eGenix.com

mxBeeBase

On-disk B+Tree Database Kit
for Python

Version 3.2

Printed in Germany.

Contents

1. Introduction

mxBeeBase is a high performance construction kit for disk based indexed databases. It offers components which you can plug together to easily build your own custom mid-sized databases.

The two basic building blocks in mxBeeBase are *storage* and *index*. Storage is implemented as variable record length data storage with integrated data protection features, automatic data recovery and locking for multi process access. Indexes use a high performance optimized B+Tree implementation built on top of *Thomas Niemann*'s Cookbook B+Tree implementation.

> **Note:**
> This documentation is still incomplete. The software itself has been in use for quite a while, but the documentation still needs to be completed. For now, all we can offer you is to read the code which is well documented.

1.1 Limitations

The current storage size limit is `sizeof(long)` which gives you an address range of around 2GB on 32-bit platforms and a few TB on 64-bit platforms. The number of items in the index(es) is limited by `sizeof(int)`.

BeeDict index files are also **not portable** between 32-bit and 64-bit platforms.

> **Windows x64 is currently not supported.** Even though mxBeeBase will compile on Windows x64 with warnings and the resulting code will work most of the time, there are cases where the implementation can cause segfaults due to the way C integers and longs are sized on Windows x64.

2. mx.BeeBase.BeeDict Module

The mx.BeeBase.BeeDict module provides three high level on-disk dictionary implementations. The first (*BeeDict*) can work with arbitrary hashable key objects, the second (*BeeStringDict*) uses limited sized strings as basis providing slightly better performance and the third (*BeeFixedLengthStringDict*) uses fixed length strings which may also contain \0 characters. All variants need pickleable Python objects as keys and values.

Data transfer to and from the dictionaries is done in the same way as for in-memory dictionaries, e.g. `d['key'] = 1; print d['key']; del d['key]`, so usage should be mostly transparent to Python programs using either in-memory or on-disk dictionaries.

There are a few differences to keep in mind:

- Not all dictionary methods are implemented.
- In-place modifications of mutable values, such as lists or dictionaries, are not automatically written to the on-disk dictionary. You have to explicitly reassign the value to mark the entry as modified: `listvalue = d['key']; listvalue.append(1); d['key'] = listvalue`.
- Since on-disk dictionaries tend to be large, the implementations provide the Python iterator API which allows iterating over the complete dictionary without loading the complete data into memory. In Python 2.2 and up, you can use `'for key, value in beedict:'` for iteration. In Python 2.1 and below, use `'for key, value in beedict.iteritems():'` instead.

2.1 BeeDict Objects

BeeDict objects are on-disk dictionaries which use a hash-to-address index. Both Keys and values must be pickleable and can have arbitrary size (keys shouldn't be too long though); keys have to be hashable.

Hash collisions are treated by sequential reads of all records with the same hash value and testing for equality of keys. This can be expensive !

BeeDicts use a BeeStorage.BeeKeyValueStorage instance as storage object and a BeeIndex.BeeIntegerIndex instance as index object.

Calling `len()` on a `BeeDict` object will raise an `Error` in case uncommitted changes exist.

2.1.1 BeeDict Object Constructors

BeeDict objects are constructed using:

```
BeeDict(name, min_recordsize=0, readonly=0, recover=0,
   autocommit=0, validate=0, maxcachesize=None)
```

Create an instance using `name` as basename for the data and index files. Two files will be created: name.dat and name.idx.

`min_recordsize` is passed to the `BeeStorage` as indicator of the minimum size for data records. `readonly` can be set to true to open the files in read-only mode, preventing any disk modifications.

To open the dictionary in recovery mode, pass a keyword `recover=1`. Then run `.recover()` and reopen using the normal settings. The `AutoRecover()` wrapper can take care of this action for you automatically.

If `autocommit` is true the cache control will do an automatic `.commit()` whenever the transaction log overflows.

If `validate` is true, the dictionary will run a validation check after having successfully opened storage and index. `RecreateIndexError` or `RecoverError` exceptions could be raised in case inconsistencies are found.

`maxcachesize` defines the maximum size of the in-memory transaction cache. It defaults to `MAXCACHESIZE` if not given.

2.1.2 BeeDict Object Methods

BeeDict objects have the following methods:

`.flush()`

Flush buffers to disk.

`.close()`

Flush buffers and close.

This does not issue a `.commit()`, so the current transaction is rolled back.

`.commit()`

Commits all changes and starts a new transaction.

`.rollback()`

Rolls back the current transaction. All changes are undone.

`.changed()`

Return true in case the current transaction includes changes to the database, false otherwise.

`.has_key()`

Check if the dictionary has an item indexed by key.

Successfully found items are put in the cache for fast subsequent access.

`.get(key,default=None)`

Get item indexed by key from the dictionary or default if no such item exists.

This first tries to read the item from cache and reverts to the disk storage if it is not found.

`.cursor(key=FirstKey, default=None)`

Return a `BeeDictCursor` instance for the dictionary.

If key is given, the cursor is positioned on that key in the dictionary. Otherwise, the first entry in the dictionary is used which guarantees that all entries are scanned.

In case the key is not found, default is returned instead.

Note that cursors operate with the data on disk meaning that any uncommitted changes will not be seen by the cursor.

`.garbage()`

Determine the amount of garbage in bytes that has accumulated in the storage file.

This amount would be freed if .collect() were run.

`.collect()`

Run the storage garbage collector.

Storage collection can only be done for writeable dictionaries and then only if the current transaction does not contain any pending changes.

This can take a while depending on the size of the dictionary.

`.recover()`

Recover all valid records and recreate the index.

`.keys()`

Return a list of keys.

The method does not load any data into the cache, but does take notice of uncommitted changes.

For an iterative approach which uses less memory, see the .iterkeys() method.

`.values()`

Return a list of values.

The method does not load any data into the cache, but does take notice of uncommitted changes.

For an iterative approach which uses less memory, see the .itervalues() method.

`.items()`

Return a list of items.

The method does not load any data into the cache, but does take notice of uncommitted changes.

For an iterative approach which uses less memory, see the .iteritems() method.

`.remove_files()`

Deletes the storage and index files used for the `BeeDict`.

Closes the `BeeDict` before proceeding with the removal.

USE WITH CARE !

`.iteritems()`

Return an iterator which iterates over the dictionary items and returns (key, value) tuples. The iterator is compatible to Python's for-statement.

The dictionary may not have uncommitted changes !

`.iterkeys()`

Return an iterator which iterates over the dictionary keys. The iterator is compatible to Python's for-statement.

The dictionary may not have uncommitted changes !

`.itervalues()`

Return an iterator which iterates over the dictionary values. The iterator is compatible to Python's for-statement.

The dictionary may not have uncommitted changes !

2.1.3 BeeDict Object Attributes

BeeDict objects have the following read-only attributes:

`.name`

Base name of the dictionary. The implementation uses two files for the on-disk representation: name.dat and name.idx.

`.closed`

Closed flag.

`.readonly`

Read-only flag.

`.autocommit`

Auto-commit flag.

`.FirstKey`

Special key object useable to represent the first key in the (sorted) B+Tree index.

`.LastKey`

Special key object useable to represent the last key in the (sorted) B+Tree index.

2.2 BeeDictCursor Objects

BeeDictCursor objects are intended to iterate over the database one item at a time without the need to read all keys. You can then read/write to the current cursor position and thus modify the dictionary in place.

Note that modifying the targeted dictionary while using a cursor can cause the cursor to skip new entries or fail due to deleted items. Especially deleting the key to which the cursor currently points can cause errors to be raised. In all other cases, the cursor will be repositioned.

Calling `len()` on a `BeeStringDict` object will raise an `Error` in case uncommitted changes exist.

2.2.1 BeeDictCursor Object Constructors

BeeDictCursor objects are constructed using the `BeeDict .cursor()` method.

2.2.2 BeeDictCursor Object Methods

BeeDictCursor objects have the following methods:

`.position(key, value=None)`

Position the index cursor to index[key]. If value is given, index[key] == value is assured.

`key` may also be `FirstKey` or `LastKey` (in which case value has to be None).

`.next()`

Moves to the next entry in the dictionary.

Returns true on success, false if the end-of-data has been reached.

`.prev()`

Moves to the previous entry in the dictionary.

Returns true on success, false if the end-of-data has been reached.

`.read()`

Reads the value object from the dict to which the cursor currently points.

`.read_value()`

Alias for .read().

`.read_key()`

Reads the key object from the dict to which the cursor currently points.

`.read_item()`

Reads the (key, value) item object from the dict to which the cursor currently points.

`.write()`

Writes the object to the dict under the key to which the cursor currently points.

The new data is not written to disk until the dictionary's current transaction is committed.

2.2.3 BeeDictCursor Object Attributes

BeeDictCursor don't have any useful attributes. Use the instance methods to query the key and value objects.

2.3 BeeStringDict Objects

BeeStringDict objects are on-disk dictionaries which use limited size string keys as index. The strings may not contain embedded \0 characters (if you need these, have a look at the *BeeFixedLengthStringDict* object). Values must be pickleable and can have arbitrary size.

Since hash collisions cannot occur this dictionary type may have some performance advantages over the standard `BeeDict` dictionary.

2.3.1 BeeStringDict Object Constructors

BeeStringDict objects are constructed using:

```
BeeStringDict(name, keysize=10, min_recordsize=0, readonly=0,
   recover=0, autocommit=0, validate=0, maxcachesize=None)
```

Create an instance using `name` as base name for the data and index files. Two files will be created: name.dat and name.idx.

`keysize` gives the maximal size of the strings used as index keys. `min_recordsize` is passed to the `BeeStorage` as indicator of the minimum size for data records. `readonly` can be set to true to open the files in read-only mode, preventing any disk modifications.

To open the dictionary in recovery mode, pass a keyword `recover=1`. Then run `.recover()` and reopen using the normal settings. The `AutoRecover()` wrapper can take care of this action for you automatically.

If `autocommit` is true the cache control will do an automatic `.commit()` whenever the transaction log overflows.

If `validate` is true, the dictionary will run a validation check after having successfully opened storage and index. `RecreateIndexError` or `RecoverError` exceptions could be raised in case inconsistencies are found.

`maxcachesize` defines the maximum size of the in-memory transaction cache. It defaults to `MAXCACHESIZE` if not given.

Note that the `keysize` is currently not stored in the dictionary itself -- you'll have to store this information in some other form (this may change in future versions of mxBeeBase).

2.3.2 BeeStringDict Object Methods

BeeStringDict objects have the following methods:

`.commit()`

Commits all changes and starts a new transaction.

`.rollback()`

Rolls back the current transaction. All changes are undone.

`.changed()`

Return true in case the current transaction includes changes to the database, false otherwise.

`.has_key()`

Check if the dictionary has an item indexed by key.

Successfully found items are put in the cache for fast subsequent access.

`.get(key,default=None)`

Get item indexed by key from the dictionary or default if no such item exists.

This first tries to read the item from cache and reverts to the disk storage if it is not found.

`.cursor(key=FirstKey, default=None)`

Return a `BeeStringDictCursor` instance for the dictionary.

If key is given, the cursor is positioned on that key in the dictionary. Otherwise, the first entry in the dictionary is used which guarantees that all entries are scanned.

In case the key is not found, default is returned instead.

Note that cursors operate with the data on disk meaning that any uncommitted changes will not be seen by the cursor.

`.garbage()`

Determine the amount of garbage in bytes that has accumulated in the storage file.

This amount would be freed if .collect() were run.

`.collect()`

Run the storage garbage collector.

Storage collection can only be done for writeable dictionaries and then only if the current transaction does not contain any pending changes.

This can take a while depending on the size of the dictionary.

`.recover()`

Recover all valid records and recreate the index.

`.keys()`

Return a list of keys.

The method will raise an error if there are uncommitted changes pending. Output is sorted ascending according to keys.

For an iterative approach which uses less memory, see the .iterkeys() method.

`.values()`

Return a list of values.

The method will raise an error if there are uncommitted changes pending. Output is sorted ascending according to keys.

For an iterative approach which uses less memory, see the .itervalues() method.

`.items()`

Return a list of items.

The method will raise an error if there are uncommitted changes pending. Output is sorted ascending according to keys.

For an iterative approach which uses less memory, see the .iteritems() method.

`.remove_files()`

Deletes the storage and index files used for the `BeeDict`.

Closes the `BeeDict` before proceeding with the removal.

USE WITH CARE !

`.iteritems()`

Return an iterator which iterates over the dictionary items and returns (key, value) tuples. The iterator is compatible to Python's for-statement.

The dictionary may not have uncommitted changes !

The iteration is done in ascending key order.

`.iterkeys()`

Return an iterator which iterates over the dictionary keys. The iterator is compatible to Python's for-statement.

The dictionary may not have uncommitted changes !

The iteration is done in ascending key order.

`.itervalues()`

Return an iterator which iterates over the dictionary values. The iterator is compatible to Python's for-statement.

The dictionary may not have uncommitted changes !

Even though only the values are returned, the iteration is done in ascending key order.

2.3.3 BeeStringDict Object Attributes

BeeDict objects have the following read-only attributes:

`.name`

Base name of the dictionary. The implementation uses two files for the on-disk representation: name.dat and name.idx.

`.closed`

Closed flag.

`.readonly`

Read-only flag.

`.autocommit`

Auto-commit flag.

`.FirstKey`

Special key object useable to represent the first key in the (sorted) B+Tree index.

`.LastKey`

Special key object useable to represent the last key in the (sorted) B+Tree index.

2.4 BeeStringDictCursor Objects

BeeStringDictCursor objects are intended to iterate over the database one item at a time without the need to read all keys. You can then read/write to the current cursor position and thus modify the dictionary in place.

Note that modifying the targeted dictionary while using a cursor can cause the cursor to skip new entries or fail due to deleted items. Especially deleting the key to which the cursor currently points can cause errors to be raised. In all other cases, the cursor will be repositioned.

2.4.1 BeeStringDictCursor Object Constructors

BeeStringDictCursor objects are constructed using the `BeeStringDict.cursor()` method.

2.4.2 BeeStringDictCursor Object Methods

BeeStringDictCursor objects have the following methods:

`.position(key, value=None)`

Position the index cursor to index[key]. If value is given, index[key] == value is assured.

`key` may also be `FirstKey` or `LastKey` (in which case value has to be None).

`.next()`

Moves to the next entry in the dictionary.

Returns true on success, false if the end-of-data has been reached.

`.prev()`

Moves to the previous entry in the dictionary.

Returns true on success, false if the end-of-data has been reached.

`.read_value()`

Alias for .read().

`.read_key()`

Reads the key object from the dict to which the cursor currently points.

`.read_item()`

Reads the (key, value) item object from the dict to which the cursor currently points.

`.write()`

Writes the object to the dict under the key to which the cursor currently points.

The new data is not written to disk until the dictionary's current transaction is committed.

2.4.3 BeeStringDictCursor Object Attributes

BeeStringDictCursor objects have the following read-only attributes:

`.key`

Key string which dereferences to the current cursor position.

2.5 BeeFixedLengthStringDict Objects

BeeFixedLengthStringDict objects are on-disk dictionaries which use fixed length string keys as index.

All key strings must have the same length (the keysize). This allows them to also contain embedded \0 characters, which the BeeStringDict does not.

Values must be pickleable and can have arbitrary size.

Since hash collisions cannot occur this dictionary type may have some performance advantages over the standard `BeeDict` dictionary.

2.5.1 Relationship between BeeFixedLengthStringDict and BeeStringDict

BeeFixedLengthStringDict objects inherit from BeeStringDict objects, so they expose the same API as BeeStringDict objects. The only difference is their index, which is a BeeFixedLengthStringIndex for BeeFixedLengthStringDicts.

BeeFixedLengthStringDicts also use BeeStringDictCursors as their cursor implementation.

2.5.2 BeeFixedLengthStringDict Object Constructors

BeeFixedLengthStringDict objects are constructed using:

```
BeeFixedLengthStringDict(name, keysize=10, min_recordsize=0,
   readonly=0, recover=0, autocommit=0, validate=0,
   maxcachesize=None)
```

Create an instance using `name` as base name for the data and index files. Two files will be created: name.dat and name.idx.

`keysize` gives the maximal size of the strings used as index keys. `min_recordsize` is passed to the `BeeStorage` as indicator of the minimum size for data records. `readonly` can be set to true to open the files in read-only mode, preventing any disk modifications.

To open the dictionary in recovery mode, pass a keyword `recover=1`. Then run `.recover()` and reopen using the normal settings. The `AutoRecover()` wrapper can take care of this action for you automatically.

If `autocommit` is true the cache control will do an automatic `.commit()` whenever the transaction log overflows.

If `validate` is true, the dictionary will run a validation check after having successfully opened storage and index. `RecreateIndexError` or `RecoverError` exceptions could be raised in case inconsistencies are found.

`maxcachesize` defines the maximum size of the in-memory transaction cache. It defaults to `MAXCACHESIZE` if not given.

Note that the `keysize` is currently not stored in the dictionary itself -- you'll have to store this information in some other form (this may change in future versions of mxBeeBase).

2.5.3 BeeFixedLengthStringDict Object Methods

BeeFixedLengthStringDict objects have the same methods as BeeStringDict objects. Please see section 2.1.2 BeeDict Object Methods for details.

2.5.4 BeeFixedLengthStringDict Object Attributes

BeeFixedLengthStringDict objects have the same attributes as BeeStringDict objects. Please see section 2.1.3 BeeDict Object Attributes for details.

2.6 Functions

These functions are available in mx.BeeBase:

`AutoRecover(Class,*args,**kws)`

Wrapper that can be used around the dictionary constructors to provide automatic recovery whenever needed (if possible).

Example: `diskdict = AutoRecover(BeeDict.BeeDict, 'test')`

2.7 Constants

These constants are available:

`Error`

Baseclass for errors related to this module. It is a subclass of Python's `StandardError`.

`RecreateIndexError`

This error is raised in case the index for a dictionary was not found and/or needs to be recreated by running recovery. It is a subclass of Error.

`RecoverError`

This error is raised in case the storage for a dictionary was found to be in an inconsistent state. It is a subclass of Error.

`FirstKey`

Special index key object representing the key of the first entry in the index (B+Tree's sort their data).

`LastKey`

Special index key object representing the key of the last entry in the index (B+Tree's sort their data).

`MAXCACHESIZE = 1000`

Default value for the maximum size of the in-memory transaction cache used by BeeDict/BeeStringDict objects.

3. mx.BeeBase.BeeStorage Module

The BeeStorage module provides an on-disk storage format that supports:

- locking, ie. it can be used from multiple processes
- caching, for improved performance
- easy data recovery

Please refer to the source code in mx/BeeBase/BeeStorage.py.

Note:
If you just want to use an on-disk dictionary, you don't need to access the BeeStorage module directly.

4. mx.BeeBase.BeeIndex Module

The BeeIndex module implements fast B+Tree indexes in C. It currently provides these index types which all map different data types to address integers:

- BeeStringIndex – keys are size limited strings (may not contain \0 characters)
- BeeFixedLengthStringIndex – keys are fixed size strings (may contain \0 characters)
- BeeIntegerIndex – keys are Python integers
- BeeFloatIndex – keys are Python floats

Please refer to the source code for details.

> **Note:**
> If you just want to use an on-disk dictionary, you don't need to access the BeeIndex module directly.

5. Examples of Use

Here is a very simple one:

```
from mx.BeeBase import BeeDict

# Here is a very simple file based string dictionary:
d = BeeDict.BeeStringDict('BeeStringDict.example', keysize=10)
d['Marc'] = 'Sveta'
d['Thorsten'] = 'Petra'
d['Christian'] = 'Silvia'
d.commit()
print d.values()
d.close()
```

More examples will eventually appear in the Examples subdirectory of the package. The `mxBeeBase/test.py` test script also provides a few examples which might be handy at times.

6. Package Structure

```
[BeeBase]
        Doc/
        [mxBeeBase]
                test.py
        BeeBase.py
        BeeDict.py
        BeeIndex.py
        BeeStorage.py
        showBeeDict.py
```

Entries enclosed in brackets are packages (i.e. they are directories that include a `__init__.py` file). Ones without brackets are just simple subdirectories that are not accessible via `import`.

The package imports all symbols from the BeeBase sub module which in turn imports the extension module, so you only need to '`import BeeBase`' to start working.

7. Support

eGenix.com is providing commercial support for this package. If you are interested in receiving information about this service please see the *eGenix.com Support Conditions*.

8. Copyright & License

Parts of this package are based on an ANSI C implementation of a B+Tree implementation written by Thomas Nieman, Portland, Oregon. The files in question are btr.c and btr.h which were heavily modified for the purpose of inclusion in this package by the above author.

The original files were extracted from btr.c -- an ANSI C implementation included in the source code distribution of

SORTING AND SEARCHING ALGORITHMS: A COOKBOOK

by THOMAS NIEMANN Portland, Oregon

home: *http://epaperpress.com/*

From the above cookbook:

> Permission to reproduce this document, in whole or in part, is given provided the original web site listed below is referenced, and no additional restrictions apply. Source code, when part of a software project, may be used freely without reference to the author.

By downloading, copying, installing or otherwise using the software, you agree to be bound by the terms and conditions of the following *eGenix.com Public License Agreement*.

EGENIX.COM PUBLIC LICENSE AGREEMENT

Version 1.1.0

This license agreement is based on the Python CNRI License Agreement, a widely accepted open-source license.

1. Introduction

This "License Agreement" is between eGenix.com Software, Skills and Services GmbH ("eGenix.com"), having an office at Pastor-Loeh-Str. 48, D-40764 Langenfeld, Germany, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. License

Subject to the terms and conditions of this eGenix.com Public License Agreement, eGenix.com hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the eGenix.com Public License Agreement is retained in the Software, or in any derivative version of the Software prepared by Licensee.

3. NO WARRANTY

eGenix.com is making the Software available to Licensee on an "AS IS" basis. SUBJECT TO ANY STATUTORY WARRANTIES WHICH CAN NOT BE EXCLUDED, EGENIX.COM MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, EGENIX.COM MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. LIMITATION OF LIABILITY

EGENIX.COM SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS) AS A RESULT OF USING, MODIFYING OR

DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE EXCLUSION OR LIMITATION MAY NOT APPLY TO LICENSEE.

5. Termination

This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. Third Party Rights

Any software or documentation in source or binary form provided along with the Software that is associated with a separate license agreement is licensed to Licensee under the terms of that license agreement. This License Agreement does not apply to those portions of the Software. Copies of the third party licenses are included in the Software Distribution.

7. General

Nothing in this License Agreement affects any statutory rights of consumers that cannot be waived or limited by contract.

Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between eGenix.com and Licensee.

If any provision of this License Agreement shall be unlawful, void, or for any reason unenforceable, such provision shall be modified to the extent necessary to render it enforceable without losing its intent, or, if no such modification is possible, be severed from this License Agreement and shall not affect the validity and enforceability of the remaining provisions of this License Agreement.

This License Agreement shall be governed by and interpreted in all respects by the law of Germany, excluding conflict of law provisions. It shall not be governed by the United Nations Convention on Contracts for International Sale of Goods.

This License Agreement does not grant permission to use eGenix.com trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party.

The controlling language of this License Agreement is English. If Licensee has received a translation into another language, it has been provided for Licensee's convenience only.

8. Agreement

By downloading, copying, installing or otherwise using the Software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

For question regarding this License Agreement, please write to:

eGenix.com Software, Skills and Services GmbH

Pastor-Loeh-Str. 48

D-40764 Langenfeld

Germany