;
```

The standard Soundex function returns a four character string beginning with the first letter of S, followed by three numbers between 1 and 6 (e.g. 'B253'). This is a slightly modified version of the standard Soundex. Instead of returning a four character string, a five character string is returned with the first two characters being the first two letters of S and the remaining characters being the numbers 1..6 (e.g. 'BE253'). This modification improves its performance as a Likeness function.

If you have enabled the HaveSysTools compiler directive in TARUBICN.INC, Soundex will rely on SoundexS or SoundexL to perform most of the conversion. Generally, there is no functional difference between these routines. However, Soundex performs no case conversion since it assumes S is already in uppercase, while SoundexS and SoundexL perform AnsiUpperCase conversions. This would only become an issue if the HaveSysTools compiler directive were changed after a dictionary build since the version of Soundex used to create the dictionary and the version used to search it would be slightly different.

Example

```
SearchDictionary1.Likeness := Soundex;
```

See also

LikeFieldSize, Likeness

SourceRange property

Applies to

TSearchDictionary

Declaration

```
property SourceRange: LongInt;
```

Searches may be conducted without setting a DataSource by setting SourceRange to a value larger than the number of records (if IndexMode is imRecordNo or imSeqNo) in the DataSource or the difference between the highest and lowest index values (if IndexMode is imOrdinalIndex) of the DataSource.

Another way of determining the appropriate SourceRange is to use the value of IndexRange when DataSource is set (SourceRange is basically a substitute for IndexRange when there is no DataSource).

For SQL tables, using SourceRange eliminates a call to DataSource.DataSet.Last and may speed up some operations.   When used in conjunction with MinOrdIndex, MinOrdIndex plus SourceRange must be greater than or equal to the maximum value of the index.   This is not checked.

Run-time.

Example

```
SearchDictionary1.SourceRange := 100000;
```

See also

DataSource, IndexMode, IndexRange, MinOrdIndex

SourceReads property

Applies to

TSearchDictionary

Declaration

```
property SourceReads: LongInt;
```

Number of times the DataSource has been read. Reads occur when CreateMatchTable is called, a search uses SubFieldNames, or if SearchLogic is slNear or slPhrase. Use ResetStats to reset this indicator.

Run-time.

Example

```
SourceReadsLabel.Caption :=
IntToStr(SearchDictionary1.SourceReads);
```

See also

CreateMatchTable, DataSource, ResetStats, SearchLogic, SubFieldNames

State property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property State: TDictionaryStates;
```

The current state of the component.

There are four primary states:

- dsPhaseOne indicates phase one of TMakeDictionary.Execute
- dsPhaseTwo indicates phase two of TMakeDictionary.Execute
- dsUpdating indicates TUpdateDictionary is performing an update
- dsSearching indicates TSearchDictionary is performing a search

For all the primary states except dsUpdating, the State may be additionally qualified as dsStart or dsDone which are set at the beginning and ending of a process.

For TSearchDictionary, State may also be qualified by dsMatching, which is set during the creation of a match table, or by dsLocating, which is set during a search that requires the DataSource to be read (i.e. when SearchLogic is slNear and slPhrase).

An empty State indicates the component is idle.   A State of dsAbort indicates that the process was aborted.

The only time an application may change State is during OnPhaseOne, OnPhaseTwo, or OnSearch event, and then the only valid change is to add [dsAbort] to the State (see example in OnPhaseOne property).

Run-time.

Example

```
procedure TMainForm.MakeDictionary1PhaseOne(Sender: TObject);
begin
 with TMakeDictionary(Sender),PhaseForm do
  begin
   if dsStart in State then
    begin
     Gauge.MinValue := 0;
     { RecordCount is approximate for some table types! }
     Gauge.MaxValue := DataSource1.DataSet.RecordCount;
     Gauge.Progress := 0
    end;
   Gauge.Progress := RecordNo
  end
end;
State = libRubicon.getProperty(hMake, rblState)
```

See also

OnPhaseOne, OnPhaseTwo

StrictChecking property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property StrictChecking: Boolean;
```

When True, this only allows the imOrdinalIndex option to be used when the table is open on a single ordinal field primary key or unique secondary index.   When False, the restriction on the ordinal field is relaxed to include all other field types.   This assumes that the field will evaluate to a LongInt value.   Floating point fields will only work if there are no fractional values in the index (this is not checked).

StrictChecking also determines the level of checks performed on the IndexFieldName. If set to True, checks are performed to confirm that the IndexFieldName is a single field unique index and that it's DataType is ftSmallInt, ftWord, or ftInteger.   When StrictChecking is False, the only check performed is that the IndexFieldName is a defined field in the table.   The value of StrictChecking may affect how many field(s) are displayed in the Delphi Object Inspector for the IndexFieldName property.

Default is True.

Example

```
UpdateDictionary1.StrictChecking := False;
```

See also

IndexFieldName, IndexMode

SubFieldNames property

Applies to

TSearchDictionary

Declaration

```
property SubFieldNames: TStrings;
```

By default, all searches are conducted against all the fields included in the WordsTable. However, SubFieldNames can be used to search against a subset of fields (searches may not be conducted on excluded fields without rebuilding the WordsTable).

The words in the WordsTable are selected on the basis of available fields in the DataSource (as controlled by the field editor), the DataTypes/FieldTypes property, and the FieldNames property. When SubFieldNames is not empty, the search is further narrowed to the field names included in SubFieldNames.

Using SubFieldNames forces all searches to read the DataSource (and related DataSets), and therefore slows the search process. Before any reads take place, the words in the search and the SearchLogic are used to narrow the list of possible records. This means that fairly specific searches will only read a small number of records, and search performance will remain fast.

However, very broad searches, and especially searches containing slNot SearchLogic (which will be forced to read each record of the DataSource), will have their performance adversely affected. You may want to use a timer and the OnSearch event to abort the search after some period of time and advise the user to take a different approach to the search.

Example

```
SearchDictionary1.SubFieldNames := ListBox1.Items;
```

See also

DataTypes, FieldNames, FieldTypes

TDataType type

Declaration

```
TDataType =
(dtString,dtSmallInt,dtInteger,dtWord,dtBoolean,dtFloat,
             dtCurrency,dtBCD,dtDate,dtTime,dtDateTime,
           {$IFDEF Win32} dtAutoInc, {$ENDIF}
             dtBytes,dtVarBytes,dtBlob,dtMemo);
```

TDataType is a subset of TFieldType (ftUnknown, ftGraphic, ftFmtMemo, ftParadoxOle, ftDBaseOle, and ftTypedBinary are omitted). The only purpose of TDataType is to provide a set small enough to appear on the property editor (a set of TFieldType exceeds the 16 bit set limit of the property editor).

See also

DataTypes, FieldTypes, TFieldTypes

TDataTypes type

Declaration

```
TDataTypes = set of TDataType;
```

TDataType is a subset of TFieldTypes (ftUnknown, ftGraphic, ftFmtMemo, ftParadoxOle, ftDBaseOle, and ftTypedBinary are omitted). The only purpose of TDataTypes is to provide a set small enough to appear on the property editor (TFieldTypes exceeds the 16 bit set limit of the property editor).

See also

DataTypes, FieldTypes, TDataType, TFieldTypes

TDictionaryState type

Declaration

```
TDictionaryState = (dsPhaseOne, dsPhaseTwo, dsUpdating,
dsSearching,
               dsMatching, dsLocating, dsCompress, dsStart,
               dsDone, dsAbort, dsMayAbort);
```

State	Meaning
dsPhaseOne	Indicates TMakeDictionary is in phase one of execution
dsPhaseTwo	Indicates TMakeDictionary is in phase two of execution
dsUpdating	Indicates TUpdateDictionary is in the process of updating
dsSearching	Indicates TSearchDictionary is in the process of searching
dsMatching	Indicates TSearchDictionary is in the process of creating a match table
dsLocating	Indicates TSearchDictionary is in the process of locating words in the DataSource due to the use of SubFieldNames and/or slNear or slPhrase SearchLogic
dsCompress	Indicates that the cache is being compressed
dsStart	Used in conjunction with dsPhaseOne, dsPhaseTwo, dsSearching, and dsCompress to indicate the start of the process
dsDone	Same as dsStart, but indicates the end of the process
dsAbort	Setting this state aborts the process
dsMayAbort	Used by TUpdateDictionary to indicate that the OnWrite event may be aborted

See also

TDictionaryStates, State

TDictionaryStates type

Declaration

```
TDictionaryStates = set of TDictionaryState;
```

See also

TDictionaryState, State

TFieldTypes type

Declaration

```
TFieldTypes = set of TFieldType;
```

See also

DataTypes, FieldTypes, TDataType

TIndexMode type

Declaration

```
TIndexMode = (imOrdinalIndex, imRecordNo, imSeqNo, imNone);
```

See also

IndexMode, Index Modes

TMakeProgress component

TMakeProgress is a drop in form with will automatically configure itself to display the progress of a TMakeDictionary build.   Double or right click at run time on the form to set or hide build statistics.   The OnPhaseOne and OnPhaseTwo event properties of the TMakeDictionary it attaches to will not display anything, but these properties are actually set.

Properties include AutoClose (close automatically when done), Expanded (shows more statistics), Form (pointer to the form, run-time), Maker (source TMakeDictionary), and RefreshInterval (how often to update).

(Delphi only   Paradox users see MAKEPROG.FSL)

TProcessFieldEvent type

Declaration

```
TProcessFieldEvent = procedure(Sender: TObject; Field: TField;
                          Location: LongInt) of object;
```

Event type for OnProcessField.

See also

OnProcessField

TRankMode type

Declaration

```
TRankMode = (rmNone, rmCount, rmPercent);
```

See also

RankMode

TStatPanel type

Declaration

```
TStatPanel = (spCache, spMemory, spWords, spLRU);
```

See also

Panels, TStatPanels

TStatPanels type

Declaration

```
TStatPanel = set of TStatPanel;
```

See also

Panels, TStatPanel

TSearchLogic type

Declaration

```
TSearchLogic = (slAnd, slPhrase, slNear, slLike, slOr, slNot,
               slExpression);
```

See also

SearchLogic

TSearchMode type

Declaration

```
TSearchMode = (smSearch, smNarrow, smWiden);
```

See also

SearchMode

TStringFunc type

Declaration

```
TStringFunc = function(const S: string): string;
```

Function to convert a mixed case string to upper case.

See also

Likeness, UpperCase

TUpdateStats component

TUpdateStats drop in form that will automatically configure and display itself when TUpdateDictionary is used.   Primarily designed for monitoring the update process during development.   Double or right click at run time on the form to set or hide various panels.   Open rbUpdate.pas in the Delphi IDE to see a description of all the properties displayed.   Displaying the Words or LRU panels will slow performance as these panels require a call to a routines which has to iterate through the cache.

Panel	Description
Cache	Displays all the major properties
Memory	Displays current and maximum memory usage, as well as the memory limit
Words	Displays the number of uncompressed, compressed, and unwritten words in the cache. The number or uncompressed and compressed words plus the number of omit words (not displayed) make up all the words in the cache (CacheCount, not displayed).
LRU	Least Recently Used statistics. The highest LRU is the most recently used word, the lowest is the least recently used. The Current LRU is the index of the most recently used word. Words with LRUs greater than the Compress LRU are held in memory uncompressed. If the LRU is less than or equal to Compress LRU and greater than Flush LRU, the word is held in memory and compressed. Words with LRUs less than or equal to Flush LRU are removed from memory.

Properties include Expanded (displays three columns of data instead of two), Form (pointer to the form, run-time), Panels (which panels to display), RefreshInterval (how often to update the data), and Updater (source TUpdateDictionary).

The component does not hook itself into the TUpdateDictionary.OnWrite event.   You can do this by adding the following to the OnWrite event:

```
      UpdateStats1.UpdateStats;
```

(Delphi only)

TUpdateTable component

The TUpdateTable component is a descendent of TTable that has the TUpdateDictionary AfterDelete, AfterPost, BeforeDelete, BeforeEdit, and BeforeInsert methods build in.   TUpdateDictionary checks to see if its DataSource.DataSet is a TUpdateTable, and if so, will automatically connect the appropriate methods. TUpdateTable adds one property TTable component, UpdateStats.

(Delphi only)

UnwrittenWords property

Applies to

TUpdateDictionary

Declaration

```
property UnwrittenWords: LongInt;
```

If DelayedWrites is False, always returns zero, otherwise returns the number of words in the cache that have been edited, but not written to disk.   This property requires that all the words in the cache be checked, so performance sensitive applications should minimize the number of calls to this property.

Run-time.

Example

```
UnwrittenWordsLabel.Caption :=
     IntToStr(UpdateDictionary1.UnwrittenWords);
```

See also

CacheCount, DelayedWrites, OnWrite

Updater property

Applies to

TUpdateStats

Declaration

```
property Updater: TUpdateDictionary;
```

Determines the TUpdateDictionary whose statistics are being displayed.

Example

```
UpdateStats1.Updater := UpdateDictionary2;
```

UpdateStats property

Applies to

TUpdateTable

Declaration

```
property UpdateStats: TUpdateStats;
```

If set, the TUpdateTable will call UpdateStats after every change to the table.

Example

```
UpdateTable1.UpdateStats := UpdateStats1;
```

UpperCase property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property UpperCase: TStringFunc;
```

Function used to convert words to upper case. Since this routine is called so often, it is critical for performance reasons to make it as efficient as possible. AnsiUpperCase was not used since SysUtils.UpperCase is much faster. However, international users may prefer to use AnsiUpperCase (or, if $H-, AnsiUpperCaseShort32).

Case conversion can be turned off by defining your own upper case function that performs no case conversion.

Default is SysUtils.UpperCase. Run-time.

Example

```
MakeDictionary1.UpperCase := AnsiUpperCase;
```

See also

Builder

WidenSearch method

Applies to

TSearchDictionary

Declaration

```
procedure WidenSearch(S: string);
```

WidenSearch is a shorthand equivalent to:

```
     SearchDictionary1.SearchFor := S;
     SearchDictionary1.SearchMode := smWiden;
     SearchDictionary1.Execute;
```

Example

```
with SearchDirectory1 do
 begin
  SearchLogic := slAnd;
  Search('delphi paradox');
  WidenSearch('borland')
 end;
```

See also

Execute, NarrowSearch, Search, SearchFor, SearchLogic, SearchMode

WordDelims property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property WordDelims: string;
```

These characters define the beginning and end of a word. Control characters can be entered as ^M and ^J. To enter a ^, use ^^.

The most common delimiters are spaces, commas, and periods. You will probably want to include other punctuation (colon, semi-colon, double quotes, single quotes, question marks, exclamation marks), parentheses, braces, brackets, and mathematical symbols.

Whether you want to include any of these @#$%&\~ depends on your database.

In some instances, you may consider using numbers as word delimiters. This will effectively eliminate all numbers from the dictionary.

Be aware that the period delimiter causes havoc with numerical values embedded in string or memo fields: a number like '19.95' would become two words: '19' and '95'.

Word delimiters are not applied to any numerical, boolean, date, or time fields, only to string and memo fields.

The same set of WordDelims used to make the dictionary should be used when searching or updating the dictionary.

Example

```
UpdateDictionary1.WordDelims := ' ,.(){}[]!;:?/\';
```

See also

Execute, NarrowSearch, Search

WordFieldSize property

Applies to

TMakeDictionary

Declaration

```
property WordFieldSize: Integer;
```

WordFieldSize defines the size of the Word field in the WordsTable. Changing the value of WordFieldSize causes the WordsTable to be recreated. While words added to the WordsTable that exceed WordFieldSize will be processed (albeit they are truncated), you run the risk of a key violation that will terminate phase two of the execution process.

Setting WordFieldSize too low causes a number of problems: it will increase the number of key violations during the building of a dictionary, it will increase the number of false matches during a dictionary update, and it can make searches more ambiguous.

Default is 20.

Example

```
MakeDictionary1.WordFieldSize := 25;
```

See also

WordsTable

WordsTable property

Applies to

TMakeDictionary, TSearchDictionary, TUpdateDictionary

Declaration

```
property WordsTable: TTable;
```

This table contains the results of the Execute procedure, is the table updated by TUpdateDictionary, and is used by Search and NarrowSearch to locate records.

For TMakeDictionary, if table does not exist, one will be created. If the table does exist, it will be recreated to ensure the field sizes are up to date. Because TMakeDictionary creates and /or recreates this table, the table should be inactive in design mode in order to avoid the 'Table is busy' error.

For TSearchDictionary and TUpdateDictionary, the table must exist.

Example

```
SearchDictionary1.WordsTable := Table1;
```

See also

DataSource

WriteCache method

Applies to

TUpdateDictionary

Declaration

```
procedure WriteCache;
```

Forces any unwritten records in the memory cache to be written to disk.

Example

```
with UpdateDictionary1 do
 if CacheEdits + CacheInserts > 0 then
  begin
   WriteCache;
   ResetStats
  end;
```

See also

CacheEdits, CacheInserts, DelayedWrites, FlushCache, ResetStats

Basic Methods & Properties

DataSource property

DataTypes property

Execute method

FieldNames property

FieldTypes property

IndexMode property

MatchCount property

MatchTable property

MemoryLimit property

MinWordLen property

NarrowSearch method

RankMode property

RecordCount property

Search method

SearchFor property

SearchLogic property

SearchMode property

WidenSearch method

WordDelims property

WordsTable property

Intermediate Methods & Properties

AfterDelete method

AfterPost method

AnyChar property

BeforeDelete method

BeforeEdit method

BeforeInsert method

BlobBytesRead property

BlobBytesWritten property

Builder property

CacheCount property

CacheEdits property

CacheInserts property

CacheReads property

CreateMatchTable method

DiskDeletes property

DiskEdits property

DiskInserts property

DiskReads property

IndexFieldName property

MatchingWords method

MaxMemUsed property

MemCompression prop.

MemoryUsage property

NearWord property

OneChar property

RecordLimit property

RecordNo property

SourceReads property

State property

StrictChecking property

SubFieldNames property

Advanced Methods & Properties

AltMemMgr property

BatchAdd method

BatchDelete method

BlobFieldSize property

DelayedWrites property

EDictionary object

ErrorPos property

FileCompression property

FindXxxx methods

FlushCache method

IndexRange property

KeyViolName property

LikeFieldSize property

Likeness property

LoadOmitsFromTable

MatchBits property

Matches method

MinIndex property

MinOrdIndex property

OmitList property

OnPhaseOne property

OnPhaseTwo property

OnProcessField property

OnSearch property

ProcessField method

ProcessList method

ProcessPChar method

ProcessRecord method

ProcessWord method

ResetStats method

SourceRange property

UnwrittenWords property

UpperCase property

WordFieldSize property

WriteCache method

Paradox Interface

The Paradox interface to the Rubicon DLL uses the same naming conventions as used in the Delphi code. Rubicon for Paradox uses the Libraries and handles to replace the equivalent Delphi objects. Where the Delphi syntax is:

```
Object.Method(paramater list)
```

The Paradox interface is:

```
Library.Method(Handle, parameter list)
```

For Delphi properties the syntax is:

```
Object.Property := Value;
Value := Object.Property;
```

The Paradox interface is:

```
Value = Library.getProperty(Handle, rbxPropertyName)
Library.setProperty(Handle, rbxPropertyName, Value)
```

where

```
rbxPropertyName is a property constant
Value is the variable to set or get
```

rbx may be one of the following:

```
rbbXxxx is a Logical or Boolean property
rbfXxxx is a function pointer address
rbhXxxx is a Handle property
rblXxxx is a LongInt property
rbsXxxx is a String property
```

For a complete listing of constants, refer to RUBICON.LSL.

Some properties may be set either by a constant or a string. The following are equivalent:

```
libRubicon.setProperty(Handle, rblIndexMode, imSeqNo)
libRubicon.setProperty(Handle, rbsIndexMode, "imSeqNo")
```

addLookupField method

Applies to

Rubicon for Paradox

Declaration

```
method addLookupField(Handle LongInt, KeyFields String,
          LookupTable String, LookupKeyFields String,
          LookupResultField String)
```

Use this method to build a multi table data model by adding lookup fields to the DataSource. The new lookup field is given the name LookupResultField. If this name already exists in the DataSource, the name becomes LookupTable + LookupResultField.

Example

```
libRubicon.setProperty(hMake, rbsDataSource, ":project:orders")
libRubicon.addLookupField(hMake, "CustID", ":project:customer",
                           "CustNo", "Name")
libRubicon.setProperty(hMake, rbsFieldNames, "Name")
```

check method

Applies to

Rubicon for Paradox

Declaration

```
method check(Code LongInt)
method checkErrorCode()
```

Used to check the result codes returned by direct calls to the Rubicon API. If an error occurs, check calls fail(). checkErrorCode() is the same, but used for API calls that do not return error codes directly.

Example

```
Handle = rbiCreateDictionary(TMakeDictionary)
libRubicon.checkErrorCode()
libRubicon.check(rbiSetProperty(Handle, rblMemoryLimit,
16000000))
```

convertWorkPriv method

Applies to

Rubicon for Paradox

Declaration

```
method convertWorkPriv(S String) String
```

The Rubicon DLL can handle table names that contain aliases that are defined in the BDECFG. However, the DLL cannot resolve table names that contain :WORK: or :PRIV:. To work around this limitation, simply enclose any table name parameter with convertWorkPriv.

Example

```
libRubicon.setProperty(hMake, rbsDataSource,
    libRubicon.convertWorkPriv(fldSourceTable.value))
```

enumHandle method

Applies to

Rubicon for Paradox

Declaration

```
type
 InfoArray = DynArray[] String
endType

method enumHandle(Handle LongInt, var Ary InfoArray)
```

A very useful way of displaying all the property settings for the Handle.

Example

```
libRubicon.enumHandle(hMake,Properties)
Properties.view("Make Properties")
```

getLocation method

Applies to

Rubicon for Paradox

Declaration

```
method getLocation(Handle LongInt, var TC TCursor) LongInt
```

Returns the location of the cursor within the table.   The value of location is dependent on which record the cursor is currently positioned on and the IndexMode.

Example

```
Location = libRubicon.getLocation(hSearch, TC)
```

See also

gotoLocation

getPropertyType method

Applies to

Rubicon for Paradox

Declaration

```
method getPropertyType(Property LongInt) String
```

Returns the property type of the property: Function, Handle, Logical, LongInt, or String.

Example

See getProperty example

getProperty method

Applies to

Rubicon for Paradox

Declaration

```
method getProperty(Handle LongInt, Property LongInt) AnyType
method getPropertyL(Handle LongInt, Property LongInt) LongInt
method getPropertyS(Handle LongInt, Property LongInt) String
```

Low level routines to get properties of a specific type.

Example

```
method getProperty(Handle LongInt, Property LongInt) AnyType
var
 L        LongInt
 PropType String
endVar
 PropType = getPropertyType(Property)
 if PropType = "String" then
  return getPropertyS(Handle,Property)
 else
  L = getPropertyL(Handle,Property)
  if PropType = "Logical" then
     if L = 0 then
      return True
     else
      return False
     endIf
  else
     return L
  endIf
 endIf
endMethod
```

See also

setProptery

gotoLocation method

Applies to

Rubicon for Paradox

Declaration

```
method gotoLocation(Handle LongInt, var TC TCursor,
                    Location LongInt) Logical
```

Moves the cursor to Location.

Example

```
libRubicon.getLocation(hSearch, TC, Location)
```

See also

getLocation

MakeProg form

Applies to

Rubicon for Paradox

Declaration

```
MakeProg.fsl
```

A form that displays a progress bar and statistics while the WordsTable is being built. MakeProg has not been optimized to work with SQL tables.

Example

```
if F.open("MakeProg") then
 F.build(hMake)
 F.wait()
 F.close()
endIf
```

Rubicon Library

Applies to

Rubicon for Paradox

Declaration

This library contains all the constants, methods, and types needed to implement Rubicon for Paradox. The following methods closely match their Delphi equivalents and have already been documented in the Reference Section.

```
method afterDelete(Handle LongInt)
method afterPost(Handle LongInt, var TC TCursor)
method batchAdd(Handle LongInt, var TC TCursor)
method batchDelete(Handle LongInt, var TC TCursor)
method beforeDelete(Handle LongInt, var TC TCursor)
method beforeEdit(Handle LongInt, var TC TCursor)
method beforeInsert(Handle LongInt)
method createDictionary(DictionaryType LongInt) LongInt
method execute(Handle LongInt)
method findFirst(Handle LongInt, var TC TCursor) Logical
method findLast(Handle LongInt, var TC TCursor) Logical
method findNext(Handle LongInt, var TC TCursor) Logical
method findPrior(Handle LongInt, var TC TCursor) Logical
method flushCache(Handle LongInt)
method getMatchingWords(Handle LongInt) String
method loadOmitsFromTable(Handle LongInt, TableName String,
                      FieldName String)
method writeCache(Handle LongInt)
```

search method

Applies to

Rubicon for Paradox

Declaration

```
method search(Handle LongInt, TimeOut LongInt)
```

Same as execute, but places a time limit of TimeOut milliseconds.

Example

```
libRubicon.search(hSearch, 5000)  ;// give up after 5 seconds
```

See also

execute

setProperty method

Applies to

Rubicon for Paradox

Declaration

```
method setProperty(Handle LongInt, Property LongInt, Value
AnyType)
method setPropertyL(Handle LongInt, Property LongInt, Value
LongInt)
method setPropertyS(Handle LongInt, Property LongInt, Value
String)
```

Low level routines to set properties of a specific type.

Example

```
method setProperty(Handle LongInt, Property LongInt, Value
AnyType)
var
 PropType String
endVar
 PropType = getPropertyType(Property)
 if PropType = "String" then
  setPropertyS(Handle,Property,Value)
 else
  if (PropType = "Logical") and
       (Value.dataType() = "Logical") then
     Value = LongInt(Value)
  endIf
  setPropertyL(Handle,Property,Value)
 endIf
endMethod
```

See also

getProptery

setMenuAction method

Applies to

Rubicon for Paradox

Declaration

```
method setMenuAction(Handle LongInt, MenuID LongInt, Tics LongInt)
```

The Rubicon DLL will post a message to the calling form with id MenuID every Tics milliseconds during the execution of a dictionary build or search.

Example

See code in MakeProg.fsl

Template form

Applies to

Rubicon for Paradox

Declaration

```
Template.fsl
```

This form is designed for 16 bit Paradox users. It contains all the constants, types, and uses statements necessary to construct a form using Rubicon for Paradox. In addition, it contains open and close methods that will create the necessary handles. You will need to edit the open method and select which handles you want created.

This form is not necessary for Paradox 7 32 bit users as this version implements the extended uses syntax which imports all the constants, types, and uses statements from Rubicon.lsl without having to redeclare them.

Definitions

Search Table: The table to be searched.

Words Table: A table that contains all the words used in the search table and their locations with the table.

Dictionary: Same as the words table.

Ordinal: A SmallInt, Word, Integer, or LongInt field type.

Common Questions

Does Rubicon handle memo fields?

Yes, Rubicon handles all standard field types. Memo fields are limited to 64k in 16 bit applications. Nonstandard fields such as ftBlob, ftVarBytes, and 16 bit memo fields exceeding 64k can be handled via the OnProcessField event handler.

How can I expand acronyms?

Use the OnProcessField event.

How can I use TSearchDictionary as a filter for my DataSource?

In Delphi 2.0, you may simply define an OnFilterRecord event handler and test whether the current record matches the search criteria by calling SearchDictionary1.Matches. In Delphi 1.0, you will have to define a dbiAddFilter routine or equivalent.

Can I reduce the size of WordsTable by setting WordFieldSize to a lower value?

Yes, but you run the risk of increasing the number of key violations, false matches during updates, and ambiguous search results.

How can I limit a dictionary build to a maximum amount of RAM?

In order to conserve RAM, be sure to set MemoryLimit to the desired value. Then in the OnPhaseOne event handler, include code that adds dsAbort to State when the maximum amount of RAM is exceeded (this should be approximately 512kb higher than MemoryLimit).

Does the AltMemMgr option replace the standard Delphi memory manager?

No, it merely supplements the standard memory manager during GetMem and FreeMem calls, and only is used for cache memory.

MemoryUsage includes what kinds of memory?

It is primarily made up of the memory used to cache the indexes. If AltMemMgr is True, it also includes any memory in the memory pool. Some internal buffers are also included. It does not included the memory used by the FCache data structure (a StDictionary), various TLists, and other ancillary data structures.

My application seems to stall while using TUpdateDictionary. Can this be avoided?

If you have set DelayedWrites to True, TUpdateDictionary will write records to disk when the cache is full. You may use the OnWrite event to do some processing while the cache is being compressed. You should not interrupt this process. Calls to WriteCache or FlushCache may also cause delays. Here, you may abort the process and then resume it later.

Troubleshooting

The blob portion of the WordsTable seems excessively large

Check to see if the table type being used has a default or minimum blob size. If so, see if the default size can be reduced to 32 or 64 bytes.

Words seem to be missing or incorrectly associated in the dictionary

If the length of the words in question exceeds the WordFieldSize property, increase WordFieldSize and rebuild the dictionary.

Searches aren't finding the correct records

If the dictionary was made with a non-zero value for MinOrdIndex, be sure you are using the same value for all subsequent searches and updates. Also check that the same values for IndexMode and WordDelims were used to build and/or update the dictionary and the searches.

"Decompress Buffer Too Small" error raised during searches

TSearchDictionary allocates a decompression buffer to hold an index of IndexRange size, but has tried to read an index with a size greater than IndexRange. This is usually a result of the properties of the TSearchDictionary not being set to the same values as TMakeDictionary or TUpdateDictionary. Check the IndexMode, MinOrdIndex, and SourceRange properties. It may also be caused by having records deleted from the DataSource without having updated the WordsTable.

Specifying the :WORK: or :PRIV: alias in Paradox for Windows doesn't work

These aliases are know only to Paradox for Windows. See the method convertWordPriv to use these aliases with the DLL.

A slOr search on '*' followed by a slNot search should return zero matches, but doesn't

The IndexMode is probably imOrdinalIndex and what is being returned are the gaps between index values. Since these records don't really exists, a call to CreateMatchTable will return an empty table. The correct way to perform the above search is to follow the slOr search followed by a slNot NarrowSearch. You may also encounter this problem when using the imRecordNo or imSeqNo IndexModes and RecordLimit is set to a positive value.

All the values for WordCount and BlobSize are zero in my WordsTable

Check to be sure that the dbiWrite compiler option in TARUBICN.INC is disabled. If enabled and the database format of WordsTable does not support 32 bit integers, then the problems described will result.

Words at the end of memos are not indexed

16 bit applications are limited to memo lengths of 64K. If possible, compile your application with Delphi 2.

The Matches method doesn't seem to be working

Matches returns a value that indicates whether the current record in the DataSource meets the search criteria. You may have to call UpdateCursorPos before calling Matches. In addition, when the IndexMode is imRecordNo or imSeqNo and Matches is called from within a filter, it may not be possible to synchronize the DataSet to the physical record number.

Number of WordsTable records varies with table type

Normally, the number of unique words should not vary with table type. Differences can arise when the source table(s) contain nonstandard characters that are treated differently by the table types, and therefore result in key violations that cause a word to be excluded from the table. For instance, one table may interpret Canada and Cañada as two different words, the other may treat them as the same (and thus one would be excluded because of a key violation).

Processing TMakeDictionary.Execute slows down exponentially

If using Delphi 2.00/2.01, you have probably run into the memory fragmentation bug. Set the AltMemMgr property to True. See Delphi 2.0 Memory Fragmentation for more details.

How to Order

To receive a registered version of Rubicon, technical support, along with free updates of version 1.x, just send $99 U.S. with the order form that appears at the end of this document. Or you may email your name, address, MasterCard or Visa number, and expiration date to Tamarack Associates at sales@tamaracka.com or 72365.46@compuserve.com. Sales tax will be added to California orders. Delivery is free via CompuServe or Internet, $5 in North America (Canada, Mexico, & U.S)., $10 outside of North America. Please specify 3.5" or 5.25" diskettes.

Rubicon for Delphi includes all source code (except the SysTools units). Rubicon for Paradox comes with unrestricted DLLs.

Rubicon for Delphi and Rubicon for Paradox are also available through CompuServe SWREG for $99 U.S. The SWREG registration ID is 11536 for the Delphi version, 13217 for the Paradox version.

Order both the Delphi and Paradox versions for $149US. Existing users of one version may purchase the other version for $50US. These must be ordered directly from Tamarack Associates.

Orders are generally filled the day they are received, with the exception of holidays and vacations. If your order has not been filled within 48 hours, please email us at admin@tamaracka.com.

Please read the Purchase Agreement before registering.

Other Products from Tamarack Associates

TtaDBMRO 2.0 is a popular data aware control for Borland's Delphi development environment. Building on the success of TtaDBMRO 1.x, version 2.0 delivers a 32 bit performance and compatibility while maintaining the ability to be used in both 16 and 32 bit environments.

TtaDBMRO provides a TDBCtrlGrid-like control that allows the developer to display data aware controls in a scrollable manner.

Unlike Borland's TDBCtrlGrid, TtaDBMRO supports all Borland field data aware controls, compatible with both Delphi 1.0 and 2.0, allows the use of data aware controls with different DataSources, provides several ways to customize the appearance of records, and supports titles.

TtaDBMRO is compatible with InfoPower 1.2, Orpheus 2.0, TDBLookupComboPlus 4.1, and TDBComboBoxPlus 2.1. It has been tested running under Windows 3.11, Windows 95, and Windows NT 3.51.

Trial run and demonstration versions of TtaDBMRO can be found in the Delphi and Bdelphi forums on CompuServe, Library 22, MRO.ZIP and MRODEMO.ZIP. These files are also available on many Internet sites.

TtaDBMRO is available directly from Tamarack Associates for $25.00US, and includes free 2.xx updates and support via email. The product may also be ordered via CompuServe shareware registration ID 8213 for $29.95US.

Look for Rubicon for Visual dBase and Rubicon for C++ in Q1'97.

Version History

The latest version of Rubicon can always be found on our web site, www.tamaracka.com, or on CompuServe in the Delphi and BDelphi forums, Lib 22 (3d Party Products), in RUBICON.ZIP.

10/18/96	Version 1.20	BatchAdd and BatchDelete procedures added IndexFieldName property added OnWrite event added to TUpdateDictionary AfterDelete method added to TUpdateDictionary LoadOmitsFromTable method added MakeWordDelims function added TMakeProgress component added TUpdateStats component added TUpdateTable component added dsMayAbort added to TDictionaryState MinIndex made visible to TMakeDictionary & TUpdateDictionary Limit parameter removed from CreateMatchTable procedure slAnd searches no longer return records if >=1 word is not found SourceRange & MinOrdIndex behavior changed to eliminate the use of TTable.First/Last with SQL tables. IsIndexUnique rewritten to eliminate DBI calls Fixed TMakeDictionary handling of dbf files w/ deleted records FlushCache frees all memory when AltMemMgr = True TUpdateDictionary improperly deleting words under Delphi 1.0 Delphi Tamarack tab renamed Rubicon Rubicon for Paradox released
09/06/96	Version 1.11	Error messages moved to resource file Exceptions return ErrorCode (see EDictionary) Property editors added for FieldNames & SubFieldNames tarconst.pas, taredit.pas/dfm, tarubicn.rc files added Duplicate field name problem fixed in CreateMatchTable TUpdateDictionary.Builder can no longer be set to itself IndexName required for SQL tables fixed
08/20/96	Version 1.10	slNear, slLike, slPhrase, & slExpression logic types added dsMatching & dsLocating added to TDictionaryState Likeness & LikeFieldSize properties added to TMakeDictionary The following properties were added to TSearchDictionary: DataTypes, ErrorPos, FieldNames, FieldTypes, NearWord, OnProcessField, RankMode, and SourceReads. The following procedures were added to TSearchDictionary

MatchingWords, ProcessField, ProcessList,
ProcessPChar,
ProcessRecord, and ProcessWord
MatchingWords procedure added to TSearchDictionary
ftAutoInc fields become ftInteger in CreateMatchTable
taXpress unit added (expression evaluation)
ResultBits renamed MatchBits
TLogicType renamed TSearchLogic (lt prefixes changed to
sl)
TUpperCaseFunc renamed TStringFunc
Implemented Delphi 2.0x memory fragmentation solution
TMakeDictionary not compressing indexes fixed
FindLast bug fixed

07/08/96 Version 1.00 Initial release

Purchase Agreement

Terms of License Agreement

The Rubicon programs and documentation are the property of Tamarack Associates and are protected by United States Copyright Law, Title 17 U.S. Code, are licensed for use by one person only on as many computers as that person uses.

Where a group of programmers are working together on a project that makes use of Rubicon, we expect that a copy of the software and documentation will be purchased for each member of the group. Contact Tamarack Associates for volume discounts.

You may duplicate the Rubicon programs and documentation files for backup use only.

You may distribute without further licenses or run time fees applications that make use of Rubicon. You may not distribute or duplicate any documentation, source code, or DCU files other than described above.

Limited Warranty

TAMARACK ASSOCIATES MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION ANY WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT WILL TAMARACK ASSOCIATES BE LIABLE TO YOU OR ANY THIRD PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THE PROGRAM OR MANUAL.

By using this product, you agree to this. If you do not agree, immediately return this product for refund.

Development Environment

Rubicon was developed with Delphi 1.02 and 2.01 running under WFWG 3.11, Win95, and NT 3.51 with 40MB of RAM using Paradox tables. SysTools 1.0.

Rubicon for Paradox was also developed using Paradox for Windows 5.0, 7/16, and 7/32.

TurboPower SysTools

Rubicon relies on several routines that are part of TurboPower's SysTools product. Tamarack Associates has a license to redistribute certain SysTools DCUs with Rubicon for Delphi, but cannot redistribute the source code.

If you own SysTools 1.01 or higher, you should enable the HaveSysTools option in TARUBICN.INC.   It is highly recommended that you also disable the ThreadSafe option in STDEFINE.INC for performance reasons.

Note: SysTools 1.00 should not be used with Rubicon because there is a bug in TStDictionary.

For more information about TurboPower products, call 1-800-333-4160, GO TURBOPOWER on CompuServe, or visit www.tpower.com.

Special thanks to Kim Kokkonen for his help on the RLE assembler code!

Trademarks

Rubicon is a trademark of Tamarack Associates

Borland and Paradox are trademarks of Borland International.

SysTools is a trademark of TurboPower Software.

Technical Support

Questions, bug reports and suggestions may be directed to:

Tamarack Associates
CompuServe 72365,46
Internet tech@tamaracka.com
(415) 322-2827 (Voice & Fax)

Delphi Files

Trial run version includes:

BOLTS.ICO	Icon file
COMPARE.DPR	Utility program
DEMO.DPR	Demonstration program
PREVINST.PAS	16/32 bit previous instance code used by Demo
README.TXT	Brief installation instructions
README.1xx	Brief description of version changes
RBABOUT.PAS/DFM	Demo form
RBCOMPAR.PAS/DFM	Compare form
RBDBGRID.PAS/DFM	Demo form
RBDBMEMO.PAS/DFM	Demo form
RBLINK.PAS/DFM	Demo form
RBMAIN.PAS/DFM	Demo form
RBPHASE.PAS/DFM	TMakeProgress, TMakeProgressForm
RBUPDATE.PAS/DFM	TUpdateStats TUpdateStatsForm, TUpdateTable
RBVERIFY.PAS/DFM	Verify form
RUBICN16.RES	16 bit resource file
RUBICN32.RES	32 bit resource file
RUBICON.DOC	This file
RUBICON.HLP	Help file
RUBICON.KWF	Help keyword file
RUBICON.PAS	Source code for registering Rubicon
ST16TR.ZIP	SysTools 16 bit DCUs (Trial Run only)
STBASE.DCU	
STBITS.DCU	
STCONST.DCU	
STCONST.R16	
STDICT.DCU	
ST32TR.ZIP	SysTools 32 bit DCUs (Trial Run only)
STBASE.DCU	
STBITS.DCU	
STCONST.DCU	
STCONST.R32	
STDICT.DCU	
TARCONST.PAS	Rubicon constants
TAREDIT.PAS/DFM	Rubicon property editors
TARUBICN.INC	Include file
TARUBICN.INT	TARUBICN.PAS interface section (Trial Run only)
TARB16TR.ZIP	16 bit trial run DCUs (Trial Run only)
TAXPRESS.DCU	
TALINK.DCU	
TARLE.DCU	
TARUBICN.DCU	

```
  TATOOLS.DCU
TARB32TR.ZIP   32 bit trial run DCUs (Trial Run only)
  TAXPRESS.DCU
  TALINK.DCU
  TARLE.DCU
  TARUBICN.DCU
  TATOOLS.DCU
VERIFY.DPR     Utility program
WILDCARD.PAS   Wildcard matching unit
```

Registered version includes these additional files:

```
ST16.ZIP       SysTools 16 bit DCUs (replaces ST16TR.ZIP)
ST32.ZIP       SysTools 32 bit DCUs (replaces ST32TR.ZIP)
TAXPRESS.PAS   Expression evaluation source code
TALINK.PAS     Source code
TARLE.PAS      Run Length Encoding source code
TARUBICN.PAS   Component source code
TARUBICN.RC    Resource file
TATOOLS.PAS    Extensions to SysTools
```

Paradox Files

CODEVIEW.FSL	Code viewer utility
FILESS16.ZIP	16 bit forms and library
EXMAKE.FSL	Example of TMakeDictionary
EXNAV.FSL	Example of navigating with TSearchDictionary
EXSEARCH.FSL	Example of TSearchDictionary
EXUPDATE.FSL	Example of TUpdateDictionary
MAKEPROG.FSL	Make progress form
RBCNB16.DLL	16 bit Rubicon DLL
RBCNDEMO.FSL	Comprehensive demo program (Paradox 7 only)
RUBICON.LSL	Rubicon library
TEMPLATE.FSL	Template form (16 bit only)
FILES32.ZIP	Same as FILSE16.ZIP, 32 bit versions, plus
RBCNB32.DLL	32 bit Rubicon DLL (replaces RBCNB16.DLL)
RBCNMAKE.LSL	Paradox 7 32 Make library
RBCNSRCH.LSL	Paradox 7 32 Search library
RBCNUPDT.LSL	Paradox 7 32 Update library
MESSAGES.DB/MB	Sample table
README.TXT	Brief installation instructions
README.1xx	Brief description of version changes
RUBICON.DOC	This file
RUBICON.HLP	Help file
_SOURCE.DB/MB	Used by CodeView

Order Form

Rubicon for Delphi/Paradox 1.x
Tamarack Associates
868 Lincoln Avenue
Palo Alto, CA 94301 USA
415-322-2827 (Voice & Fax*)
sales@tamaracka.com
CompuServe 72365,46

Name ______________________________

Company ______________________________

Address ______________________________

City ______________________________

State ______________________________

Country ______________________________

Zip/Postal Code ______________________________

Email ______________________________

Phone ______________________________

Credit Card ___ MasterCard ___ Visa

Card Number ______________________________

Expiration Date ______________________________

Number of copies _____ 3.5" ___ 5.25" ___

Version _____ Delphi ____ Paradox ____ Both

Price per copy $99.00 U.S. ($149 U.S. for both)

Subtotal _____

Sales tax _____ (California residents only)

Shipping & handling _____ (see below)

Total _____

How did you hear about Rubicon? ______________________________________

Shipping & handling: CIS/Internet - none; North America - $5; outside North America - $10.

Rubicon may also be registered through CompuServe SWREG.

SWREG ID: 11536 for Delphi, 13217 for Paradox.

Please read Purchase Agreement before ordering.

*The fax machine can take as long as 45 seconds to answer. Set your fax accordingly.

Properties

AltMemMgr
BlobBytesWritten
BlobFieldSize
CacheCount
DataSource
DataTypes
DiskInserts
FieldNames
FieldTypes
FileCompression
IndexFieldName
IndexMode
IndexRange
KeyViolName
LikeFieldSize
Likeness
MaxMemUsed
MemCompression
MemoryLimit
MemoryUsage
MinOrdIndex
MinWordLen
OmitList
RecordLimit
RecordNo
State
StrictChecking
UpperCase
WordDelims
WordFieldSize
WordsTable

Methods

Execute

LoadOmitsFromTable

ProcessField

ProcessList

ProcessPChar

ProcessRecord

ProcessWord

ResetStats

Events

OnPhaseOne

OnPhaseTwo

OnProcessField

Tasks

16 vs. 32 Bit Memory Issues

Delphi 2.0 Memory Fragmenation

Delphi Installation

Expression Evaluation

Index Modes

Performance Optimization

Search Strategies

Working with Huge Tables

Working with Link, Lookup, or Normalized Tables

Using the Components

See Also

Common Questions

Definitions

Delphi Files

Demo Program

How to Order

Introduction

Technical Support

TMakeDictionary

Troubleshooting

TSearchDictionary

TUpdateDictionary

Utility Programs

Properties

AnyChar
BlobBytesRead
Builder
CacheCount
CacheReads
DataSource
DataTypes
DiskReads
ErrorPos
Execute
FieldNames
FieldTypes
IndexFieldName
IndexMode
IndexRange
Likeness
MatchCount
MatchTable
MaxMemUsed
MemCompression
MemoryLimit
MemoryUsage
MinIndex
MinOrdIndex
NearWord
OmitList
OneChar
RankMode
RecordCount
RecordLimit
MatchBits
SearchFor
SearchLogic
SearchMode
SourceReads
State
StrictChecking
SubFieldNames
UpperCase
WordDelims
WordsTable

Methods

CreateMatchTable

FindXxxx

FlushCache

LoadOmitsFromTable

Matches

MatchingWords

NarrowSearch

ProcessField

ProcessList

ProcessPChar

ProcessRecord

ProcessWord

ResetStats

Search

WidenSearch

Events

OnProcessField

OnSearch

Properties

AltMemMgr
BlobBytesRead
BlobBytesWritten
Builder
CacheCount
CacheEdits
CacheInserts
CacheReads
DataSource
DataTypes
DelayedWrites
DiskDeletes
DiskEdits
DiskInserts
DiskReads
FieldNames
FieldTypes
FileCompression
IndexFieldName
IndexMode
IndexRange
Likeness
MaxMemUsed
MemCompression
MemoryLimit
MemoryUsage
MinOrdIndex
MinWordLen
OmitList
State
StrictChecking
UnwrittenWords
UpperCase
WordDelims
WordsTable

Methods

AfterDelete

AfterPost

BatchAdd

BatchDelete

BeforeDelete

BeforeEdit

BeforeInsert

FlushCache

LoadOmitsFromTable

ProcessField

ProcessList

ProcessPChar

ProcessRecord

ProcessWord

ResetStats

WriteCache

Events

OnProcessField

OnWrite