Fundamental Specifications of Tokyo Cabinet Version 1

Last Update: Thu, 05 Aug 2010 15:05:11 +0900

Table of Contents

1. Introduction
2. Features
3. Installation
4. The Utility API
5. The Hash Database API
6. The B+ Tree Database API
7. The Fixed-length Database API
8. The Table Database API
9. The Abstract Database API
10. File Format
11. License

Introduction

Tokyo Cabinet is a library of routines for managing a database. The database is a simple data file containing records, each is a pair of a key and a value. Every key and value is serial bytes with variable length. Both binary data and character string can be used as a key and a value. There is neither concept of data tables nor data types. Records are organized in hash table, B+ tree, or fixed-length array.

As for database of hash table, each key must be unique within a database, so it is impossible to store two or more records with a key overlaps. The following access methods are provided to the database: storing a record with a key and a value, deleting a record by a key, retrieving a record by a key. Moreover, traversal access to every key are provided, although the order is arbitrary. These access methods are similar to ones of DBM (or its followers: NDBM and GDBM) library defined in the UNIX standard. Tokyo Cabinet is an alternative for DBM because of its higher performance.

As for database of B+ tree, records whose keys are duplicated can be stored. Access methods of storing, deleting, and retrieving are provided as with the database of hash table. Records are stored in order by a comparison function assigned by a user. It is possible to access each record with the cursor in ascending or descending order. According to this mechanism, forward matching search for strings and range search for integers are realized.

As for database of fixed-length array, records are stored with unique natural numbers. It is impossible to store two or more records with a key overlaps. Moreover, the length of each record is limited by the specified length. Provided operations are the same as ones of hash database.

Table database is also provided as a variant of hash database. Each record is identified by the primary key and has a set of named columns. Although there is no concept of data schema, it is possible to search for records with complex conditions efficiently by using indices of arbitrary columns.

Tokyo Cabinet is written in the C language, and provided as API of C, Perl, Ruby, Java, and Lua. Tokyo Cabinet is available on platforms which have API conforming to C99 and POSIX. Tokyo Cabinet is a free software licensed under the GNU Lesser General Public License.

Features

Tokyo Cabinet is the successor of QDBM and improves time and space efficiency. This section describes the features of Tokyo Cabinet.

The Dinosaur Wing of the DBM Forks

Tokyo Cabinet is developed as the successor of GDBM and QDBM on the following purposes. They are achieved and Tokyo Cabinet replaces conventional DBM products.

- improves **space efficiency** : smaller size of database file.
- improves **time efficiency** : faster processing speed.
- improves **parallelism** : higher performance in multi-thread environment.
- improves **usability** : simplified API.
- improves **robustness** : database file is not corrupted even under catastrophic situation.
- supports **64-bit architecture** : enormous memory space and database file are available.

As with QDBM, the following three restrictions of traditional DBM: a process can handle only one database, the size of a key and a value is bounded, a database file is sparse, are cleared. Moreover, the following three restrictions of QDBM: the size of a database file is limited to 2GB, environments with different byte orders can not share a database file, only one thread can search a database at the same time, are cleared.

Tokyo Cabinet runs very fast. For example, elapsed time to store 1 million records is 0.7 seconds for hash database, and 1.6 seconds for B+ tree database. Moreover, the size of database of Tokyo Cabinet is very small. For example, overhead for a record is 16 bytes for hash database, and 5 bytes for B+ tree database. Furthermore, scalability of Tokyo Cabinet is great. The database size can be up to 8EB (9.22e18 bytes).

Effective Implementation of Hash Database

Tokyo Cabinet uses hash algorithm to retrieve records. If a bucket array has sufficient number of elements, the time complexity of retrieval is "O(1)". That is, time required for retrieving a record is constant, regardless of the scale of a database. It is also the same about storing and deleting. Collision of hash values is managed by separate chaining. Data structure of the chains is binary search tree. Even if a bucket array has unusually scarce elements, the time complexity of retrieval is "O(log n)".

Tokyo Cabinet attains improvement in retrieval by loading RAM with the whole of a bucket array. If a bucket array is on RAM, it is possible to access a region of a target record by about one path of file operations. A bucket array saved in a file is not read into RAM with the `read' call but directly mapped to RAM with the `mmap' call. Therefore, preparation time on connecting to a database is very short, and two or more processes can share the same memory map.

If the number of elements of a bucket array is about half of records stored within a database, although it depends on characteristic of the input, the probability of collision of hash values is about 56.7% (36.8% if the

same, 21.3% if twice, 11.5% if four times, 6.0% if eight times). In such case, it is possible to retrieve a record by two or less paths of file operations. If it is made into a performance index, in order to handle a database containing one million of records, a bucket array with half a million of elements is needed. The size of each element is 4 bytes. That is, if 2M bytes of RAM is available, a database containing one million records can be handled.

Traditional DBM provides two modes of the storing operations: "insert" and "replace". In the case a key overlaps an existing record, the insert mode keeps the existing value, while the replace mode transposes it to the specified value. In addition to the two modes, Tokyo Cabinet provides "concatenate" mode. In the mode, the specified value is concatenated at the end of the existing value and stored. This feature is useful when adding an element to a value as an array.

Generally speaking, while succession of updating, fragmentation of available regions occurs, and the size of a database grows rapidly. Tokyo Cabinet deal with this problem by coalescence of dispensable regions and reuse of them. When overwriting a record with a value whose size is greater than the existing one, it is necessary to remove the region to another position of the file. Because the time complexity of the operation depends on the size of the region of a record, extending values successively is inefficient. However, Tokyo Cabinet deal with this problem by alignment. If increment can be put in padding, it is not necessary to remove the region.

The "free block pool" to reuse dispensable regions efficiently is also implemented. It keeps a list of dispensable regions and reuse the "best fit" region, that is the smallest region in the list, when a new block is requested. Because fragmentation is inevitable even then, two kinds of optimization (defragmentation) mechanisms are implemented. The first is called static optimization which deploys all records into another file and then writes them back to the original file at once. The second is called dynamic optimization which gathers up dispensable regions by replacing the locations of records and dispensable regions gradually.

Useful Implementation of B+ Tree Database

Although B+ tree database is slower than hash database, it features ordering access to each record. The order can be assigned by users. Records of B+ tree are sorted and arranged in logical pages. Sparse index organized in B tree that is multiway balanced tree are maintained for each page. Thus, the time complexity of retrieval and so on is "O(log n)". Cursor is provided to access each record in order. The cursor can jump to a position specified by a key and can step forward or backward from the current position. Because each page is arranged as double linked list, the time complexity of stepping cursor is "O(1)".

B+ tree database is implemented, based on the above hash database. Because each page of B+ tree is stored as each record of hash database, B+ tree database inherits efficiency of storage management of hash database. Because the header of each record is smaller and alignment of each page is adjusted according to the page size, in most cases, the size of database file is cut by half compared to one of hash database. Although operation of many pages are required to update B+ tree, Tokyo Cabinet expedites the process by caching pages and reducing file operations. In most cases, because whole of the sparse index is cached on memory, it is possible to retrieve a record by one or less path of file operations.

Each pages of B+ tree can be stored with compressed. Two compression method; Deflate of ZLIB and Block Sorting of BZIP2, are supported. Because each record in a page has similar patterns, high efficiency of compression is expected due to the Lempel-Ziv or the BWT algorithms. In case handling text data, the size of a database is reduced to about 25%. If the scale of a database is large and disk I/O is the bottleneck, featuring compression makes the processing speed improved to a large extent.

Naive Implementation of Fixed-length Database

Fixed-length database has restrictions that each key should be a natural number and that the length of each value is limited. However, time efficiency and space efficiency are higher than the other data structures as long as the use case is within the restriction.

Because the whole region of the database is mapped on memory by the `mmap' call and referred as a multidimensional array, the overhead related to the file I/O is minimized. Due to this simple structure, fixed-length database works faster than hash database, and its concurrency in multi-thread environment is prominent.

The size of the database is proportional to the range of keys and the limit size of each value. That is, the smaller the range of keys is or the smaller the length of each value is, the higher the space efficiency is. For example, if the maximum key is 1000000 and the limit size of the value is 100 bytes, the size of the database will be about 100MB. Because regions around referred records are only loaded on the RAM, you can increase the size of the database to the size of the virtual memory.

Flexible Implementation of Table Database

Table database does not express simple key/value structure but expresses a structure like a table of relational database. Each record is identified by the primary key and has a set of multiple columns named with arbitrary strings. For example, a stuff in your company can be expressed by a record identified by the primary key of the employee ID number and structured by columns of his name, division, salary, and so on. Unlike relational database, table database does not need to define any data schema and can contain records of various structures different from each other.

Table database supports query functions with not only the primary key but also with conditions about arbitrary columns. Each column condition is composed of the name of a column and a condition expression. Operators of full matching, forward matching, regular expression matching, and so on are provided for the string type. Operators of full matching, range matching and so on are provided for the number type. Operators for tag search and full-text search are also provided. A query can contain multiple conditions for logical intersection. Search by multiple queries for logical union is also available. The order of the result set can be specified as the ascending or descending order of strings or numbers.

You can create indices for arbitrary columns to improve performance of search and sorting. Although columns do not have data types, indices have types for strings or numbers. Inverted indices for space separated tokens and character N-gram tokens are also supported. The query optimizer uses indices in suitable way according to each query. Indices are implemented as different files of B+ tree database.

Practical Functionality

Databases on the filesystem feature transaction mechanisms. It is possible to commit a series of operations between the beginning and the end of the transaction in a lump, or to abort the transaction and perform rollback to the state before the transaction. Two isolation levels are supported; serializable and read uncommitted. Durability is secured by write ahead logging and shadow paging.

Tokyo Cabinet provides two modes to connect to a database: "reader" and "writer". A reader can perform retrieving but neither storing nor deleting. A writer can perform all access methods. Exclusion control between processes is performed when connecting to a database by file locking. While a writer is connected to a database, neither readers nor writers can be connected. While a reader is connected to a database, other readers can be connect, but writers can not. According to this mechanism, data consistency is guaranteed with simultaneous connections in multitasking environment.

Functions of API of Tokyo cabinet are reentrant and available in multi-thread environment. Discrete database object can be operated in parallel entirely. For simultaneous operations of the same database object, read-write lock is used for exclusion control. That is, while a writing thread is operating the database, other reading threads and writing threads are blocked. However, while a reading thread is operating the database, reading threads are not blocked. The locking granularity of hash database and fixed-length database is per record, and that of the other databases is per file.

Simple but Various Interfaces

Tokyo Cabinet provides simple API based on the object oriented design. Every operation for database is encapsulated and published as lucid methods as `open' (connect), `close' (disconnect), `put' (insert), `out' (remove), `get' (retrieve), and so on. Because the three of hash, B+ tree, and fixed-length array database APIs are very similar with each other, porting an application from one to the other is easy. Moreover, the abstract API is provided to handle these databases with the same interface. Applications of the abstract API can determine the type of the database in runtime.

The utility API is also provided. Such fundamental data structure as list and map are included. And, some useful features; memory pool, string processing, encoding, are also included.

Six kinds of API; the utility API, the hash database API, the B+ tree database API, the fixed-length database API, the table database API, and the abstract database API, are provided for the C language. Command line interfaces are also provided corresponding to each API. They are useful for prototyping, test, and debugging. Except for C, Tokyo Cabinet provides APIs for Perl, Ruby, Java, and Lua. APIs for other languages will hopefully be provided by third party.

In cases that multiple processes access a database at the same time or some processes access a database on a remote host, the remote service is useful. The remote service is composed of a database server and its access library. Applications can access the database server by using the remote database API. The server implements HTTP and the memcached protocol partly so that client programs on almost all platforms can access the server easily.

Installation

This section describes how to install Tokyo Cabinet with the source package. As for a binary package, see its installation manual.

Preparation

Tokyo Cabinet is available on UNIX-like systems. At least, the following environments are supported.

- Linux 2.4 and later (x86-32/x86-64/PowerPC/Alpha/SPARC)
- Mac OS X 10.3 and later (x86-32/x86-64/PowerPC)

gcc 3.1 or later and **make** are required to install Tokyo Cabinet with the source package. They are installed by default on Linux, FreeBSD and so on.

As Tokyo Cabinet depends on the following libraries, install them beforehand.

- zlib : for loss-less data compression. 1.2.3 or later is suggested.
- bzip2 : for loss-less data compression. 1.0.5 or later is suggested.

Installation

When an archive file of Tokyo Cabinet is extracted, change the current working directory to the generated directory and perform installation.

Run the configuration script.

```
./configure
```

Build programs.

```
make
```

Perform self-diagnostic test.

```
make check
```

Install programs. This operation must be carried out by the **root** user.

```
make install
```

Result

When a series of work finishes, the following files will be installed.

```
/usr/local/include/tcutil.h
/usr/local/include/tchdb.h
/usr/local/include/tcbdb.h
/usr/local/include/tcfdb.h
```

```
/usr/local/include/tctdb.h
/usr/local/include/tcadb.h
/usr/local/lib/libtokyocabinet.a
/usr/local/lib/libtokyocabinet.so.x.y.z
/usr/local/lib/libtokyocabinet.so.x
/usr/local/lib/libtokyocabinet.so
/usr/local/lib/pkgconfig/tokyocabinet.pc
/usr/local/bin/tcutest
/usr/local/bin/tcumttest
/usr/local/bin/tcucodec
/usr/local/bin/tchtest
/usr/local/bin/tchmttest
/usr/local/bin/tchmgr
/usr/local/bin/tcbmgr
/usr/local/bin/tcbtest
/usr/local/bin/tcbmttest
/usr/local/bin/tcftest
/usr/local/bin/tcfmttest
/usr/local/bin/tcfmgr
/usr/local/bin/tcttest
/usr/local/bin/tctmttest
/usr/local/bin/tctmgr
/usr/local/bin/tcatest
/usr/local/bin/tcamttest
/usr/local/bin/tcamgr
/usr/local/libexec/tcawmgr.cgi
/usr/local/share/tokyocabinet/...
/usr/local/man/man1/...
/usr/local/man/man3/...
```

Options of Configure

The following options can be specified with `**./configure**'.

--enable-debug : build for debugging. Enable debugging symbols, do not perform optimization, and perform static linking.
--enable-devel : build for development. Enable debugging symbols, perform optimization, and perform dynamic linking.
--enable-profile : build for profiling. Enable profiling symbols, perform optimization, and perform dynamic linking.
--enable-static : build by static linking.
--enable-fastest : build for fastest run.
--enable-off64 : build with 64-bit file offset on 32-bit system.
--enable-swab : build for swapping byte-orders.
--enable-uyield : build for detecting race conditions.
--disable-zlib : build without ZLIB compression.
--disable-bzip : build without BZIP2 compression.
--disable-pthread : build without POSIX thread support.
--disable-shared : avoid to build shared libraries.

`**--prefix**' and other options are also available as with usual UNIX software packages. If you want to install Tokyo Cabinet under `**/usr**' not `**/usr/local**', specify `**--prefix=/usr**'. As well, the library search path does not include `**/usr/local/lib**', it is necessary to set the environment variable `**LD_LIBRARY_PATH**' to include `**/usr/local/lib**' before running applications of Tokyo Cabinet.

How to Use the Library

Tokyo Cabinet provides API of the C language and it is available by programs conforming to the C89 (ANSI C) standard or the C99 standard. As the header files of Tokyo Cabinet are provided as `**tcutil.h**', `**tchdb.h**', `**tcbdb.h**', and `**tcadb.h**', applications should include one or more of them accordingly to use the API. As the library is provided as `**libtokyocabinet.a**' and `**libtokyocabinet.so**' and they depend on `**libz.so**', `**libbz2.so**', `**librt.so**', `**libpthread.so**', `**libm.so**', and `**libc.so**', linker options corresponding to them are required by the build command. The typical build command is the following.

```
gcc -I/usr/local/include tc_example.c -o tc_example \
  -L/usr/local/lib -ltokyocabinet -lz -lbz2 -lrt -lpthread -lm -lc
```

You can also use Tokyo Cabinet in programs written in C++. Because each header is wrapped in C linkage (`**extern "C"**' block), you can simply include them into your C++ programs.

The Utility API

The utility API is a set of routines to handle records on memory easily. Especially, extensible string, array list, hash map, and ordered tree are useful. See `**tcutil.h**' for the entire specification.

Description

To use the utility API, include `**tcutil.h**' and related standard header files. Usually, write the following description near the front of a source file.

```
#include <tcutil.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
```

Objects whose type is pointer to `**TCXSTR**' are used for extensible string. An extensible string object is created with the function `**tcxstrnew**' and is deleted with the function `**tcxstrdel**'. Objects whose type is pointer to `**TCLIST**' are used for array list. A list object is created with the function `**tclistnew**' and is deleted with the function `**tclistdel**'. Objects whose type is pointer to `**TCMAP**' are used for hash map. A map object is created with the function `**tcmapnew**' and is deleted with the function `**tcmapdel**'. Objects whose type is pointer to `**TCTREE**' are used for ordered tree. A tree object is created with the function `**tctreenew**' and is deleted with the function `**tctreedel**'. To avoid memory leak, it is important to delete every object when it is no longer in use.

API of Basic Utilities

The constant `tcversion' is the string containing the version information.

```
extern const char *tcversion;
```

The variable `tcfatalfunc' is the pointer to the call back function for handling a fatal error.

```
extern void (*tcfatalfunc)(const char *);
```

The argument specifies the error message.
The initial value of this variable is `NULL'. If the value is `NULL', the default function is called when a fatal error occurs. A fatal error occurs when memory allocation is failed.

The function `tcmalloc' is used in order to allocate a region on memory.

```
void *tcmalloc(size_t size);
```

`*size*' specifies the size of the region.
The return value is the pointer to the allocated region.
This function handles failure of memory allocation implicitly. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tccalloc' is used in order to allocate a nullified region on memory.

```
void *tccalloc(size_t nmemb, size_t size);
```

`*nmemb*' specifies the number of elements.
`*size*' specifies the size of each element.
The return value is the pointer to the allocated nullified region.
This function handles failure of memory allocation implicitly. Because the region of the return value is allocated with the `calloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcrealloc' is used in order to re-allocate a region on memory.

```
void *tcrealloc(void *ptr, size_t size);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
The return value is the pointer to the re-allocated region.
This function handles failure of memory allocation implicitly. Because the region of the return value is allocated with the `realloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcmemdup' is used in order to duplicate a region on memory.

```
void *tcmemdup(const void *ptr, size_t size);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
The return value is the pointer to the allocated region of the duplicate.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcstrdup' is used in order to duplicate a string on memory.

```
char *tcstrdup(const void *str);
```

`*str*' specifies the string.
The return value is the allocated string equivalent to the specified string.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcfree' is used in order to free a region on memory.

```
void tcfree(void *ptr);
```

`*ptr*' specifies the pointer to the region. If it is `NULL', this function has no effect.
Although this function is just a wrapper of `free' call, this is useful in applications using another package of the `malloc' series.

API of Extensible String

The function `tcxstrnew' is used in order to create an extensible string object.

```
TCXSTR *tcxstrnew(void);
```

The return value is the new extensible string object.

The function `tcxstrnew2' is used in order to create an extensible string object from a character string.

```
TCXSTR *tcxstrnew2(const char *str);
```

`*str*' specifies the string of the initial content.
The return value is the new extensible string object containing the specified string.

The function `tcxstrnew3' is used in order to create an extensible string object with the initial allocation size.

```
TCXSTR *tcxstrnew3(int asiz);
```

`*asiz*' specifies the initial allocation size.
The return value is the new extensible string object.

The function `tcxstrdup' is used in order to copy an extensible string object.

```
TCXSTR *tcxstrdup(const TCXSTR *xstr);
```

`*xstr*' specifies the extensible string object.
The return value is the new extensible string object equivalent to the specified object.

The function `tcxstrdel' is used in order to delete an extensible string object.

```
void tcxstrdel(TCXSTR *xstr);
```

`*xstr*' specifies the extensible string object.
Note that the deleted object and its derivatives can not be used anymore.

The function `tcxstrcat' is used in order to concatenate a region to the end of an extensible string object.

```
void tcxstrcat(TCXSTR *xstr, const void *ptr, int size);
```

`*xstr*' specifies the extensible string object.
`*ptr*' specifies the pointer to the region to be appended.
`*size*' specifies the size of the region.

The function `tcxstrcat2' is used in order to concatenate a character string to the end of an extensible string object.

```
void tcxstrcat2(TCXSTR *xstr, const char *str);
```

`*xstr*' specifies the extensible string object.
`*str*' specifies the string to be appended.

The function `tcxstrptr' is used in order to get the pointer of the region of an extensible string object.

```
const void *tcxstrptr(const TCXSTR *xstr);
```

`*xstr*' specifies the extensible string object.
The return value is the pointer of the region of the object.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string.

The function `tcxstrsize' is used in order to get the size of the region of an extensible string object.

```
int tcxstrsize(const TCXSTR *xstr);
```

`*xstr*' specifies the extensible string object.
The return value is the size of the region of the object.

The function `tcxstrclear' is used in order to clear an extensible string object.

```
void tcxstrclear(TCXSTR *xstr);
```

`*xstr*' specifies the extensible string object.
The internal buffer of the object is cleared and the size is set zero.

The function `tcxstrprintf' is used in order to perform formatted output into an extensible string object.

```
void tcxstrprintf(TCXSTR *xstr, const char *format, ...);
```

`*xstr*' specifies the extensible string object.
`*format*' specifies the printf-like format string. The conversion character `%' can be used with such flag characters as `s', `d', `o', `u', `x', `X', `c', `e', `E', `f', `g', `G', `@', `?', `b', and `%'. `@' works as with `s' but escapes meta characters of XML. `?' works as with `s' but escapes meta characters of URL. `b' converts an integer to the string as binary numbers. The other conversion character work as with each original.
The other arguments are used according to the format string.

The function `tcsprintf' is used in order to allocate a formatted string on memory.

```
char *tcsprintf(const char *format, ...);
```

`*format*' specifies the printf-like format string. The conversion character `%' can be used with such flag characters as `s', `d', `o', `u', `x', `X', `c', `e', `E', `f', `g', `G', `@', `?', `b', and `%'. `@' works as with `s' but escapes meta characters of XML. `?' works as with `s' but escapes meta characters of URL. `b' converts an integer to the string as binary numbers. The other conversion character work as with each original.
The other arguments are used according to the format string.
The return value is the pointer to the region of the result string.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

API of Array List

The function `tclistnew' is used in order to create a list object.

```
TCLIST *tclistnew(void);
```

The return value is the new list object.

The function `tclistnew2' is used in order to create a list object with expecting the number of elements.

```
TCLIST *tclistnew2(int anum);
```

`*anum*' specifies the number of elements expected to be stored in the list.
The return value is the new list object.

The function `tclistnew3' is used in order to create a list object with initial string elements.

```
TCLIST *tclistnew3(const char *str, ...);
```

`*str*' specifies the string of the first element.
The other arguments are other elements. They should be trailed by a `NULL' argument.
The return value is the new list object.

The function `tclistdup' is used in order to copy a list object.

```
TCLIST *tclistdup(const TCLIST *list);
```

`*list*' specifies the list object.
The return value is the new list object equivalent to the specified object.

The function `tclistdel' is used in order to delete a list object.

```
void tclistdel(TCLIST *list);
```

`*list*' specifies the list object.
Note that the deleted object and its derivatives can not be used anymore.

The function `tclistnum' is used in order to get the number of elements of a list object.

```
int tclistnum(const TCLIST *list);
```

`*list*' specifies the list object.
The return value is the number of elements of the list.

The function `tclistval' is used in order to get the pointer to the region of an element of a list object.

```
const void *tclistval(const TCLIST *list, int index, int *sp);
```

`*list*' specifies the list object.
`*index*' specifies the index of the element.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
The return value is the pointer to the region of the value.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. If `index' is equal to or more than the number of elements, the return value is `NULL'.

The function `tclistval2' is used in order to get the string of an element of a list object.

```
const char *tclistval2(const TCLIST *list, int index);
```

`*list*' specifies the list object.
`*index*' specifies the index of the element.
The return value is the string of the value.
If `index' is equal to or more than the number of elements, the return value is `NULL'.

The function `tclistpush' is used in order to add an element at the end of a list object.

```
void tclistpush(TCLIST *list, const void *ptr, int size);
```

`*list*' specifies the list object.
`*ptr*' specifies the pointer to the region of the new element.
`*size*' specifies the size of the region.

The function `tclistpush2' is used in order to add a string element at the end of a list object.

```
void tclistpush2(TCLIST *list, const char *str);
```

`*list*' specifies the list object.
`*str*' specifies the string of the new element.

The function `tclistpop' is used in order to remove an element of the end of a list object.

```
void *tclistpop(TCLIST *list, int *sp);
```

`*list*' specifies the list object.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
The return value is the pointer to the region of the removed element.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. If the list is empty, the return value is `NULL'.

The function `tclistpop2' is used in order to remove a string element of the end of a list object.

```
char *tclistpop2(TCLIST *list);
```

`*list*' specifies the list object.
The return value is the string of the removed element.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. If the list is empty, the return value is `NULL'.

The function `tclistunshift' is used in order to add an element at the top of a list object.

```
void tclistunshift(TCLIST *list, const void *ptr, int size);
```

`*list*' specifies the list object.
`*ptr*' specifies the pointer to the region of the new element.
`*size*' specifies the size of the region.

The function `tclistunshift2' is used in order to add a string element at the top of a list object.

```
void tclistunshift2(TCLIST *list, const char *str);
```

`*list*' specifies the list object.
`*str*' specifies the string of the new element.

The function `tclistshift' is used in order to remove an element of the top of a list object.

```
void *tclistshift(TCLIST *list, int *sp);
```

`*list*' specifies the list object.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
The return value is the pointer to the region of the removed element.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. If the list is empty, the return value is `NULL'.

The function `tclistshift2' is used in order to remove a string element of the top of a list object.

```
char *tclistshift2(TCLIST *list);
```

`*list*' specifies the list object.
The return value is the string of the removed element.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. If the list is empty, the return value is `NULL'.

The function `tclistinsert' is used in order to add an element at the specified location of a list object.

```
void tclistinsert(TCLIST *list, int index, const void *ptr, int size);
```

`*list*' specifies the list object.
`*index*' specifies the index of the new element.
`*ptr*' specifies the pointer to the region of the new element.
`*size*' specifies the size of the region.
If `index' is equal to or more than the number of elements, this function has no effect.

The function `tclistinsert2' is used in order to add a string element at the specified location of a list object.

```
void tclistinsert2(TCLIST *list, int index, const char *str);
```

`*list*' specifies the list object.
`*index*' specifies the index of the new element.
`*str*' specifies the string of the new element.
If `index' is equal to or more than the number of elements, this function has no effect.

The function `tclistremove' is used in order to remove an element at the specified location of a list object.

```
void *tclistremove(TCLIST *list, int index, int *sp);
```

`*list*' specifies the list object.
`*index*' specifies the index of the element to be removed.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
The return value is the pointer to the region of the removed element.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. If `index' is equal to or more than the number of elements, no element is removed and the return value is `NULL'.

The function `tclistremove2' is used in order to remove a string element at the specified location of a list object.

```
char *tclistremove2(TCLIST *list, int index);
```

`*list*' specifies the list object.
`*index*' specifies the index of the element to be removed.
The return value is the string of the removed element.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. If `index' is equal to or more than the number of elements, no element is removed and the return value is `NULL'.

The function `tclistover' is used in order to overwrite an element at the specified location of a list object.

```
void tclistover(TCLIST *list, int index, const void *ptr, int size);
```

`*list*' specifies the list object.
`*index*' specifies the index of the element to be overwritten.
`*ptr*' specifies the pointer to the region of the new content.
`*size*' specifies the size of the new content.
If `index' is equal to or more than the number of elements, this function has no effect.

The function `tclistover2' is used in order to overwrite a string element at the specified location of a list object.

```
void tclistover2(TCLIST *list, int index, const char *str);
```

`*list*' specifies the list object.
`*index*' specifies the index of the element to be overwritten.
`*str*' specifies the string of the new content.
If `index' is equal to or more than the number of elements, this function has no effect.

The function `tclistsort' is used in order to sort elements of a list object in lexical order.

```
void tclistsort(TCLIST *list);
```

`*list*' specifies the list object.

The function `tclistlsearch' is used in order to search a list object for an element using liner search.

```
int tclistlsearch(const TCLIST *list, const void *ptr, int size);
```

`*list*' specifies the list object.
`*ptr*' specifies the pointer to the region of the key.
`*size*' specifies the size of the region.
The return value is the index of a corresponding element or -1 if there is no corresponding element.
If two or more elements correspond, the former returns.

The function `tclistbsearch' is used in order to search a list object for an element using binary search.

```
int tclistbsearch(const TCLIST *list, const void *ptr, int size);
```

`*list*' specifies the list object. It should be sorted in lexical order.
`*ptr*' specifies the pointer to the region of the key.
`*size*' specifies the size of the region.
The return value is the index of a corresponding element or -1 if there is no corresponding element.
If two or more elements correspond, which returns is not defined.

The function `tclistclear' is used in order to clear a list object.

```
void tclistclear(TCLIST *list);
```

`*list*' specifies the list object.
All elements are removed.

The function `tclistdump' is used in order to serialize a list object into a byte array.

```
void *tclistdump(const TCLIST *list, int *sp);
```

`*list*' specifies the list object.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
The return value is the pointer to the region of the result serial region.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tclistload' is used in order to create a list object from a serialized byte array.

```
TCLIST *tclistload(const void *ptr, int size);
```

`*ptr*' specifies the pointer to the region of serialized byte array.
`*size*' specifies the size of the region.
The return value is a new list object.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

API of Hash Map

The function `tcmapnew' is used in order to create a map object.

```
TCMAP *tcmapnew(void);
```

The return value is the new map object.

The function `tcmapnew2' is used in order to create a map object with specifying the number of the buckets.

```
TCMAP *tcmapnew2(uint32_t bnum);
```

`*bnum*' specifies the number of the buckets.
The return value is the new map object.

The function `tcmapnew3' is used in order to create a map object with initial string elements.

```
TCMAP *tcmapnew3(const char *str, ...);
```

`*str*' specifies the string of the first element.
The other arguments are other elements. They should be trailed by a `NULL' argument.
The return value is the new map object.
The key and the value of each record are situated one after the other.

The function `tcmapdup' is used in order to copy a map object.

```
TCMAP *tcmapdup(const TCMAP *map);
```

`*map*' specifies the map object.
The return value is the new map object equivalent to the specified object.

The function `tcmapdel' is used in order to delete a map object.

```
void tcmapdel(TCMAP *map);
```

`*map*' specifies the map object.
Note that the deleted object and its derivatives can not be used anymore.

The function `tcmapput' is used in order to store a record into a map object.

```
void tcmapput(TCMAP *map, const void *kbuf, int ksiz, const void *vbuf, int vsiz);
```

`*map*' specifies the map object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If a record with the same key exists in the map, it is overwritten.

The function `tcmapput2' is used in order to store a string record into a map object.

void tcmapput2(TCMAP *_map_, const char *_kstr_, const char *_vstr_);

`_map_' specifies the map object.
`_kstr_' specifies the string of the key.
`_vstr_' specifies the string of the value.
If a record with the same key exists in the map, it is overwritten.

The function `tcmapputkeep' is used in order to store a new record into a map object.

bool tcmapputkeep(TCMAP *_map_, const void *_kbuf_, int _ksiz_, const void *_vbuf_, int _vsiz_);

`_map_' specifies the map object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_vbuf_' specifies the pointer to the region of the value.
`_vsiz_' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the map, this function has no effect.

The function `tcmapputkeep2' is used in order to store a new string record into a map object.

bool tcmapputkeep2(TCMAP *_map_, const char *_kstr_, const char *_vstr_);

`_map_' specifies the map object.
`_kstr_' specifies the string of the key.
`_vstr_' specifies the string of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the map, this function has no effect.

The function `tcmapputcat' is used in order to concatenate a value at the end of the value of the existing record in a map object.

void tcmapputcat(TCMAP *_map_, const void *_kbuf_, int _ksiz_, const void *_vbuf_, int _vsiz_);

`_map_' specifies the map object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_vbuf_' specifies the pointer to the region of the value.
`_vsiz_' specifies the size of the region of the value.
If there is no corresponding record, a new record is created.

The function `tcmapputcat2' is used in order to concatenate a string value at the end of the value of the existing record in a map object.

void tcmapputcat2(TCMAP *_map_, const char *_kstr_, const char *_vstr_);

`_map_' specifies the map object.
`_kstr_' specifies the string of the key.
`_vstr_' specifies the string of the value.
If there is no corresponding record, a new record is created.

The function `tcmapout' is used in order to remove a record of a map object.

bool tcmapout(TCMAP *_map_, const void *_kbuf_, int _ksiz_);

`_map_' specifies the map object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
If successful, the return value is true. False is returned when no record corresponds to the specified key.

The function `tcmapout2' is used in order to remove a string record of a map object.

bool tcmapout2(TCMAP *_map_, const char *_kstr_);

`_map_' specifies the map object.
`_kstr_' specifies the string of the key.
If successful, the return value is true. False is returned when no record corresponds to the specified key.

The function `tcmapget' is used in order to retrieve a record in a map object.

const void *tcmapget(const TCMAP *_map_, const void *_kbuf_, int _ksiz_, int *_sp_);

`_map_' specifies the map object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_sp_' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the value of the corresponding record. `NULL' is returned when no record corresponds.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string.

The function `tcmapget2' is used in order to retrieve a string record in a map object.

const char *tcmapget2(const TCMAP *_map_, const char *_kstr_);

`_map_' specifies the map object.
`_kstr_' specifies the string of the key.
If successful, the return value is the string of the value of the corresponding record. `NULL' is returned when no record corresponds.

The function `tcmapmove' is used in order to move a record to the edge of a map object.

bool tcmapmove(TCMAP *_map_, const void *_kbuf_, int _ksiz_, bool _head_);

`_map_' specifies the map object.
`_kbuf_' specifies the pointer to the region of a key.
`_ksiz_' specifies the size of the region of the key.
`_head_' specifies the destination which is the head if it is true or the tail if else.
If successful, the return value is true. False is returned when no record corresponds to the specified key.

The function `tcmapmove2' is used in order to move a string record to the edge of a map object.

bool tcmapmove2(TCMAP *_map_, const char *_kstr_, bool _head_);

`_map_' specifies the map object.
`_kstr_' specifies the string of a key.
`_head_' specifies the destination which is the head if it is true or the tail if else.
If successful, the return value is true. False is returned when no record corresponds to the specified key.

The function `tcmapiterinit' is used in order to initialize the iterator of a map object.

```
void tcmapiterinit(TCMAP *map);
```

`*map*' specifies the map object.
The iterator is used in order to access the key of every record stored in the map object.

The function `tcmapiternext' is used in order to get the next key of the iterator of a map object.

```
const void *tcmapiternext(TCMAP *map, int *sp);
```

`*map*' specifies the map object.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the next key, else, it is `NULL'. `NULL' is returned when no record can be fetched from the iterator.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. The order of iteration is assured to be the same as the stored order.

The function `tcmapiternext2' is used in order to get the next key string of the iterator of a map object.

```
const char *tcmapiternext2(TCMAP *map);
```

`*map*' specifies the map object.
If successful, the return value is the pointer to the region of the next key, else, it is `NULL'. `NULL' is returned when no record can be fetched from the iterator.
The order of iteration is assured to be the same as the stored order.

The function `tcmaprnum' is used in order to get the number of records stored in a map object.

```
uint64_t tcmaprnum(const TCMAP *map);
```

`*map*' specifies the map object.
The return value is the number of the records stored in the map object.

The function `tcmapmsiz' is used in order to get the total size of memory used in a map object.

```
uint64_t tcmapmsiz(const TCMAP *map);
```

`*map*' specifies the map object.
The return value is the total size of memory used in a map object.

The function `tcmapkeys' is used in order to create a list object containing all keys in a map object.

```
TCLIST *tcmapkeys(const TCMAP *map);
```

`*map*' specifies the map object.
The return value is the new list object containing all keys in the map object.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcmapvals' is used in order to create a list object containing all values in a map object.

```
TCLIST *tcmapvals(const TCMAP *map);
```

`*map*' specifies the map object.
The return value is the new list object containing all values in the map object.

Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcmapaddint' is used in order to add an integer to a record in a map object.

```
int tcmapaddint(TCMAP *map, const void *kbuf, int ksiz, int num);
```

`*map*' specifies the map object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*num*' specifies the additional value.
The return value is the summation value.
If the corresponding record exists, the value is treated as an integer and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tcmapadddouble' is used in order to add a real number to a record in a map object.

```
double tcmapadddouble(TCMAP *map, const void *kbuf, int ksiz, double num);
```

`*map*' specifies the map object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*num*' specifies the additional value.
The return value is the summation value.
If the corresponding record exists, the value is treated as a real number and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tcmapclear' is used in order to clear a map object.

```
void tcmapclear(TCMAP *map);
```

`*map*' specifies the map object.
All records are removed.

The function `tcmapcutfront' is used in order to remove front records of a map object.

```
void tcmapcutfront(TCMAP *map, int num);
```

`*map*' specifies the map object.
`*num*' specifies the number of records to be removed.

The function `tcmapdump' is used in order to serialize a map object into a byte array.

```
void *tcmapdump(const TCMAP *map, int *sp);
```

`*map*' specifies the map object.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
The return value is the pointer to the region of the result serial region.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcmapload' is used in order to create a map object from a serialized byte array.

```
TCMAP *tcmapload(const void *ptr, int size);
```

`*ptr*' specifies the pointer to the region of serialized byte array.

`*size*' specifies the size of the region.
The return value is a new map object.
Because the object of the return value is created with the function `tcmapnew', it should be deleted with the function `tcmapdel' when it is no longer in use.

API of Ordered Tree

The function `tctreenew' is used in order to create a tree object.

`TCTREE *tctreenew(void);`
The return value is the new tree object.

The function `tctreenew2' is used in order to create a tree object with specifying the custom comparison function.

`TCTREE *tctreenew2(TCCMP *cmp, void *cmpop);`
`*cmp*' specifies the pointer to the custom comparison function. It receives five parameters. The first parameter is the pointer to the region of one key. The second parameter is the size of the region of one key. The third parameter is the pointer to the region of the other key. The fourth parameter is the size of the region of the other key. The fifth parameter is the pointer to the optional opaque object. It returns positive if the former is big, negative if the latter is big, 0 if both are equivalent.
`*cmpop*' specifies an arbitrary pointer to be given as a parameter of the comparison function. If it is not needed, `NULL' can be specified.
The return value is the new tree object.
The default comparison function compares keys of two records by lexical order. The functions `tccmplexical' (dafault), `tccmpdecimal', `tccmpint32', and `tccmpint64' are built-in.

The function `tctreedup' is used in order to copy a tree object.

`TCTREE *tctreedup(const TCTREE *tree);`
`*tree*' specifies the tree object.
The return value is the new tree object equivalent to the specified object.

The function `tctreedel' is used in order to delete a tree object.

`void tctreedel(TCTREE *tree);`
`tree' specifies the tree object.
Note that the deleted object and its derivatives can not be used anymore.

The function `tctreeput' is used in order to store a record into a tree object.

`void tctreeput(TCTREE *tree, const void *kbuf, int ksiz, const void *vbuf, int vsiz);`
`*tree*' specifies the tree object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If a record with the same key exists in the tree, it is overwritten.

The function `tctreeput2' is used in order to store a string record into a tree object.

```
void tctreeput2(TCTREE *tree, const char *kstr, const char *vstr);
```

`*tree*' specifies the tree object.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If a record with the same key exists in the tree, it is overwritten.

The function `tctreeputkeep' is used in order to store a new record into a tree object.

```
bool tctreeputkeep(TCTREE *tree, const void *kbuf, int ksiz, const void *vbuf, int vsiz);
```

`*tree*' specifies the tree object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the tree, this function has no effect.

The function `tctreeputkeep2' is used in order to store a new string record into a tree object.

```
bool tctreeputkeep2(TCTREE *tree, const char *kstr, const char *vstr);
```

`*tree*' specifies the tree object.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the tree, this function has no effect.

The function `tctreeputcat' is used in order to concatenate a value at the end of the value of the existing record in a tree object.

```
void tctreeputcat(TCTREE *tree, const void *kbuf, int ksiz, const void *vbuf, int vsiz);
```

`*tree*' specifies the tree object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If there is no corresponding record, a new record is created.

The function `tctreeputcat2' is used in order to concatenate a string value at the end of the value of the existing record in a tree object.

```
void tctreeputcat2(TCTREE *tree, const char *kstr, const char *vstr);
```

`*tree*' specifies the tree object.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If there is no corresponding record, a new record is created.

The function `tctreeout' is used in order to remove a record of a tree object.

```
bool tctreeout(TCTREE *tree, const void *kbuf, int ksiz);
```

`*tree*' specifies the tree object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
If successful, the return value is true. False is returned when no record corresponds to the specified key.

The function `tctreeout2' is used in order to remove a string record of a tree object.

```
bool tctreeout2(TCTREE *tree, const char *kstr);
```

`*tree*' specifies the tree object.
`*kstr*' specifies the string of the key.
If successful, the return value is true. False is returned when no record corresponds to the specified key.

The function `tctreeget' is used in order to retrieve a record in a tree object.

```
const void *tctreeget(TCTREE *tree, const void *kbuf, int ksiz, int *sp);
```

`*tree*' specifies the tree object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the value of the corresponding record. `NULL' is returned when no record corresponds.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string.

The function `tctreeget2' is used in order to retrieve a string record in a tree object.

```
const char *tctreeget2(TCTREE *tree, const char *kstr);
```

`*tree*' specifies the tree object.
`*kstr*' specifies the string of the key.
If successful, the return value is the string of the value of the corresponding record. `NULL' is returned when no record corresponds.

The function `tctreeiterinit' is used in order to initialize the iterator of a tree object.

```
void tctreeiterinit(TCTREE *tree);
```

`*tree*' specifies the tree object.
The iterator is used in order to access the key of every record stored in the tree object.

The function `tctreeiternext' is used in order to get the next key of the iterator of a tree object.

```
const void *tctreeiternext(TCTREE *tree, int *sp);
```

`*tree*' specifies the tree object.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the next key, else, it is `NULL'. `NULL' is returned when no record can be fetched from the iterator.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. The order of iteration is assured to be ascending of the keys.

The function `tctreeiternext2' is used in order to get the next key string of the iterator of a tree object.

```
const char *tctreeiternext2(TCTREE *tree);
```

`*tree*' specifies the tree object.
If successful, the return value is the pointer to the region of the next key, else, it is `NULL'. `NULL' is returned when no record can be fetched from the iterator.
The order of iteration is assured to be ascending of the keys.

The function `tctreernum' is used in order to get the number of records stored in a tree object.

```
uint64_t tctreernum(const TCTREE *tree);
```

`*tree*' specifies the tree object.
The return value is the number of the records stored in the tree object.

The function `tctreemsiz' is used in order to get the total size of memory used in a tree object.

```
uint64_t tctreemsiz(const TCTREE *tree);
```

`*tree*' specifies the tree object.
The return value is the total size of memory used in a tree object.

The function `tctreekeys' is used in order to create a list object containing all keys in a tree object.

```
TCLIST *tctreekeys(const TCTREE *tree);
```

`*tree*' specifies the tree object.
The return value is the new list object containing all keys in the tree object.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tctreevals' is used in order to create a list object containing all values in a tree object.

```
TCLIST *tctreevals(const TCTREE *tree);
```

`*tree*' specifies the tree object.
The return value is the new list object containing all values in the tree object.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tctreeaddint' is used in order to add an integer to a record in a tree object.

```
int tctreeaddint(TCTREE *tree, const void *kbuf, int ksiz, int num);
```

`*tree*' specifies the tree object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*num*' specifies the additional value.
The return value is the summation value.
If the corresponding record exists, the value is treated as an integer and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tctreeadddouble' is used in order to add a real number to a record in a tree object.

```
double tctreeadddouble(TCTREE *tree, const void *kbuf, int ksiz, double num);
```

`*tree*' specifies the tree object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*num*' specifies the additional value.
The return value is the summation value.
If the corresponding record exists, the value is treated as a real number and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tctreeclear' is used in order to clear a tree object.

```
void tctreeclear(TCTREE *tree);
```

`*tree*' specifies the tree object.
All records are removed.

The function `tctreecutfringe' is used in order to remove fringe records of a tree object.

```
void tctreecutfringe(TCTREE *tree, int num);
```

`*tree*' specifies the tree object.
`*num*' specifies the number of records to be removed.

The function `tctreedump' is used in order to serialize a tree object into a byte array.

```
void *tctreedump(const TCTREE *tree, int *sp);
```

`*tree*' specifies the tree object.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
The return value is the pointer to the region of the result serial region.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tctreeload' is used in order to create a tree object from a serialized byte array.

```
TCTREE *tctreeload(const void *ptr, int size, TCCMP cmp, void *cmpop);
```

`*ptr*' specifies the pointer to the region of serialized byte array.
`*size*' specifies the size of the region.
`*cmp*' specifies the pointer to the custom comparison function.
`*cmpop*' specifies an arbitrary pointer to be given as a parameter of the comparison function.
If it is not needed, `NULL' can be specified.
The return value is a new tree object.
Because the object of the return value is created with the function `tctreenew', it should be deleted with the function `tctreedel' when it is no longer in use.

API of On-memory Hash Database

The function `tcmdbnew' is used in order to create an on-memory hash database object.

```
TCMDB *tcmdbnew(void);
```

The return value is the new on-memory hash database object.
The object can be shared by plural threads because of the internal mutex.

The function `tcmdbnew2' is used in order to create an on-memory hash database object with specifying the

number of the buckets.

TCMDB *tcmdbnew2(uint32_t *bnum*);

`*bnum*' specifies the number of the buckets.
The return value is the new on-memory hash database object.
The object can be shared by plural threads because of the internal mutex.

The function `tcmdbdel' is used in order to delete an on-memory hash database object.

void tcmdbdel(TCMDB **mdb*);

`*mdb*' specifies the on-memory hash database object.

The function `tcmdbput' is used in order to store a record into an on-memory hash database object.

void tcmdbput(TCMDB **mdb*, const void **kbuf*, int *ksiz*, const void **vbuf*, int *vsiz*);

`*mdb*' specifies the on-memory hash database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If a record with the same key exists in the database, it is overwritten.

The function `tcmdbput2' is used in order to store a string record into an on-memory hash database object.

void tcmdbput2(TCMDB **mdb*, const char **kstr*, const char **vstr*);

`*mdb*' specifies the on-memory hash database object.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If a record with the same key exists in the database, it is overwritten.

The function `tcmdbputkeep' is used in order to store a new record into an on-memory hash database object.

bool tcmdbputkeep(TCMDB **mdb*, const void **kbuf*, int *ksiz*, const void **vbuf*, int *vsiz*);

`*mdb*' specifies the on-memory hash database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tcmdbputkeep2' is used in order to store a new string record into an on-memory hash database object.

bool tcmdbputkeep2(TCMDB **mdb*, const char **kstr*, const char **vstr*);

`*mdb*' specifies the on-memory hash database object.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If successful, the return value is true, else, it is false.

If a record with the same key exists in the database, this function has no effect.

The function `tcmdbputcat' is used in order to concatenate a value at the end of the existing record in an on-memory hash database.

void tcmdbputcat(TCMDB *_mdb_, const void *_kbuf_, int _ksiz_, const void *_vbuf_, int _vsiz_);

`_mdb_' specifies the on-memory hash database object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_vbuf_' specifies the pointer to the region of the value.
`_vsiz_' specifies the size of the region of the value.
If there is no corresponding record, a new record is created.

The function `tcmdbputcat2' is used in order to concatenate a string at the end of the existing record in an on-memory hash database.

void tcmdbputcat2(TCMDB *_mdb_, const char *_kstr_, const char *_vstr_);

`_mdb_' specifies the on-memory hash database object.
`_kstr_' specifies the string of the key.
`_vstr_' specifies the string of the value.
If there is no corresponding record, a new record is created.

The function `tcmdbout' is used in order to remove a record of an on-memory hash database object.

bool tcmdbout(TCMDB *_mdb_, const void *_kbuf_, int _ksiz_);

`_mdb_' specifies the on-memory hash database object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
If successful, the return value is true. False is returned when no record corresponds to the specified key.

The function `tcmdbout2' is used in order to remove a string record of an on-memory hash database object.

bool tcmdbout2(TCMDB *_mdb_, const char *_kstr_);

`_mdb_' specifies the on-memory hash database object.
`_kstr_' specifies the string of the key.
If successful, the return value is true. False is returned when no record corresponds to the specified key.

The function `tcmdbget' is used in order to retrieve a record in an on-memory hash database object.

void *tcmdbget(TCMDB *_mdb_, const void *_kbuf_, int _ksiz_, int *_sp_);

`_mdb_' specifies the on-memory hash database object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_sp_' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the value of the corresponding record. `NULL' is returned when no record corresponds.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the

`malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcmdbget2' is used in order to retrieve a string record in an on-memory hash database object.

```
char *tcmdbget2(TCMDB *mdb, const char *kstr);
```

`*mdb*' specifies the on-memory hash database object.
`*kstr*' specifies the string of the key.
If successful, the return value is the string of the value of the corresponding record. `NULL' is returned when no record corresponds.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcmdbvsiz' is used in order to get the size of the value of a record in an on-memory hash database object.

```
int tcmdbvsiz(TCMDB *mdb, const void *kbuf, int ksiz);
```

`*mdb*' specifies the on-memory hash database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tcmdbvsiz2' is used in order to get the size of the value of a string record in an on-memory hash database object.

```
int tcmdbvsiz2(TCMDB *mdb, const char *kstr);
```

`*mdb*' specifies the on-memory hash database object.
`*kstr*' specifies the string of the key.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tcmdbiterinit' is used in order to initialize the iterator of an on-memory hash database object.

```
void tcmdbiterinit(TCMDB *mdb);
```

`*mdb*' specifies the on-memory hash database object.
The iterator is used in order to access the key of every record stored in the on-memory hash database.

The function `tcmdbiternext' is used in order to get the next key of the iterator of an on-memory hash database object.

```
void *tcmdbiternext(TCMDB *mdb, int *sp);
```

`*mdb*' specifies the on-memory hash database object.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the next key, else, it is `NULL'. `NULL' is returned when no record can be fetched from the iterator.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. The order of iteration is assured to be the same as the stored order.

The function `tcmdbiternext2' is used in order to get the next key string of the iterator of an on-memory hash database object.

```
char *tcmdbiternext2(TCMDB *mdb);
```

`*mdb*' specifies the on-memory hash database object.

If successful, the return value is the pointer to the region of the next key, else, it is `NULL'. `NULL' is returned when no record can be fetched from the iterator.

Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. The order of iteration is assured to be the same as the stored order.

The function `tcmdbfwmkeys' is used in order to get forward matching keys in an on-memory hash database object.

```
TCLIST *tcmdbfwmkeys(TCMDB *mdb, const void *pbuf, int psiz, int max);
```

`*mdb*' specifies the on-memory hash database object.

`*pbuf*' specifies the pointer to the region of the prefix.

`*psiz*' specifies the size of the region of the prefix.

`*max*' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.

The return value is a list object of the corresponding keys. This function does never fail. It returns an empty list even if no key corresponds.

Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use. Note that this function may be very slow because every key in the database is scanned.

The function `tcmdbfwmkeys2' is used in order to get forward matching string keys in an on-memory hash database object.

```
TCLIST *tcmdbfwmkeys2(TCMDB *mdb, const char *pstr, int max);
```

`*mdb*' specifies the on-memory hash database object.

`*pstr*' specifies the string of the prefix.

`*max*' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.

The return value is a list object of the corresponding keys. This function does never fail. It returns an empty list even if no key corresponds.

Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use. Note that this function may be very slow because every key in the database is scanned.

The function `tcmdbrnum' is used in order to get the number of records stored in an on-memory hash database object.

```
uint64_t tcmdbrnum(TCMDB *mdb);
```

`*mdb*' specifies the on-memory hash database object.

The return value is the number of the records stored in the database.

The function `tcmdbmsiz' is used in order to get the total size of memory used in an on-memory hash database object.

```
uint64_t tcmdbmsiz(TCMDB *mdb);
```

`*mdb*' specifies the on-memory hash database object.
The return value is the total size of memory used in the database.

The function `tcmdbaddint' is used in order to add an integer to a record in an on-memory hash database object.

```
int tcmdbaddint(TCMDB *mdb, const void *kbuf, int ksiz, int num);
```

`*mdb*' specifies the on-memory hash database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*num*' specifies the additional value.
The return value is the summation value.
If the corresponding record exists, the value is treated as an integer and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tcmdbadddouble' is used in order to add a real number to a record in an on-memory hash database object.

```
double tcmdbadddouble(TCMDB *mdb, const void *kbuf, int ksiz, double num);
```

`*mdb*' specifies the on-memory hash database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*num*' specifies the additional value.
The return value is the summation value.
If the corresponding record exists, the value is treated as a real number and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tcmdbvanish' is used in order to clear an on-memory hash database object.

```
void tcmdbvanish(TCMDB *mdb);
```

`*mdb*' specifies the on-memory hash database object.
All records are removed.

The function `tcmdbcutfront' is used in order to remove front records of an on-memory hash database object.

```
void tcmdbcutfront(TCMDB *mdb, int num);
```

`*mdb*' specifies the on-memory hash database object.
`*num*' specifies the number of records to be removed.

API of On-memory Tree Database

The function `tcndbnew' is used in order to create an on-memory tree database object.

```
TCNDB *tcndbnew(void);
```

The return value is the new on-memory tree database object.
The object can be shared by plural threads because of the internal mutex.

The function `tcndbnew2' is used in order to create an on-memory tree database object with specifying the

custom comparison function.

TCNDB *tcndbnew2(TCCMP *cmp*, void **cmpop*);

`*cmp*' specifies the pointer to the custom comparison function.
`*cmpop*' specifies an arbitrary pointer to be given as a parameter of the comparison function. If it is not needed, `NULL' can be specified.
The return value is the new on-memory tree database object.
The default comparison function compares keys of two records by lexical order. The functions `tccmplexical' (dafault), `tccmpdecimal', `tccmpint32', and `tccmpint64' are built-in. The object can be shared by plural threads because of the internal mutex.

The function `tcndbdel' is used in order to delete an on-memory tree database object.

void tcndbdel(TCNDB **ndb*);

`*ndb*' specifies the on-memory tree database object.

The function `tcndbput' is used in order to store a record into an on-memory tree database object.

void tcndbput(TCNDB **ndb*, const void **kbuf*, int *ksiz*, const void **vbuf*, int *vsiz*);

`*ndb*' specifies the on-memory tree database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If a record with the same key exists in the database, it is overwritten.

The function `tcndbput2' is used in order to store a string record into an on-memory tree database object.

void tcndbput2(TCNDB **ndb*, const char **kstr*, const char **vstr*);

`*ndb*' specifies the on-memory tree database object.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If a record with the same key exists in the database, it is overwritten.

The function `tcndbputkeep' is used in order to store a new record into an on-memory tree database object.

bool tcndbputkeep(TCNDB **ndb*, const void **kbuf*, int *ksiz*, const void **vbuf*, int *vsiz*);

`*ndb*' specifies the on-memory tree database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tcndbputkeep2' is used in order to store a new string record into an on-memory tree database object.

bool tcndbputkeep2(TCNDB **ndb*, const char **kstr*, const char **vstr*);

`*ndb*' specifies the on-memory tree database object.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tcndbputcat' is used in order to concatenate a value at the end of the existing record in an on-memory tree database.

void tcndbputcat(TCNDB *_ndb_, const void *_kbuf_, int _ksiz_, const void *_vbuf_, int _vsiz_);

`*ndb*' specifies the on-memory tree database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If there is no corresponding record, a new record is created.

The function `tcndbputcat2' is used in order to concatenate a string at the end of the existing record in an on-memory tree database.

void tcndbputcat2(TCNDB *_ndb_, const char *_kstr_, const char *_vstr_);

`*ndb*' specifies the on-memory tree database object.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If there is no corresponding record, a new record is created.

The function `tcndbout' is used in order to remove a record of an on-memory tree database object.

bool tcndbout(TCNDB *_ndb_, const void *_kbuf_, int _ksiz_);

`*ndb*' specifies the on-memory tree database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
If successful, the return value is true. False is returned when no record corresponds to the specified key.

The function `tcndbout2' is used in order to remove a string record of an on-memory tree database object.

bool tcndbout2(TCNDB *_ndb_, const char *_kstr_);

`*ndb*' specifies the on-memory tree database object.
`*kstr*' specifies the string of the key.
If successful, the return value is true. False is returned when no record corresponds to the specified key.

The function `tcndbget' is used in order to retrieve a record in an on-memory tree database object.

void *tcndbget(TCNDB *_ndb_, const void *_kbuf_, int _ksiz_, int *_sp_);

`*ndb*' specifies the on-memory tree database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.

If successful, the return value is the pointer to the region of the value of the corresponding record. `NULL' is returned when no record corresponds.

Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcndbget2' is used in order to retrieve a string record in an on-memory tree database object.

```
char *tcndbget2(TCNDB *ndb, const char *kstr);
```

`*ndb*' specifies the on-memory tree database object.

`*kstr*' specifies the string of the key.

If successful, the return value is the string of the value of the corresponding record. `NULL' is returned when no record corresponds.

Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcndbvsiz' is used in order to get the size of the value of a record in an on-memory tree database object.

```
int tcndbvsiz(TCNDB *ndb, const void *kbuf, int ksiz);
```

`*ndb*' specifies the on-memory tree database object.

`*kbuf*' specifies the pointer to the region of the key.

`*ksiz*' specifies the size of the region of the key.

If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tcndbvsiz2' is used in order to get the size of the value of a string record in an on-memory tree database object.

```
int tcndbvsiz2(TCNDB *ndb, const char *kstr);
```

`*ndb*' specifies the on-memory tree database object.

`*kstr*' specifies the string of the key.

If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tcndbiterinit' is used in order to initialize the iterator of an on-memory tree database object.

```
void tcndbiterinit(TCNDB *ndb);
```

`*ndb*' specifies the on-memory tree database object.

The iterator is used in order to access the key of every record stored in the on-memory database.

The function `tcndbiternext' is used in order to get the next key of the iterator of an on-memory tree database object.

```
void *tcndbiternext(TCNDB *ndb, int *sp);
```

`*ndb*' specifies the on-memory tree database object.

`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.

If successful, the return value is the pointer to the region of the next key, else, it is `NULL'. `NULL' is returned when no record can be fetched from the iterator.

Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the

`malloc' call, it should be released with the `free' call when it is no longer in use. The order of iteration is assured to be the same as the stored order.

The function `tcndbiternext2' is used in order to get the next key string of the iterator of an on-memory tree database object.

```
char *tcndbiternext2(TCNDB *ndb);
```

`*ndb*' specifies the on-memory tree database object.
If successful, the return value is the pointer to the region of the next key, else, it is `NULL'. `NULL' is returned when no record can be fetched from the iterator.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. The order of iteration is assured to be the same as the stored order.

The function `tcndbfwmkeys' is used in order to get forward matching keys in an on-memory tree database object.

```
TCLIST *tcndbfwmkeys(TCNDB *ndb, const void *pbuf, int psiz, int max);
```

`*ndb*' specifies the on-memory tree database object.
`*pbuf*' specifies the pointer to the region of the prefix.
`*psiz*' specifies the size of the region of the prefix.
`*max*' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.
The return value is a list object of the corresponding keys. This function does never fail. It returns an empty list even if no key corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcndbfwmkeys2' is used in order to get forward matching string keys in an on-memory tree database object.

```
TCLIST *tcndbfwmkeys2(TCNDB *ndb, const char *pstr, int max);
```

`*ndb*' specifies the on-memory tree database object.
`*pstr*' specifies the string of the prefix.
`*max*' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.
The return value is a list object of the corresponding keys. This function does never fail. It returns an empty list even if no key corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcndbrnum' is used in order to get the number of records stored in an on-memory tree database object.

```
uint64_t tcndbrnum(TCNDB *ndb);
```

`*ndb*' specifies the on-memory tree database object.
The return value is the number of the records stored in the database.

The function `tcndbmsiz' is used in order to get the total size of memory used in an on-memory tree database object.

uint64_t tcndbmsiz(TCNDB *ndb);

`ndb' specifies the on-memory tree database object.

The return value is the total size of memory used in the database.

The function `tcndbaddint' is used in order to add an integer to a record in an on-memory tree database object.

int tcndbaddint(TCNDB *ndb, const void *kbuf, int ksiz, int num);

`ndb' specifies the on-memory tree database object.

`kbuf' specifies the pointer to the region of the key.

`ksiz' specifies the size of the region of the key.

`num' specifies the additional value.

The return value is the summation value.

If the corresponding record exists, the value is treated as an integer and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tcndbadddouble' is used in order to add a real number to a record in an on-memory tree database object.

double tcndbadddouble(TCNDB *ndb, const void *kbuf, int ksiz, double num);

`ndb' specifies the on-memory tree database object.

`kbuf' specifies the pointer to the region of the key.

`ksiz' specifies the size of the region of the key.

`num' specifies the additional value.

The return value is the summation value.

If the corresponding record exists, the value is treated as a real number and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tcndbvanish' is used in order to clear an on-memory tree database object.

void tcndbvanish(TCNDB *ndb);

`ndb' specifies the on-memory tree database object.

All records are removed.

The function `tcndbcutfringe' is used in order to remove fringe records of an on-memory tree database object.

void tcndbcutfringe(TCNDB *ndb, int num);

`ndb' specifies the on-memory tree database object.

`num' specifies the number of records to be removed.

API of Memory Pool

The function `tcmpoolnew' is used in order to create a memory pool object.

TCMPOOL *tcmpoolnew(void);

The return value is the new memory pool object.

The function `tcmpooldel' is used in order to delete a memory pool object.

void tcmpooldel(TCMPOOL *_mpool_);

`_mpool_' specifies the memory pool object.
Note that the deleted object and its derivatives can not be used anymore.

The function `tcmpoolpush' is used in order to relegate an arbitrary object to a memory pool object.

void *tcmpoolpush(TCMPOOL *_mpool_, void *_ptr_, void (*_del_)(void *));

`_mpool_' specifies the memory pool object.
`_ptr_' specifies the pointer to the object to be relegated. If it is `NULL', this function has no effect.
`_del_' specifies the pointer to the function to delete the object.
The return value is the pointer to the given object.
This function assures that the specified object is deleted when the memory pool object is deleted.

The function `tcmpoolpushptr' is used in order to relegate an allocated region to a memory pool object.

void *tcmpoolpushptr(TCMPOOL *_mpool_, void *_ptr_);

`_mpool_' specifies the memory pool object.
`_ptr_' specifies the pointer to the region to be relegated. If it is `NULL', this function has no effect.
The return value is the pointer to the given object.
This function assures that the specified region is released when the memory pool object is deleted.

The function `tcmpoolpushxstr' is used in order to relegate an extensible string object to a memory pool object.

TCXSTR *tcmpoolpushxstr(TCMPOOL *_mpool_, TCXSTR *_xstr_);

`_mpool_' specifies the memory pool object.
`_xstr_' specifies the extensible string object. If it is `NULL', this function has no effect.
The return value is the pointer to the given object.
This function assures that the specified object is deleted when the memory pool object is deleted.

The function `tcmpoolpushlist' is used in order to relegate a list object to a memory pool object.

TCLIST *tcmpoolpushlist(TCMPOOL *_mpool_, TCLIST *_list_);

`_mpool_' specifies the memory pool object.
`_list_' specifies the list object. If it is `NULL', this function has no effect.
The return value is the pointer to the given object.
This function assures that the specified object is deleted when the memory pool object is deleted.

The function `tcmpoolpushmap' is used in order to relegate a map object to a memory pool object.

TCMAP *tcmpoolpushmap(TCMPOOL *_mpool_, TCMAP *_map_);

`_mpool_' specifies the memory pool object.
`_map_' specifies the map object. If it is `NULL', this function has no effect.
The return value is the pointer to the given object.
This function assures that the specified object is deleted when the memory pool object is deleted.

The function `tcmpoolpushtree' is used in order to relegate a tree object to a memory pool object.

TCTREE *tcmpoolpushtree(TCMPOOL *_mpool_, TCTREE *_tree_);

`_mpool_' specifies the memory pool object.

`*tree*' specifies the tree object. If it is `NULL', this function has no effect.
The return value is the pointer to the given object.
This function assures that the specified object is deleted when the memory pool object is deleted.

The function `tcmpoolmalloc' is used in order to allocate a region relegated to a memory pool object.

`void *tcmpoolmalloc(TCMPOOL *mpool, size_t size);`
`*mpool*' specifies the memory pool object.
The return value is the pointer to the allocated region under the memory pool.

The function `tcmpoolxstrnew' is used in order to create an extensible string object relegated to a memory pool object.

`TCXSTR *tcmpoolxstrnew(TCMPOOL *mpool);`
The return value is the new extensible string object under the memory pool.

The function `tcmpoollistnew' is used in order to create a list object relegated to a memory pool object.

`TCLIST *tcmpoollistnew(TCMPOOL *mpool);`
The return value is the new list object under the memory pool.

The function `tcmpoolmapnew' is used in order to create a map object relegated to a memory pool object.

`TCMAP *tcmpoolmapnew(TCMPOOL *mpool);`
The return value is the new map object under the memory pool.

The function `tcmpooltreenew' is used in order to create a tree object relegated to a memory pool object.

`TCTREE *tcmpooltreenew(TCMPOOL *mpool);`
The return value is the new tree object under the memory pool.

The function `tcmpoolpop' is used in order to remove the most recently installed cleanup handler of a memory pool object.

`void tcmpoolpop(TCMPOOL *mpool, bool exe);`
`*mpool*' specifies the memory pool object.
`*exe*' specifies whether to execute the destructor of the removed handler.

The function `tcmpoolclear' is used in order to remove all cleanup handler of a memory pool object.

`void tcmpoolclear(TCMPOOL *mpool, bool exe);`
`*mpool*' specifies the memory pool object.
`*exe*' specifies whether to execute the destructors of the removed handlers.

The function `tcmpoolglobal' is used in order to get the global memory pool object.

`TCMPOOL *tcmpoolglobal(void);`
The return value is the global memory pool object.
The global memory pool object is a singleton and assured to be deleted when the process is terminating normally.

API of Miscellaneous Utilities

The function `tclmax' is used in order to get the larger value of two integers.

```
long tclmax(long a, long b);
```

`a' specifies an integer.
`b' specifies the other integer.
The return value is the larger value of the two.

The function `tclmin' is used in order to get the lesser value of two integers.

```
long tclmin(long a, long b);
```

`a' specifies an integer.
`b' specifies the other integer.
The return value is the lesser value of the two.

The function `tclrand' is used in order to get a random number as long integer based on uniform distribution.

```
unsigned long tclrand(void);
```

The return value is the random number between 0 and `ULONG_MAX'.
This function uses the random number source device and generates a real random number if possible.

The function `tcdrand' is used in order to get a random number as double decimal based on uniform distribution.

```
double tcdrand(void);
```

The return value is the random number equal to or greater than 0, and less than 1.0.
This function uses the random number source device and generates a real random number if possible.

The function `tcdrandnd' is used in order to get a random number as double decimal based on normal distribution.

```
double tcdrandnd(double avg, double sd);
```

`avg' specifies the average.
`sd' specifies the standard deviation.
The return value is the random number.
This function uses the random number source device and generates a real random number if possible.

The function `tcstricmp' is used in order to compare two strings with case insensitive evaluation.

```
int tcstricmp(const char *astr, const char *bstr);
```

`astr' specifies a string.
`bstr' specifies of the other string.
The return value is positive if the former is big, negative if the latter is big, 0 if both are equivalent.

The function `tcstrfwm' is used in order to check whether a string begins with a key.

```
bool tcstrfwm(const char *str, const char *key);
```

`str' specifies the target string.
`key' specifies the forward matching key string.

The return value is true if the target string begins with the key, else, it is false.

The function `tcstrifwm' is used in order to check whether a string begins with a key with case insensitive evaluation.

```
bool tcstrifwm(const char *str, const char *key);
```

`*str*' specifies the target string.
`*key*' specifies the forward matching key string.
The return value is true if the target string begins with the key, else, it is false.

The function `tcstrbwm' is used in order to check whether a string ends with a key.

```
bool tcstrbwm(const char *str, const char *key);
```

`*str*' specifies the target string.
`*key*' specifies the backward matching key string.
The return value is true if the target string ends with the key, else, it is false.

The function `tcstribwm' is used in order to check whether a string ends with a key with case insensitive evaluation.

```
bool tcstribwm(const char *str, const char *key);
```

`*str*' specifies the target string.
`*key*' specifies the backward matching key string.
The return value is true if the target string ends with the key, else, it is false.

The function `tcstrdist' is used in order to calculate the edit distance of two strings.

```
int tcstrdist(const char *astr, const char *bstr);
```

`*astr*' specifies a string.
`*bstr*' specifies of the other string.
The return value is the edit distance which is known as the Levenshtein distance. The cost is calculated by byte.

The function `tcstrdistutf' is used in order to calculate the edit distance of two UTF-8 strings.

```
int tcstrdistutf(const char *astr, const char *bstr);
```

`*astr*' specifies a string.
`*bstr*' specifies of the other string.
The return value is the edit distance which is known as the Levenshtein distance. The cost is calculated by Unicode character.

The function `tcstrtoupper' is used in order to convert the letters of a string into upper case.

```
char *tcstrtoupper(char *str);
```

`*str*' specifies the string to be converted.
The return value is the string itself.

The function `tcstrtolower' is used in order to convert the letters of a string into lower case.

```
char *tcstrtolower(char *str);
```

`*str*' specifies the string to be converted.
The return value is the string itself.

The function `tcstrtrim' is used in order to cut space characters at head or tail of a string.

```
char *tcstrtrim(char *str);
```

`*str*' specifies the string to be converted.
The return value is the string itself.

The function `tcstrsqzspc' is used in order to squeeze space characters in a string and trim it.

```
char *tcstrsqzspc(char *str);
```

`*str*' specifies the string to be converted.
The return value is the string itself.

The function `tcstrsubchr' is used in order to substitute characters in a string.

```
char *tcstrsubchr(char *str, const char *rstr, const char *sstr);
```

`*str*' specifies the string to be converted.
`*rstr*' specifies the string containing characters to be replaced.
`*sstr*' specifies the string containing characters to be substituted.
If the substitute string is shorter then the replacement string, corresponding characters are removed.

The function `tcstrcntutf' is used in order to count the number of characters in a string of UTF-8.

```
int tcstrcntutf(const char *str);
```

`*str*' specifies the string of UTF-8.
The return value is the number of characters in the string.

The function `tcstrcututf' is used in order to cut a string of UTF-8 at the specified number of characters.

```
char *tcstrcututf(char *str, int num);
```

`*str*' specifies the string of UTF-8.
`*num*' specifies the number of characters to be kept.
The return value is the string itself.

The function `tcstrutftoucs' is used in order to convert a UTF-8 string into a UCS-2 array.

```
void tcstrutftoucs(const char *str, uint16_t *ary, int *np);
```

`*str*' specifies the UTF-8 string.
`*ary*' specifies the pointer to the region into which the result UCS-2 codes are written. The size of the buffer should be sufficient.
`*np*' specifies the pointer to a variable into which the number of elements of the result array is assigned.

The function `tcstrucstoutf' is used in order to convert a UCS-2 array into a UTF-8 string.

```
int tcstrucstoutf(const uint16_t *ary, int num, char *str);
```

`*ary*' specifies the array of UCS-2 codes.
`*num*' specifies the number of the array.
`*str*' specifies the pointer to the region into which the result UTF-8 string is written. The size of the

buffer should be sufficient.
The return value is the length of the result string.

The function `tcstrsplit' is used in order to create a list object by splitting a string.

```
TCLIST *tcstrsplit(const char *str, const char *delims);
```

`*str*' specifies the source string.
`*delims*' specifies a string containing delimiting characters.
The return value is a list object of the split elements.
If two delimiters are successive, it is assumed that an empty element is between the two. Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcstrjoin' is used in order to create a string by joining all elements of a list object.

```
char *tcstrjoin(const TCLIST *list, char delim);
```

`*list*' specifies a list object.
`*delim*' specifies a delimiting character.
The return value is the result string.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcatoi' is used in order to convert a string to an integer.

```
int64_t tcatoi(const char *str);
```

`*str*' specifies the string.
The return value is the integer. If the string does not contain numeric expression, 0 is returned.
This function is equivalent to `atoll' except that it does not depend on the locale.

The function `tcatoix' is used in order to convert a string with a metric prefix to an integer.

```
int64_t tcatoix(const char *str);
```

`*str*' specifies the string, which can be trailed by a binary metric prefix. "K", "M", "G", "T", "P", and "E" are supported. They are case-insensitive.
The return value is the integer. If the string does not contain numeric expression, 0 is returned. If the integer overflows the domain, `INT64_MAX' or `INT64_MIN' is returned according to the sign.

The function `tcatof' is used in order to convert a string to a real number.

```
double tcatof(const char *str);
```

`*str*' specifies the string.
The return value is the real number. If the string does not contain numeric expression, 0.0 is returned.
This function is equivalent to `atof' except that it does not depend on the locale.

The function `tcregexmatch' is used in order to check whether a string matches a regular expression.

```
bool tcregexmatch(const char *str, const char *regex);
```

`*str*' specifies the target string.
`*regex*' specifies the regular expression string. If it begins with `*', the trailing substring is used as a case-insensitive regular expression.

The return value is true if matching is success, else, it is false.

The function `tcregexreplace' is used in order to replace each substring matching a regular expression string.

```
char *tcregexreplace(const char *str, const char *regex, const char *alt);
```

`*str*' specifies the target string.
`*regex*' specifies the regular expression string for substrings. If it begins with `*', the trailing substring is used as a case-insensitive regular expression.
`*alt*' specifies the alternative string with which each substrings is replaced. Each `&' in the string is replaced with the matched substring. Each `\' in the string escapes the following character. Special escapes "\1" through "\9" referring to the corresponding matching sub-expressions in the regular expression string are supported.
The return value is a new converted string. Even if the regular expression is invalid, a copy of the original string is returned.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcmd5hash' is used in order to get the MD5 hash value of a serial object.

```
void tcmd5hash(const void *ptr, int size, char *buf);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
`*buf*' specifies the pointer to the region into which the result string is written. The size of the buffer should be equal to or more than 48 bytes.

The function `tcarccipher' is used in order to cipher or decipher a serial object with the Arcfour stream cipher.

```
void tcarccipher(const void *ptr, int size, const void *kbuf, int ksiz, void *obuf);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
`*kbuf*' specifies the pointer to the region of the cipher key.
`*ksiz*' specifies the size of the region of the cipher key.
`*obuf*' specifies the pointer to the region into which the result data is written. The size of the buffer should be equal to or more than the input region.

The function `tctime' is used in order to get the time of day in seconds.

```
double tctime(void);
```

The return value is the time of day in seconds. The accuracy is in microseconds.

The function `tccalendar' is used in order to get the Gregorian calendar of a time.

```
void tccalendar(int64_t t, int jl, int *yearp, int *monp, int *dayp, int *hourp, int *minp, int *secp);
```

`*t*' specifies the source time in seconds from the epoch. If it is `INT64_MAX', the current time is specified.
`*jl*' specifies the jet lag of a location in seconds. If it is `INT_MAX', the local jet lag is specified.

`*yearp*' specifies the pointer to a variable to which the year is assigned. If it is `NULL', it is not used.
`*monp*' specifies the pointer to a variable to which the month is assigned. If it is `NULL', it is not used. 1 means January and 12 means December.
`*dayp*' specifies the pointer to a variable to which the day of the month is assigned. If it is `NULL', it is not used.
`*hourp*' specifies the pointer to a variable to which the hours is assigned. If it is `NULL', it is not used.
`*minp*' specifies the pointer to a variable to which the minutes is assigned. If it is `NULL', it is not used.
`*secp*' specifies the pointer to a variable to which the seconds is assigned. If it is `NULL', it is not used.

The function `tcdatestrwww' is used in order to format a date as a string in W3CDTF.

```
void tcdatestrwww(int64_t t, int jl, char *buf);
```

`*t*' specifies the source time in seconds from the epoch. If it is `INT64_MAX', the current time is specified.
`*jl*' specifies the jet lag of a location in seconds. If it is `INT_MAX', the local jet lag is specified.
`*buf*' specifies the pointer to the region into which the result string is written. The size of the buffer should be equal to or more than 48 bytes.
W3CDTF represents a date as "YYYY-MM-DDThh:mm:ddTZD".

The function `tcdatestrhttp' is used in order to format a date as a string in RFC 1123 format.

```
void tcdatestrhttp(int64_t t, int jl, char *buf);
```

`*t*' specifies the source time in seconds from the epoch. If it is `INT64_MAX', the current time is specified.
`*jl*' specifies the jet lag of a location in seconds. If it is `INT_MAX', the local jet lag is specified.
`*buf*' specifies the pointer to the region into which the result string is written. The size of the buffer should be equal to or more than 48 bytes.
RFC 1123 format represents a date as "Wdy, DD-Mon-YYYY hh:mm:dd TZD".

The function `tcstrmktime' is used in order to get the time value of a date string.

```
int64_t tcstrmktime(const char *str);
```

`*str*' specifies the date string in decimal, hexadecimal, W3CDTF, or RFC 822 (1123). Decimal can be trailed by "s" for in seconds, "m" for in minutes, "h" for in hours, and "d" for in days.
The return value is the time value of the date or `INT64_MIN' if the format is invalid.

The function `tcjetlag' is used in order to get the jet lag of the local time.

```
int tcjetlag(void);
```

The return value is the jet lag of the local time in seconds.

The function `tcdayofweek' is used in order to get the day of week of a date.

```
int tcdayofweek(int year, int mon, int day);
```

`*year*' specifies the year of a date.
`*mon*' specifies the month of the date.
`*day*' specifies the day of the date.
The return value is the day of week of the date. 0 means Sunday and 6 means Saturday.

API of Filesystem Utilities

The function `tcrealpath' is used in order to get the canonicalized absolute path of a file.

```
char *tcrealpath(const char *path);
```

`*path*' specifies the path of the file.
The return value is the canonicalized absolute path of a file, or `NULL' if the path is invalid.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcstatfile' is used in order to get the status information of a file.

```
bool tcstatfile(const char *path, bool *isdirp, int64_t *sizep, int64_t *mtimep);
```

`*path*' specifies the path of the file.
`*isdirp*' specifies the pointer to a variable into which whether the file is a directory is assigned. If it is `NULL', it is ignored.
`*sizep*' specifies the pointer to a variable into which the size of the file is assigned. If it is `NULL', it is ignored.
`*ntimep*' specifies the pointer to a variable into which the size of the file is assigned. If it is `NULL', it is ignored.
If successful, the return value is true, else, it is false.

The function `tcreadfile' is used in order to read whole data of a file.

```
void *tcreadfile(const char *path, int limit, int *sp);
```

`*path*' specifies the path of the file. If it is `NULL', the standard input is specified.
`*limit*' specifies the limiting size of reading data. If it is not more than 0, the limitation is not specified.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned. If it is `NULL', it is not used.
The return value is the pointer to the allocated region of the read data, or `NULL' if the file could not be opened.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when when is no longer in use.

The function `tcreadfilelines' is used in order to read every line of a file.

```
TCLIST *tcreadfilelines(const char *path);
```

`*path*' specifies the path of the file. If it is `NULL', the standard input is specified.
The return value is a list object of every lines if successful, else it is `NULL'.
Line separators are cut out. Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcwritefile' is used in order to write data into a file.

```
bool tcwritefile(const char *path, const void *ptr, int size);
```

`*path*' specifies the path of the file. If it is `NULL', the standard output is specified.
`*ptr*' specifies the pointer to the data region.
`*size*' specifies the size of the region.
If successful, the return value is true, else, it is false.

The function `tccopyfile' is used in order to copy a file.

```
bool tccopyfile(const char *src, const char *dest);
```

`*src*' specifies the path of the source file.
`*dest*' specifies the path of the destination file.
The return value is true if successful, else, it is false.
If the destination file exists, it is overwritten.

The function `tcreaddir' is used in order to read names of files in a directory.

```
TCLIST *tcreaddir(const char *path);
```

`*path*' specifies the path of the directory.
The return value is a list object of names if successful, else it is `NULL'.
Links to the directory itself and to the parent directory are ignored.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcglobpat' is used in order to expand a pattern into a list of matched paths.

```
TCLIST *tcglobpat(const char *pattern);
```

`*pattern*' specifies the matching pattern.
The return value is a list object of matched paths. If no path is matched, an empty list is returned.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcremovelink' is used in order to remove a file or a directory and its sub ones recursively.

```
bool tcremovelink(const char *path);
```

`*path*' specifies the path of the link.
If successful, the return value is true, else, it is false. False is returned when the link does not exist or the permission is denied.

The function `tcwrite' is used in order to write data into a file.

```
bool tcwrite(int fd, const void *buf, size_t size);
```

`*fd*' specifies the file descriptor.
`*buf*' specifies the buffer to be written.
`*size*' specifies the size of the buffer.
The return value is true if successful, else, it is false.

The function `tcread' is used in order to read data from a file.

```
bool tcread(int fd, void *buf, size_t size);
```

`*fd*' specifies the file descriptor.
`*buf*' specifies the buffer to store into.
`*size*' specifies the size of the buffer.
The return value is true if successful, else, it is false.

The function `tclock' is used in order to lock a file.

```
bool tclock(int fd, bool ex, bool nb);
```

`*fd*' specifies the file descriptor.

`*ex*' specifies whether an exclusive lock or a shared lock is performed.
`*nb*' specifies whether to request with non-blocking.
The return value is true if successful, else, it is false.

The function `tcunlock' is used in order to unlock a file.

```
bool tcunlock(int fd);
```

`*fd*' specifies the file descriptor.
The return value is true if successful, else, it is false.

The function `tcsystem' is used in order to execute a shell command.

```
int tcsystem(const char **args, int anum);
```

`*args*' specifies an array of the command name and its arguments.
`*anum*' specifies the number of elements of the array.
The return value is the exit code of the command or `INT_MAX' on failure.
The command name and the arguments are quoted and meta characters are escaped.

API of Encoding Utilities

The function `tcurlencode' is used in order to encode a serial object with URL encoding.

```
char *tcurlencode(const char *ptr, int size);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
The return value is the result string.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call if when is no longer in use.

The function `tcurldecode' is used in order to decode a string encoded with URL encoding.

```
char *tcurldecode(const char *str, int *sp);
```

`*str*' specifies the encoded string.
`*sp*' specifies the pointer to a variable into which the size of the region of the return value is assigned.
The return value is the pointer to the region of the result.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcurlbreak' is used in order to break up a URL into elements.

```
TCMAP *tcurlbreak(const char *str);
```

`*str*' specifies the URL string.
The return value is the map object whose keys are the name of elements. The key "self" indicates the URL itself. The key "scheme" indicates the scheme. The key "host" indicates the host of the server. The key "port" indicates the port number of the server. The key "authority" indicates the authority information. The key "path" indicates the path of the resource. The key "file" indicates the file name without the directory section. The key "query" indicates the query string. The key "fragment" indicates the fragment string.
Supported schema are HTTP, HTTPS, FTP, and FILE. Absolute URL and relative URL are supported.

Because the object of the return value is created with the function `tcmapnew', it should be deleted with the function `tcmapdel' when it is no longer in use.

The function `tcurlresolve' is used in order to resolve a relative URL with an absolute URL.

```
char *tcurlresolve(const char *base, const char *target);
```

`*base*' specifies the absolute URL of the base location.
`*target*' specifies the URL to be resolved.
The return value is the resolved URL. If the target URL is relative, a new URL of relative location from the base location is returned. Else, a copy of the target URL is returned.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcbaseencode' is used in order to encode a serial object with Base64 encoding.

```
char *tcbaseencode(const char *ptr, int size);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
The return value is the result string.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call if when is no longer in use.

The function `tcbasedecode' is used in order to decode a string encoded with Base64 encoding.

```
char *tcbasedecode(const char *str, int *sp);
```

`*str*' specifies the encoded string.
`*sp*' specifies the pointer to a variable into which the size of the region of the return value is assigned.
The return value is the pointer to the region of the result.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcquoteencode' is used in order to encode a serial object with Quoted-printable encoding.

```
char *tcquoteencode(const char *ptr, int size);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
The return value is the result string.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call if when is no longer in use.

The function `tcquotedecode' is used in order to decode a string encoded with Quoted-printable encoding.

```
char *tcquotedecode(const char *str, int *sp);
```

`*str*' specifies the encoded string.
`*sp*' specifies the pointer to a variable into which the size of the region of the return value is assigned.
The return value is the pointer to the region of the result.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcmimeencode' is used in order to encode a string with MIME encoding.

```
char *tcmimeencode(const char *str, const char *encname, bool base);
```

`*str*' specifies the string.
`*encname*' specifies the string of the name of the character encoding.
`*base*' specifies whether to use Base64 encoding. If it is false, Quoted-printable is used.
The return value is the result string.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcmimedecode' is used in order to decode a string encoded with MIME encoding.

```
char *tcmimedecode(const char *str, char *enp);
```

`*str*' specifies the encoded string.
`*enp*' specifies the pointer to the region into which the name of encoding is written. If it is `NULL', it is not used. The size of the buffer should be equal to or more than 32 bytes.
The return value is the result string.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcmimebreak' is used in order to split a string of MIME into headers and the body.

```
char *tcmimebreak(const char *ptr, int size, TCMAP *headers, int *sp);
```

`*ptr*' specifies the pointer to the region of MIME data.
`*size*' specifies the size of the region.
`*headers*' specifies a map object to store headers. If it is `NULL', it is not used. Each key of the map is an uncapitalized header name.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
The return value is the pointer to the region of the body data.
If the content type is defined, the header map has the key "TYPE" specifying the type. If the character encoding is defined, the key "CHARSET" indicates the encoding name. If the boundary string of multipart is defined, the key "BOUNDARY" indicates the string. If the content disposition is defined, the key "DISPOSITION" indicates the direction. If the file name is defined, the key "FILENAME" indicates the name. If the attribute name is defined, the key "NAME" indicates the name. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcmimeparts' is used in order to split multipart data of MIME into its parts.

```
TCLIST *tcmimeparts(const char *ptr, int size, const char *boundary);
```

`*ptr*' specifies the pointer to the region of multipart data of MIME.
`*size*' specifies the size of the region.
`*boundary*' specifies the boundary string.
The return value is a list object. Each element of the list is the data of a part.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tchexencode' is used in order to encode a serial object with hexadecimal encoding.

```
char *tchexencode(const char *ptr, int size);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
The return value is the result string.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call if when is no longer in use.

The function `tchexdecode' is used in order to decode a string encoded with hexadecimal encoding.

```
char *tchexdecode(const char *str, int *sp);
```

`*str*' specifies the encoded string.
`*sp*' specifies the pointer to a variable into which the size of the region of the return value is assigned.
The return value is the pointer to the region of the result.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcpackencode' is used in order to compress a serial object with Packbits encoding.

```
char *tcpackencode(const char *ptr, int size, int *sp);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the result object, else, it is `NULL'.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcpackdecode' is used in order to decompress a serial object compressed with Packbits encoding.

```
char *tcpackdecode(const char *ptr, int size, int *sp);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
`*sp*' specifies the pointer to a variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the result object, else, it is `NULL'.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcbsencode' is used in order to compress a serial object with TCBS encoding.

```
char *tcbsencode(const char *ptr, int size, int *sp);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the result object, else, it is `NULL'.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcbsdecode' is used in order to decompress a serial object compressed with TCBS encoding.

```
char *tcbsdecode(const char *ptr, int size, int *sp);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
`*sp*' specifies the pointer to a variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the result object, else, it is `NULL'.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcdeflate' is used in order to compress a serial object with Deflate encoding.

```
char *tcdeflate(const char *ptr, int size, int *sp);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the result object, else, it is `NULL'.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcinflate' is used in order to decompress a serial object compressed with Deflate encoding.

```
char *tcinflate(const char *ptr, int size, int *sp);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
`*sp*' specifies the pointer to a variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the result object, else, it is `NULL'.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcgzipencode' is used in order to compress a serial object with GZIP encoding.

```
char *tcgzipencode(const char *ptr, int size, int *sp);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the result object, else, it is `NULL'.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcgzipdecode' is used in order to decompress a serial object compressed with GZIP encoding.

```
char *tcgzipdecode(const char *ptr, int size, int *sp);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
`*sp*' specifies the pointer to a variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the result object, else, it is `NULL'.
Because an additional zero code is appended at the end of the region of the return value, the return

value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcgetcrc' is used in order to get the CRC32 checksum of a serial object.

```
unsigned int tcgetcrc(const char *ptr, int size);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
The return value is the CRC32 checksum of the object.

The function `tcbzipencode' is used in order to compress a serial object with BZIP2 encoding.

```
char *tcbzipencode(const char *ptr, int size, int *sp);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the result object, else, it is `NULL'.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcbzipdecode' is used in order to decompress a serial object compressed with BZIP2 encoding.

```
char *tcbzipdecode(const char *ptr, int size, int *sp);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
`*sp*' specifies the pointer to a variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the result object, else, it is `NULL'.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcberencode' is used in order to encode an array of nonnegative integers with BER encoding.

```
char *tcberencode(const unsigned int *ary, int anum, int *sp);
```

`*ary*' specifies the pointer to the array of nonnegative integers.
`*anum*' specifies the size of the array.
`*sp*' specifies the pointer to a variable into which the size of the region of the return value is assigned.
The return value is the pointer to the region of the result.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call if when is no longer in use.

The function `tcberdecode' is used in order to decode a serial object encoded with BER encoding.

```
unsigned int *tcberdecode(const char *ptr, int size, int *np);
```

`*ptr*' specifies the pointer to the region.
`*size*' specifies the size of the region.
`*np*' specifies the pointer to a variable into which the number of elements of the return value is assigned.
The return value is the pointer to the array of the result.

Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call if when is no longer in use.

The function `tcxmlescape' is used in order to escape meta characters in a string with the entity references of XML.

`char *tcxmlescape(const char *str);`

`*str*' specifies the string.
The return value is the pointer to the escaped string.
This function escapes only `&', `<', `>', and `"'. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcxmlunescape' is used in order to unescape entity references in a string of XML.

`char *tcxmlunescape(const char *str);`

`*str*' specifies the string.
The return value is the unescaped string.
This function restores only `&', `<', `>', and `"'. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

Example Code

The following code is an example using extensible string, array list, and hash map.

```
#include <tcutil.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>

int main(int argc, char **argv){

  { /* example to use an extensible string object */
    TCXSTR *xstr;
    /* create the object */
    xstr = tcxstrnew();
    /* concatenate strings */
    tcxstrcat2(xstr, "hop");
    tcxstrcat2(xstr, "step");
    tcxstrcat2(xstr, "jump");
    /* print the size and the content */
    printf("%d:%s\n", tcxstrsize(xstr), (char *)tcxstrptr(xstr));
    /* delete the object */
    tcxstrdel(xstr);
  }

  { /* example to use a list object */
    TCLIST *list;
    int i;
    /* create the object */
    list = tclistnew();
    /* add strings to the tail */
    tclistpush2(list, "hop");
    tclistpush2(list, "step");
    tclistpush2(list, "jump");
    /* print all elements */
    for(i = 0; i < tclistnum(list); i++){
      printf("%d:%s\n", i, tclistval2(list, i));
```

```
    }
    /* delete the object */
    tclistdel(list);
  }

  { /* example to use a map object */
    TCMAP *map;
    const char *key;
    /* create the object */
    map = tcmapnew();
    /* add records */
    tcmapput2(map, "foo", "hop");
    tcmapput2(map, "bar", "step");
    tcmapput2(map, "baz", "jump");
    /* print all records */
    tcmapiterinit(map);
    while((key = tcmapiternext2(map)) != NULL){
      printf("%s:%s\n", key, tcmapget2(map, key));
    }
    /* delete the object */
    tcmapdel(map);
  }

  { /* example to use a tree object */
    TCTREE *tree;
    const char *key;
    /* create the object */
    tree = tctreenew();
    /* add records */
    tctreeput2(tree, "foo", "hop");
    tctreeput2(tree, "bar", "step");
    tctreeput2(tree, "baz", "jump");
    /* print all records */
    tctreeiterinit(tree);
    while((key = tctreeiternext2(tree)) != NULL){
      printf("%s:%s\n", key, tctreeget2(tree, key));
    }
    /* delete the object */
    tctreedel(tree);
  }

  return 0;
}
```

CLI

To use the utility API easily, the commands `**tcutest**', `**tcumttest**', and `**tcucodec**' are provided.

The command `**tcutest**' is a utility for facility test and performance test. This command is used in the following format. `*rnum*' specifies the number of iterations. `*anum*' specifies the initial number of elements of array. `*bnum*' specifies the number of buckets.

tcutest xstr *rnum*
: Perform test of extensible string.

tcutest list [-rd] *rnum* [*anum*]
: Perform test of array list.

tcutest map [-rd] [-tr] [-rnd] [-dk|-dc|-dai|-dad|-dpr] *rnum* [*bnum*]
: Perform test of hash map.

tcutest tree [-rd] [-tr] [-rnd] [-dk|-dc|-dai|-dad|-dpr] *rnum*

Perform test of ordered tree.

tcutest mdb [-rd] [-tr] [-rnd] [-dk|-dc|-dai|-dad|-dpr] ***rnum*** **[*bnum*]**

Perform test of on-memory hash database.

tcutest ndb [-rd] [-tr] [-rnd] [-dk|-dc|-dai|-dad|-dpr] ***rnum***

Perform test of on-memory tree database.

tcutest misc ***rnum***

Perform test of miscellaneous routines.

tcutest wicked ***rnum***

Perform updating operations of list and map selected at random.

Options feature the following.

-rd : perform the reading test also.
-tr : perform the iterator test also.
-rnd : select keys at random.
-dk : use the function `tcxxxputkeep' instead of `tcxxxput'.
-dc : use the function `tcxxxputcat' instead of `tcxxxput'.
-dai : use the function `tcxxxaddint' instead of `tcxxxput'.
-dad : use the function `tcxxxadddouble' instead of `tcxxxput'.
-dpr : use the function `tcxxxputproc' instead of `tcxxxput'.

This command returns 0 on success, another on failure.

The command `**tcumttest**' is a utility for facility test under multi-thread situation. This command is used in the following format. `*tnum*' specifies the number of running threads. `*rnum*' specifies the number of iterations. `*bnum*' specifies the number of buckets.

tcumttest combo [-rnd] ***tnum rnum*** **[*bnum*]**

Peform storing, retrieving, and removing in turn.

tcumttest typical [-nc] [-rr ***num*****]** ***tnum rnum*** **[*bnum*]**

Perform typical operations selected at random.

Options feature the following.

-rnd : select keys at random.
-nc : omit the comparison test.
-rr ***num*** : specify the ratio of reading operation by percentage.

This command returns 0 on success, another on failure.

The command `**tcucodec**' is a tool to use encoding and decoding features. This command is used in the following format. `*file*' specifies a input file. If it is omitted, the standard input is read.

tcucodec url [-d] [-br] [-rs ***base*****] [*file*]**

Perform URL encoding and its decoding.

tcucodec base [-d] [*file*]

Perform Base64 encoding and its decoding.

tcucodec quote [-d] [*file*]

Perform quoted-printable encoding and its decoding.

tcucodec mime [-d] [-en *name*] [-q] [-on] [-hd] [-bd] [-part *num*] [*file*]

Perform MIME encoding and its decoding.

tcucodec hex [-d] [*file*]

Perform hexadecimal encoding and its decoding.

tcucodec pack [-d] [-bwt] [*file*]

Perform Packbits encoding and its decoding.

tcucodec tcbs [-d] [*file*]

Perform TCBS encoding and its decoding.

tcucodec zlib [-d] [-gz] [*file*]

Perform ZLIB encoding and its decoding.

tcucodec bzip [-d] [*file*]

Perform BZIP2 encoding and its decoding.

tcucodec xml [-d] [-br] [*file*]

Process XML. By default, escape meta characters.

tcucodec cstr [-d] [-js] [*file*]

Perform C-string escaping and its unescaping.

tcucodec ucs [-d] [-un] [-kw *str*] [*file*]

Convert UTF-8 string into UCS-2 array.

tcucodec hash [-crc] [-ch *num*] [*file*]

Calculate the hash value. By default, use MD5 function.

tcucodec cipher [-key *str*] [*file*]

Perform stream cipher and its decipher.

tcucodec date [-ds *str*] [-jl *num*] [-wf] [-rf]

Process date string. By default, print the current UNIX time.

tcucodec tmpl [-var *name value*] [*file*]

Perform template serialization.

tcucodec conf [-v|-i|-l|-p]

Print some configurations.

Options feature the following.

- **-d** : perform decoding (unescaping), not encoding (escaping).
- **-br** : break up URL or XML into elements.
- **-rs *base*** : specify the base URL and resolve the relative URL.
- **-en *name*** : specify the input encoding, which is UTF-8 by default.
- **-q** : use quoted-printable encoding, which is Base64 by default.
- **-on** : output the charset name when decoding.
- **-bd** : perform MIME parsing and output the body.
- **-hd** : perform MIME parsing and output the headers.
- **-part *num*** : perform MIME parsing and output the specified part.
- **-bwt** : convert by BWT as preprocessing.
- **-gz** : use GZIP format.
- **-crc** : use CRC32 function.
- **-js** : use JSON compatible format.
- **-un** : perform UCS normalization.

- **-kw** ***str*** : generate KWIC string.
- **-ch** ***num*** : use consistent hashing function.
- **-key** ***str*** : specify the cipher key.
- **-ds** ***str*** : specify the time.
- **-jl** ***num*** : specify the jet lag.
- **-wf** : format the output in W3CDTF.
- **-rf** : format the output in RFC 1123 format.
- **-var** ***name value*** : specify a template variable.
- **-v** : show the version number of Tokyo Cabinet.
- **-i** : show options to include the headers of Tokyo Cabinet.
- **-l** : show options to link the library of Tokyo Cabinet.
- **-p** : show the directory path of the commands of Tokyo Cabinet.

This command returns 0 on success, another on failure.

The Hash Database API

Hash database is a file containing a hash table and is handled with the hash database API. See `**tchdb.h**' for the entire specification.

Description

To use the hash database API, include `**tcutil.h**', `**tchdb.h**', and related standard header files. Usually, write the following description near the front of a source file.

```
#include <tcutil.h>
#include <tchdb.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
```

Objects whose type is pointer to `**TCHDB**' are used to handle hash databases. A hash database object is created with the function `**tchdbnew**' and is deleted with the function `**tchdbdel**'. To avoid memory leak, it is important to delete every object when it is no longer in use.

Before operations to store or retrieve records, it is necessary to open a database file and connect the hash database object to it. The function `**tchdbopen**' is used to open a database file and the function `**tchdbclose**' is used to close the database file. To avoid data missing or corruption, it is important to close every database file when it is no longer in use. It is forbidden for multible database objects in a process to open the same database at the same time.

API

The function `tchdberrmsg' is used in order to get the message string corresponding to an error code.

```
const char *tchdberrmsg(int ecode);
```

`*ecode*' specifies the error code.
The return value is the message string of the error code.

The function `tchdbnew' is used in order to create a hash database object.

```
TCHDB *tchdbnew(void);
```

The return value is the new hash database object.

The function `tchdbdel' is used in order to delete a hash database object.

```
void tchdbdel(TCHDB *hdb);
```

`*hdb*' specifies the hash database object.
If the database is not closed, it is closed implicitly. Note that the deleted object and its derivatives can not be used anymore.

The function `tchdbecode' is used in order to get the last happened error code of a hash database object.

int tchdbecode(TCHDB *_hdb_);

`_hdb_' specifies the hash database object.
The return value is the last happened error code.
The following error codes are defined: `TCESUCCESS' for success, `TCETHREAD' for threading error, `TCEINVALID' for invalid operation, `TCENOFILE' for file not found, `TCENOPERM' for no permission, `TCEMETA' for invalid meta data, `TCERHEAD' for invalid record header, `TCEOPEN' for open error, `TCECLOSE' for close error, `TCETRUNC' for trunc error, `TCESYNC' for sync error, `TCESTAT' for stat error, `TCESEEK' for seek error, `TCEREAD' for read error, `TCEWRITE' for write error, `TCEMMAP' for mmap error, `TCELOCK' for lock error, `TCEUNLINK' for unlink error, `TCERENAME' for rename error, `TCEMKDIR' for mkdir error, `TCERMDIR' for rmdir error, `TCEKEEP' for existing record, `TCENOREC' for no record found, and `TCEMISC' for miscellaneous error.

The function `tchdbsetmutex' is used in order to set mutual exclusion control of a hash database object for threading.

bool tchdbsetmutex(TCHDB *_hdb_);

`_hdb_' specifies the hash database object which is not opened.
If successful, the return value is true, else, it is false.
Note that the mutual exclusion control of the database should be set before the database is opened.

The function `tchdbtune' is used in order to set the tuning parameters of a hash database object.

bool tchdbtune(TCHDB *_hdb_, int64_t _bnum_, int8_t _apow_, int8_t _fpow_, uint8_t _opts_);

`_hdb_' specifies the hash database object which is not opened.
`_bnum_' specifies the number of elements of the bucket array. If it is not more than 0, the default value is specified. The default value is 131071. Suggested size of the bucket array is about from 0.5 to 4 times of the number of all records to be stored.
`_apow_' specifies the size of record alignment by power of 2. If it is negative, the default value is specified. The default value is 4 standing for 2^4=16.
`_fpow_' specifies the maximum number of elements of the free block pool by power of 2. If it is negative, the default value is specified. The default value is 10 standing for 2^10=1024.
`_opts_' specifies options by bitwise-or: `HDBTLARGE' specifies that the size of the database can be larger than 2GB by using 64-bit bucket array, `HDBTDEFLATE' specifies that each record is compressed with Deflate encoding, `HDBTBZIP' specifies that each record is compressed with BZIP2 encoding, `HDBTTCBS' specifies that each record is compressed with TCBS encoding.
If successful, the return value is true, else, it is false.
Note that the tuning parameters should be set before the database is opened.

The function `tchdbsetcache' is used in order to set the caching parameters of a hash database object.

bool tchdbsetcache(TCHDB *_hdb_, int32_t _rcnum_);

`_hdb_' specifies the hash database object which is not opened.
`_rcnum_' specifies the maximum number of records to be cached. If it is not more than 0, the record cache is disabled. It is disabled by default.
If successful, the return value is true, else, it is false.
Note that the caching parameters should be set before the database is opened.

The function `tchdbsetxmsiz' is used in order to set the size of the extra mapped memory of a hash database object.

```
bool tchdbsetxmsiz(TCHDB *hdb, int64_t xmsiz);
```

`*hdb*' specifies the hash database object which is not opened.
`*xmsiz*' specifies the size of the extra mapped memory. If it is not more than 0, the extra mapped memory is disabled. The default size is 67108864.
If successful, the return value is true, else, it is false.
Note that the mapping parameters should be set before the database is opened.

The function `tchdbsetdfunit' is used in order to set the unit step number of auto defragmentation of a hash database object.

```
bool tchdbsetdfunit(TCHDB *hdb, int32_t dfunit);
```

`*hdb*' specifies the hash database object which is not opened.
`*dfunit*' specifie the unit step number. If it is not more than 0, the auto defragmentation is disabled. It is disabled by default.
If successful, the return value is true, else, it is false.
Note that the defragmentation parameters should be set before the database is opened.

The function `tchdbopen' is used in order to open a database file and connect a hash database object.

```
bool tchdbopen(TCHDB *hdb, const char *path, int omode);
```

`*hdb*' specifies the hash database object which is not opened.
`*path*' specifies the path of the database file.
`*omode*' specifies the connection mode: `HDBOWRITER' as a writer, `HDBOREADER' as a reader. If the mode is `HDBOWRITER', the following may be added by bitwise-or: `HDBOCREAT', which means it creates a new database if not exist, `HDBOTRUNC', which means it creates a new database regardless if one exists, `HDBOTSYNC', which means every transaction synchronizes updated contents with the device. Both of `HDBOREADER' and `HDBOWRITER' can be added to by bitwise-or: `HDBONOLCK', which means it opens the database file without file locking, or `HDBOLCKNB', which means locking is performed without blocking.
If successful, the return value is true, else, it is false.

The function `tchdbclose' is used in order to close a hash database object.

```
bool tchdbclose(TCHDB *hdb);
```

`*hdb*' specifies the hash database object.
If successful, the return value is true, else, it is false.
Update of a database is assured to be written when the database is closed. If a writer opens a database but does not close it appropriately, the database will be broken.

The function `tchdbput' is used in order to store a record into a hash database object.

```
bool tchdbput(TCHDB *hdb, const void *kbuf, int ksiz, const void *vbuf, int vsiz);
```

`*hdb*' specifies the hash database object connected as a writer.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.

`*vsiz*' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tchdbput2' is used in order to store a string record into a hash database object.

```
bool tchdbput2(TCHDB *hdb, const char *kstr, const char *vstr);
```

`*hdb*' specifies the hash database object connected as a writer.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tchdbputkeep' is used in order to store a new record into a hash database object.

```
bool tchdbputkeep(TCHDB *hdb, const void *kbuf, int ksiz, const void *vbuf, int vsiz);
```

`*hdb*' specifies the hash database object connected as a writer.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tchdbputkeep2' is used in order to store a new string record into a hash database object.

```
bool tchdbputkeep2(TCHDB *hdb, const char *kstr, const char *vstr);
```

`*hdb*' specifies the hash database object connected as a writer.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tchdbputcat' is used in order to concatenate a value at the end of the existing record in a hash database object.

```
bool tchdbputcat(TCHDB *hdb, const void *kbuf, int ksiz, const void *vbuf, int vsiz);
```

`*hdb*' specifies the hash database object connected as a writer.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tchdbputcat2' is used in order to concatenate a string value at the end of the existing record in a hash database object.

bool tchdbputcat2(TCHDB *_hdb_, const char *_kstr_, const char *_vstr_);

`_hdb_' specifies the hash database object connected as a writer.
`_kstr_' specifies the string of the key.
`_vstr_' specifies the string of the value.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tchdbputasync' is used in order to store a record into a hash database object in asynchronous fashion.

bool tchdbputasync(TCHDB *_hdb_, const void *_kbuf_, int _ksiz_, const void *_vbuf_, int _vsiz_);

`_hdb_' specifies the hash database object connected as a writer.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_vbuf_' specifies the pointer to the region of the value.
`_vsiz_' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten. Records passed to this function are accumulated into the inner buffer and wrote into the file at a blast.

The function `tchdbputasync2' is used in order to store a string record into a hash database object in asynchronous fashion.

bool tchdbputasync2(TCHDB *_hdb_, const char *_kstr_, const char *_vstr_);

`_hdb_' specifies the hash database object connected as a writer.
`_kstr_' specifies the string of the key.
`_vstr_' specifies the string of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten. Records passed to this function are accumulated into the inner buffer and wrote into the file at a blast.

The function `tchdbout' is used in order to remove a record of a hash database object.

bool tchdbout(TCHDB *_hdb_, const void *_kbuf_, int _ksiz_);

`_hdb_' specifies the hash database object connected as a writer.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
If successful, the return value is true, else, it is false.

The function `tchdbout2' is used in order to remove a string record of a hash database object.

bool tchdbout2(TCHDB *_hdb_, const char *_kstr_);

`_hdb_' specifies the hash database object connected as a writer.
`_kstr_' specifies the string of the key.
If successful, the return value is true, else, it is false.

The function `tchdbget' is used in order to retrieve a record in a hash database object.

void *tchdbget(TCHDB *_hdb_, const void *_kbuf_, int _ksiz_, int *_sp_);

`_hdb_' specifies the hash database object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_sp_' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the value of the corresponding record. `NULL' is returned if no record corresponds.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tchdbget2' is used in order to retrieve a string record in a hash database object.

char *tchdbget2(TCHDB *_hdb_, const char *_kstr_);

`_hdb_' specifies the hash database object.
`_kstr_' specifies the string of the key.
If successful, the return value is the string of the value of the corresponding record. `NULL' is returned if no record corresponds.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tchdbget3' is used in order to retrieve a record in a hash database object and write the value into a buffer.

int tchdbget3(TCHDB *_hdb_, const void *_kbuf_, int _ksiz_, void *_vbuf_, int _max_);

`_hdb_' specifies the hash database object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_vbuf_' specifies the pointer to the buffer into which the value of the corresponding record is written.
`_max_' specifies the size of the buffer.
If successful, the return value is the size of the written data, else, it is -1. -1 is returned if no record corresponds to the specified key.
Note that an additional zero code is not appended at the end of the region of the writing buffer.

The function `tchdbvsiz' is used in order to get the size of the value of a record in a hash database object.

int tchdbvsiz(TCHDB *_hdb_, const void *_kbuf_, int _ksiz_);

`_hdb_' specifies the hash database object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tchdbvsiz2' is used in order to get the size of the value of a string record in a hash database object.

int tchdbvsiz2(TCHDB *_hdb_, const char *_kstr_);

`_hdb_' specifies the hash database object.
`_kstr_' specifies the string of the key.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tchdbiterinit' is used in order to initialize the iterator of a hash database object.

```
bool tchdbiterinit(TCHDB *hdb);
```

`*hdb*' specifies the hash database object.
If successful, the return value is true, else, it is false.
The iterator is used in order to access the key of every record stored in a database.

The function `tchdbiternext' is used in order to get the next key of the iterator of a hash database object.

```
void *tchdbiternext(TCHDB *hdb, int *sp);
```

`*hdb*' specifies the hash database object.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the next key, else, it is `NULL'. `NULL' is returned when no record is to be get out of the iterator.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. It is possible to access every record by iteration of calling this function. It is allowed to update or remove records whose keys are fetched while the iteration. However, it is not assured if updating the database is occurred while the iteration. Besides, the order of this traversal access method is arbitrary, so it is not assured that the order of storing matches the one of the traversal access.

The function `tchdbiternext2' is used in order to get the next key string of the iterator of a hash database object.

```
char *tchdbiternext2(TCHDB *hdb);
```

`*hdb*' specifies the hash database object.
If successful, the return value is the string of the next key, else, it is `NULL'. `NULL' is returned when no record is to be get out of the iterator.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. It is possible to access every record by iteration of calling this function. However, it is not assured if updating the database is occurred while the iteration. Besides, the order of this traversal access method is arbitrary, so it is not assured that the order of storing matches the one of the traversal access.

The function `tchdbiternext3' is used in order to get the next extensible objects of the iterator of a hash database object.

```
bool tchdbiternext3(TCHDB *hdb, TCXSTR *kxstr, TCXSTR *vxstr);
```

`*hdb*' specifies the hash database object.
`*kxstr*' specifies the object into which the next key is wrote down.
`*vxstr*' specifies the object into which the next value is wrote down.
If successful, the return value is true, else, it is false. False is returned when no record is to be get out of the iterator.

The function `tchdbfwmkeys' is used in order to get forward matching keys in a hash database object.

```
TCLIST *tchdbfwmkeys(TCHDB *hdb, const void *pbuf, int psiz, int max);
```

`*hdb*' specifies the hash database object.

`*pbuf*' specifies the pointer to the region of the prefix.
`*psiz*' specifies the size of the region of the prefix.
`*max*' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.
The return value is a list object of the corresponding keys. This function does never fail. It returns an empty list even if no key corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use. Note that this function may be very slow because every key in the database is scanned.

The function `tchdbfwmkeys2' is used in order to get forward matching string keys in a hash database object.

```
TCLIST *tchdbfwmkeys2(TCHDB *hdb, const char *pstr, int max);
```

`*hdb*' specifies the hash database object.
`*pstr*' specifies the string of the prefix.
`*max*' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.
The return value is a list object of the corresponding keys. This function does never fail. It returns an empty list even if no key corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use. Note that this function may be very slow because every key in the database is scanned.

The function `tchdbaddint' is used in order to add an integer to a record in a hash database object.

```
int tchdbaddint(TCHDB *hdb, const void *kbuf, int ksiz, int num);
```

`*hdb*' specifies the hash database object connected as a writer.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*num*' specifies the additional value.
If successful, the return value is the summation value, else, it is `INT_MIN'.
If the corresponding record exists, the value is treated as an integer and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tchdbdbadddouble' is used in order to add a real number to a record in a hash database object.

```
double tchdbadddouble(TCHDB *hdb, const void *kbuf, int ksiz, double num);
```

`*hdb*' specifies the hash database object connected as a writer.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*num*' specifies the additional value.
If successful, the return value is the summation value, else, it is Not-a-Number.
If the corresponding record exists, the value is treated as a real number and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tchdbsync' is used in order to synchronize updated contents of a hash database object with the file and the device.

```
bool tchdbsync(TCHDB *hdb);
```

`*hdb*' specifies the hash database object connected as a writer.

If successful, the return value is true, else, it is false.
This function is useful when another process connects to the same database file.

The function `tchdboptimize' is used in order to optimize the file of a hash database object.

bool tchdboptimize(TCHDB *_hdb_, int64_t _bnum_, int8_t _apow_, int8_t _fpow_, uint8_t _opts_);

`_hdb_' specifies the hash database object connected as a writer.
`_bnum_' specifies the number of elements of the bucket array. If it is not more than 0, the default value is specified. The default value is two times of the number of records.
`_apow_' specifies the size of record alignment by power of 2. If it is negative, the current setting is not changed.
`_fpow_' specifies the maximum number of elements of the free block pool by power of 2. If it is negative, the current setting is not changed.
`_opts_' specifies options by bitwise-or: `HDBTLARGE' specifies that the size of the database can be larger than 2GB by using 64-bit bucket array, `HDBTDEFLATE' specifies that each record is compressed with Deflate encoding, `HDBTBZIP' specifies that each record is compressed with BZIP2 encoding, `HDBTTCBS' specifies that each record is compressed with TCBS encoding. If it is `UINT8_MAX', the current setting is not changed.
If successful, the return value is true, else, it is false.
This function is useful to reduce the size of the database file with data fragmentation by successive updating.

The function `tchdbvanish' is used in order to remove all records of a hash database object.

bool tchdbvanish(TCHDB *_hdb_);

`_hdb_' specifies the hash database object connected as a writer.
If successful, the return value is true, else, it is false.

The function `tchdbcopy' is used in order to copy the database file of a hash database object.

bool tchdbcopy(TCHDB *_hdb_, const char *_path_);

`_hdb_' specifies the hash database object.
`_path_' specifies the path of the destination file. If it begins with `@', the trailing substring is executed as a command line.
If successful, the return value is true, else, it is false. False is returned if the executed command returns non-zero code.
The database file is assured to be kept synchronized and not modified while the copying or executing operation is in progress. So, this function is useful to create a backup file of the database file.

The function `tchdbtranbegin' is used in order to begin the transaction of a hash database object.

bool tchdbtranbegin(TCHDB *_hdb_);

`_hdb_' specifies the hash database object connected as a writer.
If successful, the return value is true, else, it is false.
The database is locked by the thread while the transaction so that only one transaction can be activated with a database object at the same time. Thus, the serializable isolation level is assumed if every database operation is performed in the transaction. All updated regions are kept track of by write ahead logging while the transaction. If the database is closed during transaction, the transaction is aborted

implicitly.

The function `tchdbtrancommit' is used in order to commit the transaction of a hash database object.

bool tchdbtrancommit(TCHDB *_hdb_);

`_hdb_' specifies the hash database object connected as a writer.
If successful, the return value is true, else, it is false.
Update in the transaction is fixed when it is committed successfully.

The function `tchdbtranabort' is used in order to abort the transaction of a hash database object.

bool tchdbtranabort(TCHDB *_hdb_);

`_hdb_' specifies the hash database object connected as a writer.
If successful, the return value is true, else, it is false.
Update in the transaction is discarded when it is aborted. The state of the database is rollbacked to before transaction.

The function `tchdbpath' is used in order to get the file path of a hash database object.

const char *tchdbpath(TCHDB *_hdb_);

`_hdb_' specifies the hash database object.
The return value is the path of the database file or `NULL' if the object does not connect to any database file.

The function `tchdbrnum' is used in order to get the number of records of a hash database object.

uint64_t tchdbrnum(TCHDB *_hdb_);

`_hdb_' specifies the hash database object.
The return value is the number of records or 0 if the object does not connect to any database file.

The function `tchdbfsiz' is used in order to get the size of the database file of a hash database object.

uint64_t tchdbfsiz(TCHDB *_hdb_);

`_hdb_' specifies the hash database object.
The return value is the size of the database file or 0 if the object does not connect to any database file.

Example Code

The following code is an example to use a hash database.

```
#include <tcutil.h>
#include <tchdb.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>

int main(int argc, char **argv){
  TCHDB *hdb;
  int ecode;
  char *key, *value;

  /* create the object */
  hdb = tchdbnew();
```

```
  /* open the database */
  if(!tchdbopen(hdb, "casket.tch", HDBOWRITER | HDBOCREAT)){
    ecode = tchdbecode(hdb);
    fprintf(stderr, "open error: %s\n", tchdberrmsg(ecode));
  }

  /* store records */
  if(!tchdbput2(hdb, "foo", "hop") ||
     !tchdbput2(hdb, "bar", "step") ||
     !tchdbput2(hdb, "baz", "jump")){
    ecode = tchdbecode(hdb);
    fprintf(stderr, "put error: %s\n", tchdberrmsg(ecode));
  }

  /* retrieve records */
  value = tchdbget2(hdb, "foo");
  if(value){
    printf("%s\n", value);
    free(value);
  } else {
    ecode = tchdbecode(hdb);
    fprintf(stderr, "get error: %s\n", tchdberrmsg(ecode));
  }

  /* traverse records */
  tchdbiterinit(hdb);
  while((key = tchdbiternext2(hdb)) != NULL){
    value = tchdbget2(hdb, key);
    if(value){
      printf("%s:%s\n", key, value);
      free(value);
    }
    free(key);
  }

  /* close the database */
  if(!tchdbclose(hdb)){
    ecode = tchdbecode(hdb);
    fprintf(stderr, "close error: %s\n", tchdberrmsg(ecode));
  }

  /* delete the object */
  tchdbdel(hdb);

  return 0;
}
```

CLI

To use the hash database API easily, the commands `**tchtest**', `**tchmttest**', and `**tchmgr**' are provided.

The command `**tchtest**' is a utility for facility test and performance test. This command is used in the following format. `*path*' specifies the path of a database file. `*rnum*' specifies the number of iterations. `*bnum*' specifies the number of buckets. `*apow*' specifies the power of the alignment. `*fpow*' specifies the power of the free block pool.

tchtest write [-mt] [-tl] [-td|-tb|-tt|-tx] [-rc *num*] [-xm *num*] [-df *num*] [-nl|-nb] [-as] [-rnd] *path* *rnum* [*bnum* [*apow* [*fpow*]]]

Store records with keys of 8 bytes. They change as `00000001', `00000002'...

tchtest read [-mt] [-rc *num*] [-xm *num*] [-df *num*] [-nl|-nb] [-wb] [-rnd] *path*

Retrieve all records of the database above.

tchtest remove [-mt] [-rc *num*] [-xm *num*] [-df *num*] [-nl|-nb] [-rnd] *path*

Remove all records of the database above.

tchtest rcat [-mt] [-tl] [-td|-tb|-tt|-tx] [-rc *num*] [-xm *num*] [-df *num*] [-nl|-nb] [-pn *num*] [-dai|-dad|-rl|-ru] *path rnum* [*bnum* [*apow* [*fpow*]]]

Store records with partway duplicated keys using concatenate mode.

tchtest misc [-mt] [-tl] [-td|-tb|-tt|-tx] [-nl|-nb] *path rnum*

Perform miscellaneous test of various operations.

tchtest wicked [-mt] [-tl] [-td|-tb|-tt|-tx] [-nl|-nb] *path rnum*

Perform updating operations selected at random.

Options feature the following.

- **-mt** : call the function `tchdbsetmutex'.
- **-tl** : enable the option `HDBTLARGE'.
- **-td** : enable the option `HDBTDEFLATE'.
- **-tb** : enable the option `HDBTBZIP'.
- **-tt** : enable the option `HDBTTCBS'.
- **-tx** : enable the option `HDBTEXCODEC'.
- **-rc *num*** : specify the number of cached records.
- **-xm *num*** : specify the size of the extra mapped memory.
- **-df *num*** : specify the unit step number of auto defragmentation.
- **-nl** : enable the option `HDBNOLCK'.
- **-nb** : enable the option `HDBLCKNB'.
- **-as** : use the function `tchdbputasync' instead of `tchdbput'.
- **-rnd** : select keys at random.
- **-wb** : use the function `tchdbget3' instead of `tchdbget'.
- **-pn *num*** : specify the number of patterns.
- **-dai** : use the function `tchdbaddint' instead of `tchdbputcat'.
- **-dad** : use the function `tchdbadddouble' instead of `tchdbputcat'.
- **-rl** : set the length of values at random.
- **-ru** : select update operations at random.

This command returns 0 on success, another on failure.

The command `**tchmttest**' is a utility for facility test under multi-thread situation. This command is used in the following format. `*path*' specifies the path of a database file. `*tnum*' specifies the number of running threads. `*rnum*' specifies the number of iterations. `*bnum*' specifies the number of buckets. `*apow*' specifies the power of the alignment. `*fpow*' specifies the power of the free block pool.

tchmttest write [-tl] [-td|-tb|-tt|-tx] [-rc *num*] [-xm *num*] [-df *num*] [-nl|-nb] [-as] [-rnd] *path tnum rnum* [*bnum* [*apow* [*fpow*]]]

Store records with keys of 8 bytes. They change as `00000001', `00000002'...

tchmttest read [-rc *num*] [-xm *num*] [-df *num*] [-nl|-nb] [-wb] [-rnd] *path tnum*

Retrieve all records of the database above.

tchmttest remove [-rc *num*] [-xm *num*] [-df *num*] [-nl|-nb] [-rnd] *path tnum*

Remove all records of the database above.

tchmttest wicked [-tl] [-td|-tb|-tt|-tx] [-nl|-nb] [-nc] ***path tnum rnum***

Perform updating operations selected at random.

tchmttest typical [-tl] [-td|-tb|-tt|-tx] [-rc ***num***] **[-xm** ***num***] **[-df** ***num***] **[-nl|-nb] [-nc] [-rr** ***num***] ***path tnum rnum*** **[*bnum* [*apow* [*fpow*]]**

Perform typical operations selected at random.

tchmttest race [-tl] [-td|-tb|-tt|-tx] [-xm ***num***] **[-df** ***num***] **[-nl|-nb]** ***path tnum rnum*** **[*bnum* [*apow* [*fpow*]]**

Perform race condition test.

Options feature the following.

- **-tl** : enable the option `HDBTLARGE'.
- **-td** : enable the option `HDBTDEFLATE'.
- **-tb** : enable the option `HDBTBZIP'.
- **-tt** : enable the option `HDBTTCBS'.
- **-tx** : enable the option `HDBTEXCODEC'.
- **-rc** ***num*** : specify the number of cached records.
- **-xm** ***num*** : specify the size of the extra mapped memory.
- **-df** ***num*** : specify the unit step number of auto defragmentation.
- **-nl** : enable the option `HDBNOLCK'.
- **-nb** : enable the option `HDBLCKNB'.
- **-as** : use the function `tchdbputasync' instead of `tchdbput'.
- **-rnd** : select keys at random.
- **-wb** : use the function `tchdbget3' instead of `tchdbget'.
- **-nc** : omit the comparison test.
- **-rr** ***num*** : specify the ratio of reading operation by percentage.

This command returns 0 on success, another on failure.

The command `**tchmgr**' is a utility for test and debugging of the hash database API and its applications. `*path*' specifies the path of a database file. `*bnum*' specifies the number of buckets. `*apow*' specifies the power of the alignment. `*fpow*' specifies the power of the free block pool. `*key*' specifies the key of a record. `*value*' specifies the value of a record. `*file*' specifies the input file.

tchmgr create [-tl] [-td|-tb|-tt|-tx] ***path*** **[*bnum* [*apow* [*fpow*]]]**

Create a database file.

tchmgr inform [-nl|-nb] ***path***

Print miscellaneous information to the standard output.

tchmgr put [-nl|-nb] [-sx] [-dk|-dc|-dai|-dad] ***path key value***

Store a record.

tchmgr out [-nl|-nb] [-sx] ***path key***

Remove a record.

tchmgr get [-nl|-nb] [-sx] [-px] [-pz] ***path key***

Print the value of a record.

tchmgr list [-nl|-nb] [-m ***num***] **[-pv] [-px] [-fm** ***str***] ***path***

Print keys of all records, separated by line feeds.

tchmgr optimize [-tl] [-td|-tb|-tt|-tx] [-tz] [-nl|-nb] [-df] ***path*** **[*bnum* [*apow* [*fpow*]]]**

Optimize a database file.

tchmgr importtsv [-nl|-nb] [-sc] ***path*** **[*file*]**

Store records of TSV in each line of a file.

tchmgr version

Print the version information of Tokyo Cabinet.

Options feature the following.

- **-tl** : enable the option `HDBTLARGE'.
- **-td** : enable the option `HDBTDEFLATE'.
- **-tb** : enable the option `HDBTBZIP'.
- **-tt** : enable the option `HDBTTCBS'.
- **-tx** : enable the option `HDBTEXCODEC'.
- **-nl** : enable the option `HDBNOLCK'.
- **-nb** : enable the option `HDBLCKNB'.
- **-sx** : the input data is evaluated as a hexadecimal data string.
- **-dk** : use the function `tchdbputkeep' instead of `tchdbput'.
- **-dc** : use the function `tchdbputcat' instead of `tchdbput'.
- **-dai** : use the function `tchdbaddint' instead of `tchdbput'.
- **-dad** : use the function `tchdbadddouble' instead of `tchdbput'.
- **-px** : the output data is converted into a hexadecimal data string.
- **-pz** : do not append line feed at the end of the output.
- **-m** ***num*** : specify the maximum number of the output.
- **-pv** : print values of records also.
- **-fm** ***str*** : specify the prefix of keys.
- **-tz** : enable the option `UINT8_MAX'.
- **-df** : perform defragmentation only.
- **-sc** : normalize keys as lower cases.

This command returns 0 on success, another on failure.

The B+ Tree Database API

B+ tree database is a file containing a B+ tree and is handled with the B+ tree database API. See `**tcbdb.h**' for the entire specification.

Description

To use the B+ tree database API, include `**tcutil.h**', `**tcbdb.h**', and related standard header files. Usually, write the following description near the front of a source file.

```
#include <tcutil.h>
#include <tcbdb.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
```

Objects whose type is pointer to `**TCBDB**' are used to handle B+ tree databases. A B+ tree database object is created with the function `**tcbdbnew**' and is deleted with the function `**tcbdbdel**'. To avoid memory leak, it is important to delete every object when it is no longer in use.

Before operations to store or retrieve records, it is necessary to open a database file and connect the B+ tree database object to it. The function `**tcbdbopen**' is used to open a database file and the function `**tcbdbclose**' is used to close the database file. To avoid data missing or corruption, it is important to close every database file when it is no longer in use. It is forbidden for multible database objects in a process to open the same database at the same time.

API

The function `tcbdberrmsg' is used in order to get the message string corresponding to an error code.

```
const char *tcbdberrmsg(int ecode);
```

`*ecode*' specifies the error code.
The return value is the message string of the error code.

The function `tcbdbnew' is used in order to create a B+ tree database object.

```
TCBDB *tcbdbnew(void);
```

The return value is the new B+ tree database object.

The function `tcbdbdel' is used in order to delete a B+ tree database object.

```
void tcbdbdel(TCBDB *bdb);
```

`*bdb*' specifies the B+ tree database object.
If the database is not closed, it is closed implicitly. Note that the deleted object and its derivatives can not be used anymore.

The function `tcbdbecode' is used in order to get the last happened error code of a B+ tree database object.

int tcbdbecode(TCBDB *_bdb_);

`_bdb_' specifies the B+ tree database object.
The return value is the last happened error code.
The following error codes are defined: `TCESUCCESS' for success, `TCETHREAD' for threading error, `TCEINVALID' for invalid operation, `TCENOFILE' for file not found, `TCENOPERM' for no permission, `TCEMETA' for invalid meta data, `TCERHEAD' for invalid record header, `TCEOPEN' for open error, `TCECLOSE' for close error, `TCETRUNC' for trunc error, `TCESYNC' for sync error, `TCESTAT' for stat error, `TCESEEK' for seek error, `TCEREAD' for read error, `TCEWRITE' for write error, `TCEMMAP' for mmap error, `TCELOCK' for lock error, `TCEUNLINK' for unlink error, `TCERENAME' for rename error, `TCEMKDIR' for mkdir error, `TCERMDIR' for rmdir error, `TCEKEEP' for existing record, `TCENOREC' for no record found, and `TCEMISC' for miscellaneous error.

The function `tcbdbsetmutex' is used in order to set mutual exclusion control of a B+ tree database object for threading.

bool tcbdbsetmutex(TCBDB *_bdb_);

`_bdb_' specifies the B+ tree database object which is not opened.
If successful, the return value is true, else, it is false.
Note that the mutual exclusion control of the database should be set before the database is opened.

The function `tcbdbsetcmpfunc' is used in order to set the custom comparison function of a B+ tree database object.

bool tcbdbsetcmpfunc(TCBDB *_bdb_, TCCMP _cmp_, void *_cmpop_);

`_bdb_' specifies the B+ tree database object which is not opened.
`_cmp_' specifies the pointer to the custom comparison function. It receives five parameters. The first parameter is the pointer to the region of one key. The second parameter is the size of the region of one key. The third parameter is the pointer to the region of the other key. The fourth parameter is the size of the region of the other key. The fifth parameter is the pointer to the optional opaque object. It returns positive if the former is big, negative if the latter is big, 0 if both are equivalent.
`_cmpop_' specifies an arbitrary pointer to be given as a parameter of the comparison function. If it is not needed, `NULL' can be specified.
If successful, the return value is true, else, it is false.
The default comparison function compares keys of two records by lexical order. The functions `tccmplexical' (dafault), `tccmpdecimal', `tccmpint32', and `tccmpint64' are built-in. Note that the comparison function should be set before the database is opened. Moreover, user-defined comparison functions should be set every time the database is being opened.

The function `tcbdbtune' is used in order to set the tuning parameters of a B+ tree database object.

bool tcbdbtune(TCBDB *_bdb_, int32_t _lmemb_, int32_t _nmemb_, int64_t _bnum_, int8_t _apow_, int8_t _fpow_, uint8_t _opts_);

`_bdb_' specifies the B+ tree database object which is not opened.
`_lmemb_' specifies the number of members in each leaf page. If it is not more than 0, the default value is specified. The default value is 128.
`_nmemb_' specifies the number of members in each non-leaf page. If it is not more than 0, the default

value is specified. The default value is 256.
`*bnum*' specifies the number of elements of the bucket array. If it is not more than 0, the default value is specified. The default value is 32749. Suggested size of the bucket array is about from 1 to 4 times of the number of all pages to be stored.
`*apow*' specifies the size of record alignment by power of 2. If it is negative, the default value is specified. The default value is 8 standing for 2^8=256.
`*fpow*' specifies the maximum number of elements of the free block pool by power of 2. If it is negative, the default value is specified. The default value is 10 standing for 2^10=1024.
`*opts*' specifies options by bitwise-or: `BDBTLARGE' specifies that the size of the database can be larger than 2GB by using 64-bit bucket array, `BDBTDEFLATE' specifies that each page is compressed with Deflate encoding, `BDBTBZIP' specifies that each page is compressed with BZIP2 encoding, `BDBTTCBS' specifies that each page is compressed with TCBS encoding.
If successful, the return value is true, else, it is false.
Note that the tuning parameters should be set before the database is opened.

The function `tcbdbsetcache' is used in order to set the caching parameters of a B+ tree database object.

```
bool tcbdbsetcache(TCBDB *bdb, int32_t lcnum, int32_t ncnum);
```

`*bdb*' specifies the B+ tree database object which is not opened.
`*lcnum*' specifies the maximum number of leaf nodes to be cached. If it is not more than 0, the default value is specified. The default value is 1024.
`*ncnum*' specifies the maximum number of non-leaf nodes to be cached. If it is not more than 0, the default value is specified. The default value is 512.
If successful, the return value is true, else, it is false.
Note that the caching parameters should be set before the database is opened.

The function `tcbdbsetxmsiz' is used in order to set the size of the extra mapped memory of a B+ tree database object.

```
bool tcbdbsetxmsiz(TCBDB *bdb, int64_t xmsiz);
```

`*bdb*' specifies the B+ tree database object which is not opened.
`*xmsiz*' specifies the size of the extra mapped memory. If it is not more than 0, the extra mapped memory is disabled. It is disabled by default.
If successful, the return value is true, else, it is false.
Note that the mapping parameters should be set before the database is opened.

The function `tcbdbsetdfunit' is used in order to set the unit step number of auto defragmentation of a B+ tree database object.

```
bool tcbdbsetdfunit(TCBDB *bdb, int32_t dfunit);
```

`*bdb*' specifies the B+ tree database object which is not opened.
`*dfunit*' specifie the unit step number. If it is not more than 0, the auto defragmentation is disabled. It is disabled by default.
If successful, the return value is true, else, it is false.
Note that the defragmentation parameter should be set before the database is opened.

The function `tcbdbopen' is used in order to open a database file and connect a B+ tree database object.

```
bool tcbdbopen(TCBDB *bdb, const char *path, int omode);
```

`*bdb*' specifies the B+ tree database object which is not opened.
`*path*' specifies the path of the database file.
`*omode*' specifies the connection mode: `BDBOWRITER' as a writer, `BDBOREADER' as a reader. If the mode is `BDBOWRITER', the following may be added by bitwise-or: `BDBOCREAT', which means it creates a new database if not exist, `BDBOTRUNC', which means it creates a new database regardless if one exists, `BDBOTSYNC', which means every transaction synchronizes updated contents with the device. Both of `BDBOREADER' and `BDBOWRITER' can be added to by bitwise-or: `BDBONOLCK', which means it opens the database file without file locking, or `BDBOLCKNB', which means locking is performed without blocking.
If successful, the return value is true, else, it is false.

The function `tcbdbclose' is used in order to close a B+ tree database object.

```
bool tcbdbclose(TCBDB *bdb);
```

`*bdb*' specifies the B+ tree database object.
If successful, the return value is true, else, it is false.
Update of a database is assured to be written when the database is closed. If a writer opens a database but does not close it appropriately, the database will be broken.

The function `tcbdbput' is used in order to store a record into a B+ tree database object.

```
bool tcbdbput(TCBDB *bdb, const void *kbuf, int ksiz, const void *vbuf, int vsiz);
```

`*bdb*' specifies the B+ tree database object connected as a writer.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tcbdbput2' is used in order to store a string record into a B+ tree database object.

```
bool tcbdbput2(TCBDB *bdb, const char *kstr, const char *vstr);
```

`*bdb*' specifies the B+ tree database object connected as a writer.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tcbdbputkeep' is used in order to store a new record into a B+ tree database object.

```
bool tcbdbputkeep(TCBDB *bdb, const void *kbuf, int ksiz, const void *vbuf, int
vsiz);
```

`*bdb*' specifies the B+ tree database object connected as a writer.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.

If a record with the same key exists in the database, this function has no effect.

The function `tcbdbputkeep2' is used in order to store a new string record into a B+ tree database object.

bool tcbdbputkeep2(TCBDB *_bdb_, const char *_kstr_, const char *_vstr_);

`_bdb_' specifies the B+ tree database object connected as a writer.
`_kstr_' specifies the string of the key.
`_vstr_' specifies the string of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tcbdbputcat' is used in order to concatenate a value at the end of the existing record in a B+ tree database object.

bool tcbdbputcat(TCBDB *_bdb_, const void *_kbuf_, int _ksiz_, const void *_vbuf_, int _vsiz_);

`_bdb_' specifies the B+ tree database object connected as a writer.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_vbuf_' specifies the pointer to the region of the value.
`_vsiz_' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tcbdbputcat2' is used in order to concatenate a string value at the end of the existing record in a B+ tree database object.

bool tcbdbputcat2(TCBDB *_bdb_, const char *_kstr_, const char *_vstr_);

`_bdb_' specifies the B+ tree database object connected as a writer.
`_kstr_' specifies the string of the key.
`_vstr_' specifies the string of the value.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tcbdbputdup' is used in order to store a record into a B+ tree database object with allowing duplication of keys.

bool tcbdbputdup(TCBDB *_bdb_, const void *_kbuf_, int _ksiz_, const void *_vbuf_, int _vsiz_);

`_bdb_' specifies the B+ tree database object connected as a writer.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_vbuf_' specifies the pointer to the region of the value.
`_vsiz_' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, the new record is placed after the existing one.

The function `tcbdbputdup2' is used in order to store a string record into a B+ tree database object with

allowing duplication of keys.

bool tcbdbputdup2(TCBDB *bdb, const char *kstr, const char *vstr);

`bdb' specifies the B+ tree database object connected as a writer.
`kstr' specifies the string of the key.
`vstr' specifies the string of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, the new record is placed after the existing one.

The function `tcbdbputdup3' is used in order to store records into a B+ tree database object with allowing duplication of keys.

bool tcbdbputdup3(TCBDB *bdb, const void *kbuf, int ksiz, const TCLIST *vals);

`bdb' specifies the B+ tree database object connected as a writer.
`kbuf' specifies the pointer to the region of the common key.
`ksiz' specifies the size of the region of the common key.
`vals' specifies a list object containing values.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, the new records are placed after the existing one.

The function `tcbdbout' is used in order to remove a record of a B+ tree database object.

bool tcbdbout(TCBDB *bdb, const void *kbuf, int ksiz);

`bdb' specifies the B+ tree database object connected as a writer.
`kbuf' specifies the pointer to the region of the key.
`ksiz' specifies the size of the region of the key.
If successful, the return value is true, else, it is false.
If the key of duplicated records is specified, the first one is selected.

The function `tcbdbout2' is used in order to remove a string record of a B+ tree database object.

bool tcbdbout2(TCBDB *bdb, const char *kstr);

`bdb' specifies the B+ tree database object connected as a writer.
`kstr' specifies the string of the key.
If successful, the return value is true, else, it is false.
If the key of duplicated records is specified, the first one is selected.

The function `tcbdbout3' is used in order to remove records of a B+ tree database object.

bool tcbdbout3(TCBDB *bdb, const void *kbuf, int ksiz);

`bdb' specifies the B+ tree database object connected as a writer.
`kbuf' specifies the pointer to the region of the key.
`ksiz' specifies the size of the region of the key.
If successful, the return value is true, else, it is false.
If the key of duplicated records is specified, all of them are removed.

The function `tcbdbget' is used in order to retrieve a record in a B+ tree database object.

void *tcbdbget(TCBDB *bdb, const void *kbuf, int ksiz, int *sp);

`bdb' specifies the B+ tree database object.

`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the value of the corresponding record. `NULL' is returned if no record corresponds.
If the key of duplicated records is specified, the first one is selected. Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcbdbget2' is used in order to retrieve a string record in a B+ tree database object.

```
char *tcbdbget2(TCBDB *bdb, const char *kstr);
```

`*bdb*' specifies the B+ tree database object.
`*kstr*' specifies the string of the key.
If successful, the return value is the string of the value of the corresponding record. `NULL' is returned if no record corresponds.
If the key of duplicated records is specified, the first one is selected. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcbdbget3' is used in order to retrieve a record in a B+ tree database object as a volatile buffer.

```
const void *tcbdbget3(TCBDB *bdb, const void *kbuf, int ksiz, int *sp);
```

`*bdb*' specifies the B+ tree database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the value of the corresponding record. `NULL' is returned if no record corresponds.
If the key of duplicated records is specified, the first one is selected. Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is volatile and it may be spoiled by another operation of the database, the data should be copied into another involatile buffer immediately.

The function `tcbdbget4' is used in order to retrieve records in a B+ tree database object.

```
TCLIST *tcbdbget4(TCBDB *bdb, const void *kbuf, int ksiz);
```

`*bdb*' specifies the B+ tree database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
If successful, the return value is a list object of the values of the corresponding records. `NULL' is returned if no record corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcbdbvnum' is used in order to get the number of records corresponding a key in a B+ tree database object.

int tcbdbvnum(TCBDB *_bdb_, const void *_kbuf_, int _ksiz_);

`_bdb_' specifies the B+ tree database object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
If successful, the return value is the number of the corresponding records, else, it is 0.

The function `tcbdbvnum2' is used in order to get the number of records corresponding a string key in a B+ tree database object.

int tcbdbvnum2(TCBDB *_bdb_, const char *_kstr_);

`_bdb_' specifies the B+ tree database object.
`_kstr_' specifies the string of the key.
If successful, the return value is the number of the corresponding records, else, it is 0.

The function `tcbdbvsiz' is used in order to get the size of the value of a record in a B+ tree database object.

int tcbdbvsiz(TCBDB *_bdb_, const void *_kbuf_, int _ksiz_);

`_bdb_' specifies the B+ tree database object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.
If the key of duplicated records is specified, the first one is selected.

The function `tcbdbvsiz2' is used in order to get the size of the value of a string record in a B+ tree database object.

int tcbdbvsiz2(TCBDB *_bdb_, const char *_kstr_);

`_bdb_' specifies the B+ tree database object.
`_kstr_' specifies the string of the key.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.
If the key of duplicated records is specified, the first one is selected.

The function `tcbdbrange' is used in order to get keys of ranged records in a B+ tree database object.

TCLIST *tcbdbrange(TCBDB *_bdb_, const void *_bkbuf_, int _bksiz_, bool _binc_, const void *_ekbuf_, int _eksiz_, bool _einc_, int _max_);

`_bdb_' specifies the B+ tree database object.
`_bkbuf_' specifies the pointer to the region of the key of the beginning border. If it is `NULL', the first record is specified.
`_bksiz_' specifies the size of the region of the beginning key.
`_binc_' specifies whether the beginning border is inclusive or not.
`_ekbuf_' specifies the pointer to the region of the key of the ending border. If it is `NULL', the last record is specified.
`_eksiz_' specifies the size of the region of the ending key.
`_einc_' specifies whether the ending border is inclusive or not.
`_max_' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.
The return value is a list object of the keys of the corresponding records. This function does never fail. It returns an empty list even if no record corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with

the function `tclistdel' when it is no longer in use.

The function `tcbdbrange2' is used in order to get string keys of ranged records in a B+ tree database object.

TCLIST *tcbdbrange2(TCBDB *_bdb_, const char *_bkstr_, bool _binc_, const char *_ekstr_, bool _einc_, int _max_);

`_bdb_' specifies the B+ tree database object.
`_bkstr_' specifies the string of the key of the beginning border. If it is `NULL', the first record is specified.
`_binc_' specifies whether the beginning border is inclusive or not.
`_ekstr_' specifies the string of the key of the ending border. If it is `NULL', the last record is specified.
`_einc_' specifies whether the ending border is inclusive or not.
`_max_' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.
The return value is a list object of the keys of the corresponding records. This function does never fail. It returns an empty list even if no record corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcbdbfwmkeys' is used in order to get forward matching keys in a B+ tree database object.

TCLIST *tcbdbfwmkeys(TCBDB *_bdb_, const void *_pbuf_, int _psiz_, int _max_);

`_bdb_' specifies the B+ tree database object.
`_pbuf_' specifies the pointer to the region of the prefix.
`_psiz_' specifies the size of the region of the prefix.
`_max_' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.
The return value is a list object of the corresponding keys. This function does never fail. It returns an empty list even if no key corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcbdbfwmkeys2' is used in order to get forward matching string keys in a B+ tree database object.

TCLIST *tcbdbfwmkeys2(TCBDB *_bdb_, const char *_pstr_, int _max_);

`_bdb_' specifies the B+ tree database object.
`_pstr_' specifies the string of the prefix.
`_max_' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.
The return value is a list object of the corresponding keys. This function does never fail. It returns an empty list even if no key corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcbdbaddint' is used in order to add an integer to a record in a B+ tree database object.

int tcbdbaddint(TCBDB *_bdb_, const void *_kbuf_, int _ksiz_, int _num_);

`_bdb_' specifies the B+ tree database object connected as a writer.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_num_' specifies the additional value.

If successful, the return value is the summation value, else, it is `INT_MIN'.
If the corresponding record exists, the value is treated as an integer and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tcbdbadddouble' is used in order to add a real number to a record in a B+ tree database object.

double tcbdbadddouble(TCBDB *_bdb_, const void *_kbuf_, int _ksiz_, double _num_);

`_bdb_' specifies the B+ tree database object connected as a writer.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_num_' specifies the additional value.
If successful, the return value is the summation value, else, it is Not-a-Number.
If the corresponding record exists, the value is treated as a real number and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tcbdbsync' is used in order to synchronize updated contents of a B+ tree database object with the file and the device.

bool tcbdbsync(TCBDB *_bdb_);

`_bdb_' specifies the B+ tree database object connected as a writer.
If successful, the return value is true, else, it is false.
This function is useful when another process connects to the same database file.

The function `tcbdboptimize' is used in order to optimize the file of a B+ tree database object.

bool tcbdboptimize(TCBDB *_bdb_, int32_t _lmemb_, int32_t _nmemb_, int64_t _bnum_, int8_t _apow_, int8_t _fpow_, uint8_t _opts_);

`_bdb_' specifies the B+ tree database object connected as a writer.
`_lmemb_' specifies the number of members in each leaf page. If it is not more than 0, the current setting is not changed.
`_nmemb_' specifies the number of members in each non-leaf page. If it is not more than 0, the current setting is not changed.
`_bnum_' specifies the number of elements of the bucket array. If it is not more than 0, the default value is specified. The default value is two times of the number of pages.
`_apow_' specifies the size of record alignment by power of 2. If it is negative, the current setting is not changed.
`_fpow_' specifies the maximum number of elements of the free block pool by power of 2. If it is negative, the current setting is not changed.
`_opts_' specifies options by bitwise-or: `BDBTLARGE' specifies that the size of the database can be larger than 2GB by using 64-bit bucket array, `BDBTDEFLATE' specifies that each record is compressed with Deflate encoding, `BDBTBZIP' specifies that each page is compressed with BZIP2 encoding, `BDBTTCBS' specifies that each page is compressed with TCBS encoding. If it is `UINT8_MAX', the current setting is not changed.
If successful, the return value is true, else, it is false.
This function is useful to reduce the size of the database file with data fragmentation by successive updating.

The function `tcbdbvanish' is used in order to remove all records of a B+ tree database object.

bool tcbdbvanish(TCBDB *_bdb_);

`_bdb_' specifies the B+ tree database object connected as a writer.
If successful, the return value is true, else, it is false.

The function `tcbdbcopy' is used in order to copy the database file of a B+ tree database object.

bool tcbdbcopy(TCBDB *_bdb_, const char *_path_);

`_bdb_' specifies the B+ tree database object.
`_path_' specifies the path of the destination file. If it begins with `@', the trailing substring is executed as a command line.
If successful, the return value is true, else, it is false. False is returned if the executed command returns non-zero code.
The database file is assured to be kept synchronized and not modified while the copying or executing operation is in progress. So, this function is useful to create a backup file of the database file.

The function `tcbdbtranbegin' is used in order to begin the transaction of a B+ tree database object.

bool tcbdbtranbegin(TCBDB *_bdb_);

`_bdb_' specifies the B+ tree database object connected as a writer.
If successful, the return value is true, else, it is false.
The database is locked by the thread while the transaction so that only one transaction can be activated with a database object at the same time. Thus, the serializable isolation level is assumed if every database operation is performed in the transaction. Because all pages are cached on memory while the transaction, the amount of referred records is limited by the memory capacity. If the database is closed during transaction, the transaction is aborted implicitly.

The function `tcbdbtrancommit' is used in order to commit the transaction of a B+ tree database object.

bool tcbdbtrancommit(TCBDB *_bdb_);

`_bdb_' specifies the B+ tree database object connected as a writer.
If successful, the return value is true, else, it is false.
Update in the transaction is fixed when it is committed successfully.

The function `tcbdbtranabort' is used in order to abort the transaction of a B+ tree database object.

bool tcbdbtranabort(TCBDB *_bdb_);

`_bdb_' specifies the B+ tree database object connected as a writer.
If successful, the return value is true, else, it is false.
Update in the transaction is discarded when it is aborted. The state of the database is rollbacked to before transaction.

The function `tcbdbpath' is used in order to get the file path of a B+ tree database object.

const char *tcbdbpath(TCBDB *_bdb_);

`_bdb_' specifies the B+ tree database object.
The return value is the path of the database file or `NULL' if the object does not connect to any database file.

The function `tcbdbrnum' is used in order to get the number of records of a B+ tree database object.

uint64_t tcbdbrnum(TCBDB *bdb);

`bdb' specifies the B+ tree database object.

The return value is the number of records or 0 if the object does not connect to any database file.

The function `tcbdbfsiz' is used in order to get the size of the database file of a B+ tree database object.

uint64_t tcbdbfsiz(TCBDB *bdb);

`bdb' specifies the B+ tree database object.

The return value is the size of the database file or 0 if the object does not connect to any database file.

The function `tcbdbcurnew' is used in order to create a cursor object.

BDBCUR *tcbdbcurnew(TCBDB *bdb);

`bdb' specifies the B+ tree database object.

The return value is the new cursor object.

Note that the cursor is available only after initialization with the `tcbdbcurfirst' or the `tcbdbcurjump' functions and so on. Moreover, the position of the cursor will be indefinite when the database is updated after the initialization of the cursor.

The function `tcbdbcurdel' is used in order to delete a cursor object.

void tcbdbcurdel(BDBCUR *cur);

`cur' specifies the cursor object.

The function `tcbdbcurfirst' is used in order to move a cursor object to the first record.

bool tcbdbcurfirst(BDBCUR *cur);

`cur' specifies the cursor object.

If successful, the return value is true, else, it is false. False is returned if there is no record in the database.

The function `tcbdbcurlast' is used in order to move a cursor object to the last record.

bool tcbdbcurlast(BDBCUR *cur);

`cur' specifies the cursor object.

If successful, the return value is true, else, it is false. False is returned if there is no record in the database.

The function `tcbdbcurjump' is used in order to move a cursor object to the front of records corresponding a key.

bool tcbdbcurjump(BDBCUR *cur, const void *kbuf, int ksiz);

`cur' specifies the cursor object.

`kbuf' specifies the pointer to the region of the key.

`ksiz' specifies the size of the region of the key.

If successful, the return value is true, else, it is false. False is returned if there is no record corresponding the condition.

The cursor is set to the first record corresponding the key or the next substitute if completely matching record does not exist.

The function `tcbdbcurjump2' is used in order to move a cursor object to the front of records corresponding a key string.

```
bool tcbdbcurjump2(BDBCUR *cur, const char *kstr);
```

`*cur*' specifies the cursor object.
`*kstr*' specifies the string of the key.
If successful, the return value is true, else, it is false. False is returned if there is no record corresponding the condition.
The cursor is set to the first record corresponding the key or the next substitute if completely matching record does not exist.

The function `tcbdbcurprev' is used in order to move a cursor object to the previous record.

```
bool tcbdbcurprev(BDBCUR *cur);
```

`*cur*' specifies the cursor object.
If successful, the return value is true, else, it is false. False is returned if there is no previous record.

The function `tcbdbcurnext' is used in order to move a cursor object to the next record.

```
bool tcbdbcurnext(BDBCUR *cur);
```

`*cur*' specifies the cursor object.
If successful, the return value is true, else, it is false. False is returned if there is no next record.

The function `tcbdbcurput' is used in order to insert a record around a cursor object.

```
bool tcbdbcurput(BDBCUR *cur, const void *vbuf, int vsiz, int cpmode);
```

`*cur*' specifies the cursor object of writer connection.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
`*cpmode*' specifies detail adjustment: `BDBCPCURRENT', which means that the value of the current record is overwritten, `BDBCPBEFORE', which means that the new record is inserted before the current record, `BDBCPAFTER', which means that the new record is inserted after the current record.
If successful, the return value is true, else, it is false. False is returned when the cursor is at invalid position.
After insertion, the cursor is moved to the inserted record.

The function `tcbdbcurput2' is used in order to insert a string record around a cursor object.

```
bool tcbdbcurput2(BDBCUR *cur, const char *vstr, int cpmode);
```

`*cur*' specifies the cursor object of writer connection.
`*vstr*' specifies the string of the value.
`*cpmode*' specifies detail adjustment: `BDBCPCURRENT', which means that the value of the current record is overwritten, `BDBCPBEFORE', which means that the new record is inserted before the current record, `BDBCPAFTER', which means that the new record is inserted after the current record.
If successful, the return value is true, else, it is false. False is returned when the cursor is at invalid position.
After insertion, the cursor is moved to the inserted record.

The function `tcbdbcurout' is used in order to remove the record where a cursor object is.

bool tcbdbcurout(BDBCUR *_cur_);

`_cur_' specifies the cursor object of writer connection.

If successful, the return value is true, else, it is false. False is returned when the cursor is at invalid position.

After deletion, the cursor is moved to the next record if possible.

The function `tcbdbcurkey' is used in order to get the key of the record where the cursor object is.

void *tcbdbcurkey(BDBCUR *_cur_, int *_sp_);

`_cur_' specifies the cursor object.

`_sp_' specifies the pointer to the variable into which the size of the region of the return value is assigned.

If successful, the return value is the pointer to the region of the key, else, it is `NULL'. `NULL' is returned when the cursor is at invalid position.

Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcbdbcurkey2' is used in order to get the key string of the record where the cursor object is.

char *tcbdbcurkey2(BDBCUR *_cur_);

`_cur_' specifies the cursor object.

If successful, the return value is the string of the key, else, it is `NULL'. `NULL' is returned when the cursor is at invalid position.

Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcbdbcurkey3' is used in order to get the key of the record where the cursor object is, as a volatile buffer.

const void *tcbdbcurkey3(BDBCUR *_cur_, int *_sp_);

`_cur_' specifies the cursor object.

`_sp_' specifies the pointer to the variable into which the size of the region of the return value is assigned.

If successful, the return value is the pointer to the region of the key, else, it is `NULL'. `NULL' is returned when the cursor is at invalid position.

Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is volatile and it may be spoiled by another operation of the database, the data should be copied into another involatile buffer immediately.

The function `tcbdbcurval' is used in order to get the value of the record where the cursor object is.

void *tcbdbcurval(BDBCUR *_cur_, int *_sp_);

`_cur_' specifies the cursor object.

`_sp_' specifies the pointer to the variable into which the size of the region of the return value is assigned.

If successful, the return value is the pointer to the region of the value, else, it is `NULL'. `NULL' is returned when the cursor is at invalid position.

Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcbdbcurval2' is used in order to get the value string of the record where the cursor object is.

`char *tcbdbcurval2(BDBCUR *cur);`

`*cur*' specifies the cursor object.

If successful, the return value is the string of the value, else, it is `NULL'. `NULL' is returned when the cursor is at invalid position.

Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcbdbcurval3' is used in order to get the value of the record where the cursor object is, as a volatile buffer.

`const void *tcbdbcurval3(BDBCUR *cur, int *sp);`

`*cur*' specifies the cursor object.

`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.

If successful, the return value is the pointer to the region of the value, else, it is `NULL'. `NULL' is returned when the cursor is at invalid position.

Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is volatile and it may be spoiled by another operation of the database, the data should be copied into another involatile buffer immediately.

The function `tcbdbcurrec' is used in order to get the key and the value of the record where the cursor object is.

`bool tcbdbcurrec(BDBCUR *cur, TCXSTR *kxstr, TCXSTR *vxstr);`

`*cur*' specifies the cursor object.

`*kxstr*' specifies the object into which the key is wrote down.

`*vxstr*' specifies the object into which the value is wrote down.

If successful, the return value is true, else, it is false. False is returned when the cursor is at invalid position.

Example Code

The following code is an example to use a B+ tree database.

```
#include <tcutil.h>
#include <tcbdb.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>

int main(int argc, char **argv){
  TCBDB *bdb;
  BDBCUR *cur;
  int ecode;
  char *key, *value;

  /* create the object */
  bdb = tcbdbnew();

  /* open the database */
  if(!tcbdbopen(bdb, "casket.tcb", BDBOWRITER | BDBOCREAT)){
    ecode = tcbdbecode(bdb):
```

```
    fprintf(stderr, "open error: %s\n", tcbdberrmsg(ecode));
  }

  /* store records */
  if(!tcbdbput2(bdb, "foo", "hop") ||
     !tcbdbput2(bdb, "bar", "step") ||
     !tcbdbput2(bdb, "baz", "jump")){
    ecode = tcbdbecode(bdb);
    fprintf(stderr, "put error: %s\n", tcbdberrmsg(ecode));
  }

  /* retrieve records */
  value = tcbdbget2(bdb, "foo");
  if(value){
    printf("%s\n", value);
    free(value);
  } else {
    ecode = tcbdbecode(bdb);
    fprintf(stderr, "get error: %s\n", tcbdberrmsg(ecode));
  }

  /* traverse records */
  cur = tcbdbcurnew(bdb);
  tcbdbcurfirst(cur);
  while((key = tcbdbcurkey2(cur)) != NULL){
    value = tcbdbcurval2(cur);
    if(value){
      printf("%s:%s\n", key, value);
      free(value);
    }
    free(key);
    tcbdbcurnext(cur);
  }
  tcbdbcurdel(cur);

  /* close the database */
  if(!tcbdbclose(bdb)){
    ecode = tcbdbecode(bdb);
    fprintf(stderr, "close error: %s\n", tcbdberrmsg(ecode));
  }

  /* delete the object */
  tcbdbdel(bdb);

  return 0;
}
```

CLI

To use the B+ tree database API easily, the commands `**tcbtest**', `**tcbmttest**', and `**tcbmgr**' are provided.

The command `**tcbtest**' is a utility for facility test and performance test. This command is used in the following format. `*path*' specifies the path of a database file. `*rnum*' specifies the number of iterations. `*lmemb*' specifies the number of members in each leaf page. `*nmemb*' specifies the number of members in each non-leaf page. `*bnum*' specifies the number of buckets. `*apow*' specifies the power of the alignment. `*fpow*' specifies the power of the free block pool.

tcbtest write [-mt] [-cd|-ci|-cj] [-tl] [-td|-tb|-tt|-tx] [-lc *num*] [-nc *num*] [-xm *num*] [-df *num*] [-ls *num*] [-ca *num*] [-nl|-nb] [-rnd] *path* *rnum* [*lmemb* [*nmemb* [*bnum*

[*apow* [*fpow*]]]]]

Store records with keys of 8 bytes. They change as `00000001', `00000002'...

tcbtest read [-mt] [-cd|-ci|-cj] [-lc *num*] [-nc *num*] [-xm *num*] [-df *num*] [-nl|-nb] [-wb] [-rnd] *path*

Retrieve all records of the database above.

tcbtest remove [-mt] [-cd|-ci|-cj] [-lc *num*] [-nc *num*] [-xm *num*] [-df *num*] [-nl|-nb] [-rnd] *path*

Remove all records of the database above.

tcbtest rcat [-mt] [-cd|-ci|-cj] [-tl] [-td|-tb|-tt|-tx] [-lc *num*] [-nc *num*] [-xm *num*] [-df *num*] [-ls *num*] [-ca *num*] [-nl|-nb] [-pn *num*] [-dai|-dad|-rl|-ru] *path* *rnum* [*lmemb* [*nmemb* [*bnum* [*apow* [*fpow*]]]]]

Store records with partway duplicated keys using concatenate mode.

tcbtest queue [-mt] [-cd|-ci|-cj] [-tl] [-td|-tb|-tt|-tx] [-lc *num*] [-nc *num*] [-xm *num*] [-df *num*] [-ls *num*] [-ca *num*] [-nl|-nb] *path* *rnum* [*lmemb* [*nmemb* [*bnum* [*apow* [*fpow*]]]]]

Perform queueing and dequeueing.

tcbtest misc [-mt] [-tl] [-td|-tb|-tt|-tx] [-nl|-nb] *path* *rnum*

Perform miscellaneous test of various operations.

tcbtest wicked [-mt] [-tl] [-td|-tb|-tt|-tx] [-nl|-nb] *path* *rnum*

Perform updating operations selected at random.

Options feature the following.

-mt : call the function `tchdbsetmutex'.
-cd : use the comparison function `tccmpdecimal'.
-ci : use the comparison function `tccmpint32'.
-cj : use the comparison function `tccmpint64'.
-tl : enable the option `BDBTLARGE'.
-td : enable the option `BDBTDEFLATE'.
-tb : enable the option `BDBTBZIP'.
-tt : enable the option `BDBTTCBS'.
-tx : enable the option `BDBTEXCODEC'.
-lc ***num*** : specify the number of cached leaf pages.
-nc ***num*** : specify the number of cached non-leaf pages.
-xm ***num*** : specify the size of the extra mapped memory.
-df ***num*** : specify the unit step number of auto defragmentation.
-ls ***num*** : specify the maximum size of each leaf page.
-ca ***num*** : specify the capacity number of records.
-nl : enable the option `BDBNOLCK'.
-nb : enable the option `BDBLCKNB'.
-rnd : select keys at random.
-wb : use the function `tcbdbget3' instead of `tcbdbget'.
-pn ***num*** : specify the number of patterns.
-dai : use the function `tcbdbaddint' instead of `tcbdbputcat'.
-dad : use the function `tcbdbadddouble' instead of `tcbdbputcat'.
-rl : set the length of values at random.

-ru : select update operations at random.

This command returns 0 on success, another on failure.

The command `**tcbmttest**' is a utility for facility test and performance test. This command is used in the following format. `*path*' specifies the path of a database file. `*tnum*' specifies the number of running threads. `*rnum*' specifies the number of iterations. `*lmemb*' specifies the number of members in each leaf page. `*nmemb*' specifies the number of members in each non-leaf page. `*bnum*' specifies the number of buckets. `*apow*' specifies the power of the alignment. `*fpow*' specifies the power of the free block pool.

tcbmttest write [-tl] [-td|-tb|-tt|-tx] [-xm *num*] [-df *num*] [-nl|-nb] [-rnd] *path* *tnum* *rnum* [*lmemb* [*nmemb* [*bnum* [*apow* [*fpow*]]]]]
: Store records with keys of 8 bytes. They change as `00000001', `00000002'...

tcbmttest read [-xm *num*] [-df *num*] [-nl|-nb] [-wb] [-rnd] *path* *tnum*
: Retrieve all records of the database above.

tcbmttest remove [-xm *num*] [-df *num*] [-nl|-nb] [-rnd] *path* *tnum*
: Remove all records of the database above.

tcbmttest wicked [-tl] [-td|-tb|-tt|-tx] [-nl|-nb] [-nc] *path* *tnum* *rnum*
: Perform updating operations selected at random.

tcbmttest typical [-tl] [-td|-tb|-tt|-tx] [-xm *num*] [-df *num*] [-nl|-nb] [-nc] [-rr *num*] *path* *tnum* *rnum* [*lmemb* [*nmemb* [*bnum* [*apow* [*fpow*]]]]]
: Perform typical operations selected at random.

tcbmttest race [-tl] [-td|-tb|-tt|-tx] [-xm *num*] [-df *num*] [-nl|-nb] *path* *tnum* *rnum* [*lmemb* [*nmemb* [*bnum* [*apow* [*fpow*]]]]]
: Perform race condition test.

Options feature the following.

-tl : enable the option `BDBTLARGE'.
-td : enable the option `BDBTDEFLATE'.
-tb : enable the option `BDBTBZIP'.
-tt : enable the option `BDBTTCBS'.
-tx : enable the option `BDBTEXCODEC'.
-xm *num* : specify the size of the extra mapped memory.
-df *num* : specify the unit step number of auto defragmentation.
-nl : enable the option `BDBNOLCK'.
-nb : enable the option `BDBLCKNB'.
-rnd : select keys at random.
-wb : use the function `tchdbget3' instead of `tchdbget'.
-nc : omit the comparison test.
-rr *num* : specify the ratio of reading operation by percentage.

This command returns 0 on success, another on failure.

The command `**tcbmgr**' is a utility for test and debugging of the B+ tree database API and its applications. `*path*' specifies the path of a database file. `*lmemb*' specifies the number of members in each leaf page. `*nmemb*' specifies the number of members in each non-leaf page. `*bnum*' specifies the number of buckets.

`*apow*' specifies the power of the alignment. `*fpow*' specifies the power of the free block pool. `*key*' specifies the key of a record. `*value*' specifies the value of a record. `*file*' specifies the input file.

tcbmgr create [-cd|-ci|-cj] [-tl] [-td|-tb|-tt|-tx] *path* [*lmemb* [*nmemb* [*bnum* [*apow* [*fpow*]]]]]
Create a database file.

tcbmgr inform [-nl|-nb] *path*
Print miscellaneous information to the standard output.

tcbmgr put [-cd|-ci|-cj] [-nl|-nb] [-sx] [-dk|-dc|-dd|-db|-dai|-dad] *path key value*
Store a record.

tcbmgr out [-cd|-ci|-cj] [-nl|-nb] [-sx] *path key*
Remove a record.

tcbmgr get [-cd|-ci|-cj] [-nl|-nb] [-sx] [-px] [-pz] *path key*
Print the value of a record.

tcbmgr list [-cd|-ci|-cj] [-nl|-nb] [-m *num*] [-bk] [-pv] [-px] [-j *str*] [-rb *bkey ekey*] [-fm *str*] *path*
Print keys of all records, separated by line feeds.

tcbmgr optimize [-cd|-ci|-cj] [-tl] [-td|-tb|-tt|-tx] [-tz] [-nl|-nb] [-df] *path* [*lmemb* [*nmemb* [*bnum* [*apow* [*fpow*]]]]]
Optimize a database file.

tcbmgr importtsv [-nl|-nb] [-sc] *path* [*file*]
Store records of TSV in each line of a file.

tcbmgr version
Print the version information of Tokyo Cabinet.

Options feature the following.

-cd : use the comparison function `tccmpdecimal'.
-ci : use the comparison function `tccmpint32'.
-cj : use the comparison function `tccmpint64'.
-tl : enable the option `BDBTLARGE'.
-td : enable the option `BDBTDEFLATE'.
-tb : enable the option `BDBTBZIP'.
-tt : enable the option `BDBTTCBS'.
-tx : enable the option `BDBTEXCODEC'.
-nl : enable the option `BDBNOLCK'.
-nb : enable the option `BDBLCKNB'.
-sx : the input data is evaluated as a hexadecimal data string.
-dk : use the function `tcbdbputkeep' instead of `tcbdbput'.
-dc : use the function `tcbdbputcat' instead of `tcbdbput'.
-dd : use the function `tcbdbputdup' instead of `tcbdbput'.
-db : use the function `tcbdbputdupback' instead of `tcbdbput'.
-dai : use the function `tcbdbaddint' instead of `tcbdbput'.
-dad : use the function `tcbdbadddouble' instead of `tcbdbput'.
-px : the output data is converted into a hexadecimal data string.
-pz : do not append line feed at the end of the output.

- **-m** ***num*** : specify the maximum number of the output.
- **-bk** : perform backword scanning.
- **-pv** : print values of records also.
- **-j** ***str*** : specify the key where the cursor jump to.
- **-rb** ***bkey ekey*** : specify the range of keys.
- **-fm** ***str*** : specify the prefix of keys.
- **-tz** : enable the option `UINT8_MAX'.
- **-df** : perform defragmentation only.
- **-sc** : normalize keys as lower cases.

This command returns 0 on success, another on failure.

The Fixed-length Database API

Fixed-length database is a file containing an array of fixed-length elements and is handled with the fixed-length database API. See `**tcfdb.h**' for the entire specification.

Description

To use the fixed-length database API, include `**tcutil.h**', `**tcfdb.h**', and related standard header files. Usually, write the following description near the front of a source file.

```
#include <tcutil.h>
#include <tcfdb.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
```

Objects whose type is pointer to `**TCFDB**' are used to handle fixed-length databases. A fixed-length database object is created with the function `**tcfdbnew**' and is deleted with the function `**tcfdbdel**'. To avoid memory leak, it is important to delete every object when it is no longer in use.

Before operations to store or retrieve records, it is necessary to open a database file and connect the fixed-length database object to it. The function `**tcfdbopen**' is used to open a database file and the function `**tcfdbclose**' is used to close the database file. To avoid data missing or corruption, it is important to close every database file when it is no longer in use. It is forbidden for multible database objects in a process to open the same database at the same time.

API

The function `tcfdberrmsg' is used in order to get the message string corresponding to an error code.

```
const char *tcfdberrmsg(int ecode);
```

`*ecode*' specifies the error code.
The return value is the message string of the error code.

The function `tcfdbnew' is used in order to create a fixed-length database object.

```
TCFDB *tcfdbnew(void);
```

The return value is the new fixed-length database object.

The function `tcfdbdel' is used in order to delete a fixed-length database object.

```
void tcfdbdel(TCFDB *fdb);
```

`*fdb*' specifies the fixed-length database object.
If the database is not closed, it is closed implicitly. Note that the deleted object and its derivatives can not be used anymore.

The function `tcfdbecode' is used in order to get the last happened error code of a fixed-length database object.

`int tcfdbecode(TCFDB *fdb);`

`*fdb*' specifies the fixed-length database object.
The return value is the last happened error code.
The following error codes are defined: `TCESUCCESS' for success, `TCETHREAD' for threading error, `TCEINVALID' for invalid operation, `TCENOFILE' for file not found, `TCENOPERM' for no permission, `TCEMETA' for invalid meta data, `TCERHEAD' for invalid record header, `TCEOPEN' for open error, `TCECLOSE' for close error, `TCETRUNC' for trunc error, `TCESYNC' for sync error, `TCESTAT' for stat error, `TCESEEK' for seek error, `TCEREAD' for read error, `TCEWRITE' for write error, `TCEMMAP' for mmap error, `TCELOCK' for lock error, `TCEUNLINK' for unlink error, `TCERENAME' for rename error, `TCEMKDIR' for mkdir error, `TCERMDIR' for rmdir error, `TCEKEEP' for existing record, `TCENOREC' for no record found, and `TCEMISC' for miscellaneous error.

The function `tcfdbsetmutex' is used in order to set mutual exclusion control of a fixed-length database object for threading.

`bool tcfdbsetmutex(TCFDB *fdb);`

`*fdb*' specifies the fixed-length database object which is not opened.
If successful, the return value is true, else, it is false.
Note that the mutual exclusion control is needed if the object is shared by plural threads and this function should be called before the database is opened.

The function `tcfdbtune' is used in order to set the tuning parameters of a fixed-length database object.

`bool tcfdbtune(TCFDB *fdb, int32_t width, int64_t limsiz);`

`*fdb*' specifies the fixed-length database object which is not opened.
`*width*' specifies the width of the value of each record. If it is not more than 0, the default value is specified. The default value is 255.
`*limsiz*' specifies the limit size of the database file. If it is not more than 0, the default value is specified. The default value is 268435456.
If successful, the return value is true, else, it is false.
Note that the tuning parameters should be set before the database is opened.

The function `tcfdbopen' is used in order to open a database file and connect a fixed-length database object.

`bool tcfdbopen(TCFDB *fdb, const char *path, int omode);`

`*fdb*' specifies the fixed-length database object which is not opened.
`*path*' specifies the path of the database file.
`*omode*' specifies the connection mode: `FDBOWRITER' as a writer, `FDBOREADER' as a reader. If the mode is `FDBOWRITER', the following may be added by bitwise-or: `FDBOCREAT', which means it creates a new database if not exist, `FDBOTRUNC', which means it creates a new database regardless if one exists, `FDBOTSYNC', which means every transaction synchronizes updated contents with the device. Both of `FDBOREADER' and `FDBOWRITER' can be added to by bitwise-or: `FDBONOLCK', which means it opens the database file without file locking, or `FDBOLCKNB', which means locking is performed without blocking.
If successful, the return value is true, else, it is false.

The function `tcfdbclose' is used in order to close a fixed-length database object.

bool tcfdbclose(TCFDB *_fdb_);

`_fdb_' specifies the fixed-length database object.
If successful, the return value is true, else, it is false.
Update of a database is assured to be written when the database is closed. If a writer opens a database but does not close it appropriately, the database will be broken.

The function `tcfdbput' is used in order to store a record into a fixed-length database object.

bool tcfdbput(TCFDB *_fdb_, int64_t _id_, const void *_vbuf_, int _vsiz_);

`_fdb_' specifies the fixed-length database object connected as a writer.
`_id_' specifies the ID number. It should be more than 0. If it is `FDBIDMIN', the minimum ID number of existing records is specified. If it is `FDBIDPREV', the number less by one than the minimum ID number of existing records is specified. If it is `FDBIDMAX', the maximum ID number of existing records is specified. If it is `FDBIDNEXT', the number greater by one than the maximum ID number of existing records is specified.
`_vbuf_' specifies the pointer to the region of the value.
`_vsiz_' specifies the size of the region of the value. If the size of the value is greater than the width tuning parameter of the database, the size is cut down to the width.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tcfdbput2' is used in order to store a record with a decimal key into a fixed-length database object.

bool tcfdbput2(TCFDB *_fdb_, const void *_kbuf_, int _ksiz_, const void *_vbuf_, int _vsiz_);

`_fdb_' specifies the fixed-length database object connected as a writer.
`_kbuf_' specifies the pointer to the region of the decimal key. It should be more than 0. If it is "min", the minimum ID number of existing records is specified. If it is "prev", the number less by one than the minimum ID number of existing records is specified. If it is "max", the maximum ID number of existing records is specified. If it is "next", the number greater by one than the maximum ID number of existing records is specified.
`_ksiz_' specifies the size of the region of the key.
`_vbuf_' specifies the pointer to the region of the value.
`_vsiz_' specifies the size of the region of the value. If the size of the value is greater than the width tuning parameter of the database, the size is cut down to the width.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tcfdbput3' is used in order to store a string record with a decimal key into a fixed-length database object.

bool tcfdbput3(TCFDB *_fdb_, const char *_kstr_, const void *_vstr_);

`_fdb_' specifies the fixed-length database object connected as a writer.
`_kstr_' specifies the string of the decimal key. It should be more than 0. If it is "min", the minimum ID number of existing records is specified. If it is "prev", the number less by one than the minimum ID number of existing records is specified. If it is "max", the maximum ID number of existing records is specified. If it is "next", the number greater by one than the maximum ID number of existing records is

specified.
`*vstr*' specifies the string of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tcfdbputkeep' is used in order to store a new record into a fixed-length database object.

```
bool tcfdbputkeep(TCFDB *fdb, int64_t id, const void *vbuf, int vsiz);
```

`*fdb*' specifies the fixed-length database object connected as a writer.
`*id*' specifies the ID number. It should be more than 0. If it is `FDBIDMIN', the minimum ID number of existing records is specified. If it is `FDBIDPREV', the number less by one than the minimum ID number of existing records is specified. If it is `FDBIDMAX', the maximum ID number of existing records is specified. If it is `FDBIDNEXT', the number greater by one than the maximum ID number of existing records is specified.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value. If the size of the value is greater than the width tuning parameter of the database, the size is cut down to the width.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tcfdbputkeep2' is used in order to store a new record with a decimal key into a fixed-length database object.

```
bool tcfdbputkeep2(TCFDB *fdb, const void *kbuf, int ksiz, const void *vbuf, int vsiz);
```

`*fdb*' specifies the fixed-length database object connected as a writer.
`*kbuf*' specifies the pointer to the region of the decimal key. It should be more than 0. If it is "min", the minimum ID number of existing records is specified. If it is "prev", the number less by one than the minimum ID number of existing records is specified. If it is "max", the maximum ID number of existing records is specified. If it is "next", the number greater by one than the maximum ID number of existing records is specified.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value. If the size of the value is greater than the width tuning parameter of the database, the size is cut down to the width.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tcfdbputkeep3' is used in order to store a new string record with a decimal key into a fixed-length database object.

```
bool tcfdbputkeep3(TCFDB *fdb, const char *kstr, const void *vstr);
```

`*fdb*' specifies the fixed-length database object connected as a writer.
`*kstr*' specifies the string of the decimal key. It should be more than 0. If it is "min", the minimum ID number of existing records is specified. If it is "prev", the number less by one than the minimum ID number of existing records is specified. If it is "max", the maximum ID number of existing records is specified. If it is "next", the number greater by one than the maximum ID number of existing records is specified.
`*vstr*' specifies the string of the value.

If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tcfdbputcat' is used in order to concatenate a value at the end of the existing record in a fixed-length database object.

bool tcfdbputcat(TCFDB *_fdb_, int64_t _id_, const void *_vbuf_, int _vsiz_);

`_fdb_' specifies the fixed-length database object connected as a writer.
`_id_' specifies the ID number. It should be more than 0. If it is `FDBIDMIN', the minimum ID number of existing records is specified. If it is `FDBIDPREV', the number less by one than the minimum ID number of existing records is specified. If it is `FDBIDMAX', the maximum ID number of existing records is specified. If it is `FDBIDNEXT', the number greater by one than the maximum ID number of existing records is specified.
`_vbuf_' specifies the pointer to the region of the value.
`_vsiz_' specifies the size of the region of the value. If the size of the value is greater than the width tuning parameter of the database, the size is cut down to the width.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tcfdbputcat2' is used in order to concatenate a value with a decimal key in a fixed-length database object.

bool tcfdbputcat2(TCFDB *_fdb_, const void *_kbuf_, int _ksiz_, const void *_vbuf_, int _vsiz_);

`_fdb_' specifies the fixed-length database object connected as a writer.
`_kbuf_' specifies the pointer to the region of the decimal key. It should be more than 0. If it is "min", the minimum ID number of existing records is specified. If it is "prev", the number less by one than the minimum ID number of existing records is specified. If it is "max", the maximum ID number of existing records is specified. If it is "next", the number greater by one than the maximum ID number of existing records is specified.
`_ksiz_' specifies the size of the region of the key.
`_vbuf_' specifies the pointer to the region of the value.
`_vsiz_' specifies the size of the region of the value. If the size of the value is greater than the width tuning parameter of the database, the size is cut down to the width.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tcfdbputcat3' is used in order to concatenate a string value with a decimal key in a fixed-length database object.

bool tcfdbputcat3(TCFDB *_fdb_, const char *_kstr_, const void *_vstr_);

`_fdb_' specifies the fixed-length database object connected as a writer.
`_kstr_' specifies the string of the decimal key. It should be more than 0. If it is "min", the minimum ID number of existing records is specified. If it is "prev", the number less by one than the minimum ID number of existing records is specified. If it is "max", the maximum ID number of existing records is specified. If it is "next", the number greater by one than the maximum ID number of existing records is specified.
`_vstr_' specifies the string of the value.

If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tcfdbout' is used in order to remove a record of a fixed-length database object.

```
bool tcfdbout(TCFDB *fdb, int64_t id);
```

`*fdb*' specifies the fixed-length database object connected as a writer.
`*id*' specifies the ID number. It should be more than 0. If it is `FDBIDMIN', the minimum ID number of existing records is specified. If it is `FDBIDMAX', the maximum ID number of existing records is specified.
If successful, the return value is true, else, it is false.

The function `tcfdbout2' is used in order to remove a record with a decimal key of a fixed-length database object.

```
bool tcfdbout2(TCFDB *fdb, const void *kbuf, int ksiz);
```

`*fdb*' specifies the fixed-length database object connected as a writer.
`*kbuf*' specifies the pointer to the region of the decimal key. It should be more than 0. If it is "min", the minimum ID number of existing records is specified. If it is "max", the maximum ID number of existing records is specified.
`*ksiz*' specifies the size of the region of the key.
If successful, the return value is true, else, it is false.

The function `tcfdbout3' is used in order to remove a string record with a decimal key of a fixed-length database object.

```
bool tcfdbout3(TCFDB *fdb, const char *kstr);
```

`*fdb*' specifies the fixed-length database object connected as a writer.
`*kstr*' specifies the string of the decimal key. It should be more than 0. If it is "min", the minimum ID number of existing records is specified. If it is "max", the maximum ID number of existing records is specified.
If successful, the return value is true, else, it is false.

The function `tcfdbget' is used in order to retrieve a record in a fixed-length database object.

```
void *tcfdbget(TCFDB *fdb, int64_t id, int *sp);
```

`*fdb*' specifies the fixed-length database object.
`*id*' specifies the ID number. It should be more than 0. If it is `FDBIDMIN', the minimum ID number of existing records is specified. If it is `FDBIDMAX', the maximum ID number of existing records is specified.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the value of the corresponding record. `NULL' is returned if no record corresponds.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcfdbget2' is used in order to retrieve a record with a decimal key in a fixed-length database object.

void *tcfdbget2(TCFDB *_fdb_, const void *_kbuf_, int _ksiz_, int *_sp_);

`_fdb_' specifies the fixed-length database object.

`_kbuf_' specifies the pointer to the region of the decimal key. It should be more than 0. If it is "min", the minimum ID number of existing records is specified. If it is "max", the maximum ID number of existing records is specified.

`_ksiz_' specifies the size of the region of the key.

`_sp_' specifies the pointer to the variable into which the size of the region of the return value is assigned.

If successful, the return value is the pointer to the region of the value of the corresponding record. `NULL' is returned if no record corresponds.

Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcfdbget3' is used in order to retrieve a string record with a decimal key in a fixed-length database object.

char *tcfdbget3(TCFDB *_fdb_, const char *_kstr_);

`_fdb_' specifies the fixed-length database object.

`_kstr_' specifies the string of the decimal key. It should be more than 0. If it is "min", the minimum ID number of existing records is specified. If it is "max", the maximum ID number of existing records is specified.

If successful, the return value is the string of the value of the corresponding record. `NULL' is returned if no record corresponds.

Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcfdbget4' is used in order to retrieve a record in a fixed-length database object and write the value into a buffer.

int tcfdbget4(TCFDB *_fdb_, int64_t _id_, void *_vbuf_, int _max_);

`_fdb_' specifies the fixed-length database object.

`_id_' specifies the ID number. It should be more than 0. If it is `FDBIDMIN', the minimum ID number of existing records is specified. If it is `FDBIDMAX', the maximum ID number of existing records is specified.

`_vbuf_' specifies the pointer to the buffer into which the value of the corresponding record is written.

`_max_' specifies the size of the buffer.

If successful, the return value is the size of the written data, else, it is -1. -1 is returned if no record corresponds to the specified key.

Note that an additional zero code is not appended at the end of the region of the writing buffer.

The function `tcfdbvsiz' is used in order to get the size of the value of a record in a fixed-length database object.

int tcfdbvsiz(TCFDB *_fdb_, int64_t _id_);

`_fdb_' specifies the fixed-length database object.

`_id_' specifies the ID number. It should be more than 0. If it is `FDBIDMIN', the minimum ID number of existing records is specified. If it is `FDBIDMAX', the maximum ID number of existing records is

specified.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tcfdbvsiz2' is used in order to get the size of the value with a decimal key in a fixed-length database object.

int tcfdbvsiz2(TCFDB *_fdb_, const void *_kbuf_, int _ksiz_);

`_fdb_' specifies the fixed-length database object.
`_kbuf_' specifies the pointer to the region of the decimal key. It should be more than 0. If it is "min", the minimum ID number of existing records is specified. If it is "max", the maximum ID number of existing records is specified.
`_ksiz_' specifies the size of the region of the key.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tcfdbvsiz3' is used in order to get the size of the string value with a decimal key in a fixed-length database object.

int tcfdbvsiz3(TCFDB *_fdb_, const char *_kstr_);

`_fdb_' specifies the fixed-length database object.
`_kstr_' specifies the string of the decimal key. It should be more than 0. If it is "min", the minimum ID number of existing records is specified. If it is "max", the maximum ID number of existing records is specified.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tcfdbiterinit' is used in order to initialize the iterator of a fixed-length database object.

bool tcfdbiterinit(TCFDB *_fdb_);

`_fdb_' specifies the fixed-length database object.
If successful, the return value is true, else, it is false.
The iterator is used in order to access the key of every record stored in a database.

The function `tcfdbiternext' is used in order to get the next ID number of the iterator of a fixed-length database object.

uint64_t tcfdbiternext(TCFDB *_fdb_);

`_fdb_' specifies the fixed-length database object.
If successful, the return value is the next ID number of the iterator, else, it is 0. 0 is returned when no record is to be get out of the iterator.
It is possible to access every record by iteration of calling this function. It is allowed to update or remove records whose keys are fetched while the iteration. The order of this traversal access method is ascending of the ID number.

The function `tcfdbiternext2' is used in order to get the next decimay key of the iterator of a fixed-length database object.

void *tcfdbiternext2(TCFDB *_fdb_, int *_sp_);

`_fdb_' specifies the fixed-length database object.
`_sp_' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the next decimal key, else, it is `NULL'.

`NULL' is returned when no record is to be get out of the iterator.

Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. It is possible to access every record by iteration of calling this function. It is allowed to update or remove records whose keys are fetched while the iteration. The order of this traversal access method is ascending of the ID number.

The function `tcfdbiternext3' is used in order to get the next decimay key string of the iterator of a fixed-length database object.

char *tcfdbiternext3(TCFDB *_fdb_);

`_fdb_' specifies the fixed-length database object.

If successful, the return value is the string of the next decimal key, else, it is `NULL'. `NULL' is returned when no record is to be get out of the iterator.

Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. It is possible to access every record by iteration of calling this function. It is allowed to update or remove records whose keys are fetched while the iteration. The order of this traversal access method is ascending of the ID number.

The function `tcfdbrange' is used in order to get range matching ID numbers in a fixed-length database object.

uint64_t *tcfdbrange(TCFDB *_fdb_, int64_t _lower_, int64_t _upper_, int _max_, int *_np_);

`_fdb_' specifies the fixed-length database object.

`_lower_' specifies the lower limit of the range. If it is `FDBIDMIN', the minimum ID is specified.

`_upper_' specifies the upper limit of the range. If it is `FDBIDMAX', the maximum ID is specified.

`_max_' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.

`_np_' specifies the pointer to the variable into which the number of elements of the return value is assigned.

If successful, the return value is the pointer to an array of ID numbers of the corresponding records. `NULL' is returned on failure. This function does never fail. It returns an empty array even if no key corresponds.

Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcfdbrange2' is used in order to get range matching decimal keys in a fixed-length database object.

TCLIST *tcfdbrange2(TCFDB *_fdb_, const void *_lbuf_, int _lsiz_, const void *_ubuf_, int _usiz_, int _max_);

`_fdb_' specifies the fixed-length database object.

`_lbuf_' specifies the pointer to the region of the lower key. If it is "min", the minimum ID number of existing records is specified.

`_lsiz_' specifies the size of the region of the lower key.

`_ubuf_' specifies the pointer to the region of the upper key. If it is "max", the maximum ID number of existing records is specified.

`_usiz_' specifies the size of the region of the upper key.

`_max_' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.

The return value is a list object of the corresponding decimal keys. This function does never fail. It returns an empty list even if no key corresponds.

Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use. Note that this function may be very slow because every key in the database is scanned.

The function `tcfdbrange3' is used in order to get range matching decimal keys with strings in a fixed-length database object.

`TCLIST *tcfdbrange3(TCFDB *`*`fdb`*`, const char *`*`lstr`*`, const char *`*`ustr`*`, int `*`max`*`);`

`*fdb*' specifies the fixed-length database object.

`*lstr*' specifies the string of the lower key. If it is "min", the minimum ID number of existing records is specified.

`*ustr*' specifies the string of the upper key. If it is "max", the maximum ID number of existing records is specified.

`*max*' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.

The return value is a list object of the corresponding decimal keys. This function does never fail. It returns an empty list even if no key corresponds.

Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use. Note that this function may be very slow because every key in the database is scanned.

The function `tcfdbrange4' is used in order to get keys with an interval notation in a fixed-length database object.

`TCLIST *tcfdbrange4(TCFDB *`*`fdb`*`, const void *`*`ibuf`*`, int `*`isiz`*`, int `*`max`*`);`

`*fdb*' specifies the fixed-length database object.

`*ibuf*' specifies the pointer to the region of the interval notation.

`*isiz*' specifies the size of the region of the interval notation.

`*max*' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.

The return value is a list object of the corresponding decimal keys. This function does never fail. It returns an empty list even if no key corresponds.

Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use. Note that this function may be very slow because every key in the database is scanned.

The function `tcfdbrange5' is used in order to get keys with an interval notation string in a fixed-length database object.

`TCLIST *tcfdbrange5(TCFDB *`*`fdb`*`, const void *`*`istr`*`, int `*`max`*`);`

`*fdb*' specifies the fixed-length database object.

`*istr*' specifies the pointer to the region of the interval notation string.

`*max*' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.

The return value is a list object of the corresponding decimal keys. This function does never fail. It returns an empty list even if no key corresponds.

Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use. Note that this function may be very slow because every key in the database is scanned.

The function `tcfdbaddint' is used in order to add an integer to a record in a fixed-length database object.

int tcfdbaddint(TCFDB *_fdb_, int64_t _id_, int _num_);

`*fdb*' specifies the fixed-length database object connected as a writer.
`*id*' specifies the ID number. It should be more than 0. If it is `FDBIDMIN', the minimum ID number of existing records is specified. If it is `FDBIDPREV', the number less by one than the minimum ID number of existing records is specified. If it is `FDBIDMAX', the maximum ID number of existing records is specified. If it is `FDBIDNEXT', the number greater by one than the maximum ID number of existing records is specified.
`*num*' specifies the additional value.
If successful, the return value is the summation value, else, it is `INT_MIN'.
If the corresponding record exists, the value is treated as an integer and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tcfdbadddouble' is used in order to add a real number to a record in a fixed-length database object.

double tcfdbadddouble(TCFDB *_fdb_, int64_t _id_, double _num_);

`*fdb*' specifies the fixed-length database object connected as a writer.
`*id*' specifies the ID number. It should be more than 0. If it is `FDBIDMIN', the minimum ID number of existing records is specified. If it is `FDBIDPREV', the number less by one than the minimum ID number of existing records is specified. If it is `FDBIDMAX', the maximum ID number of existing records is specified. If it is `FDBIDNEXT', the number greater by one than the maximum ID number of existing records is specified.
`*num*' specifies the additional value.
If successful, the return value is the summation value, else, it is Not-a-Number.
If the corresponding record exists, the value is treated as a real number and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tcfdbsync' is used in order to synchronize updated contents of a fixed-length database object with the file and the device.

bool tcfdbsync(TCFDB *_fdb_);

`*fdb*' specifies the fixed-length database object connected as a writer.
If successful, the return value is true, else, it is false.
This function is useful when another process connects to the same database file.

The function `tcfdboptimize' is used in order to optimize the file of a fixed-length database object.

bool tcfdboptimize(TCFDB *_fdb_, int32_t _width_, int64_t _limsiz_);

`*fdb*' specifies the fixed-length database object connected as a writer.
`*width*' specifies the width of the value of each record. If it is not more than 0, the current setting is not changed.
`*limsiz*' specifies the limit size of the database file. If it is not more than 0, the current setting is not changed.
If successful, the return value is true, else, it is false.

The function `tcfdbvanish' is used in order to remove all records of a fixed-length database object.

bool tcfdbvanish(TCFDB *_fdb_);

`_fdb_' specifies the fixed-length database object connected as a writer.
If successful, the return value is true, else, it is false.

The function `tcfdbcopy' is used in order to copy the database file of a fixed-length database object.

bool tcfdbcopy(TCFDB *_fdb_, const char *_path_);

`_fdb_' specifies the fixed-length database object.
`_path_' specifies the path of the destination file. If it begins with `@', the trailing substring is executed as a command line.
If successful, the return value is true, else, it is false. False is returned if the executed command returns non-zero code.
The database file is assured to be kept synchronized and not modified while the copying or executing operation is in progress. So, this function is useful to create a backup file of the database file.

The function `tcfdbtranbegin' is used in order to begin the transaction of a fixed-length database object.

bool tcfdbtranbegin(TCFDB *_fdb_);

`_fdb_' specifies the fixed-length database object connected as a writer.
If successful, the return value is true, else, it is false.
The database is locked by the thread while the transaction so that only one transaction can be activated with a database object at the same time. Thus, the serializable isolation level is assumed if every database operation is performed in the transaction. All updated regions are kept track of by write ahead logging while the transaction. If the database is closed during transaction, the transaction is aborted implicitly.

The function `tcfdbtrancommit' is used in order to commit the transaction of a fixed-length database object.

bool tcfdbtrancommit(TCFDB *_fdb_);

`_fdb_' specifies the fixed-length database object connected as a writer.
If successful, the return value is true, else, it is false.
Update in the transaction is fixed when it is committed successfully.

The function `tcfdbtranabort' is used in order to abort the transaction of a fixed-length database object.

bool tcfdbtranabort(TCFDB *_fdb_);

`_fdb_' specifies the fixed-length database object connected as a writer.
If successful, the return value is true, else, it is false.
Update in the transaction is discarded when it is aborted. The state of the database is rollbacked to before transaction.

The function `tcfdbpath' is used in order to get the file path of a fixed-length database object.

const char *tcfdbpath(TCFDB *_fdb_);

`_fdb_' specifies the fixed-length database object.
The return value is the path of the database file or `NULL' if the object does not connect to any database file.

The function `tcfdbrnum' is used in order to get the number of records of a fixed-length database object.

uint64_t tcfdbrnum(TCFDB *_fdb_);

`_fdb_' specifies the fixed-length database object.
The return value is the number of records or 0 if the object does not connect to any database file.

The function `tcfdbfsiz' is used in order to get the size of the database file of a fixed-length database object.

uint64_t tcfdbfsiz(TCFDB *_fdb_);

`_fdb_' specifies the fixed-length database object.
The return value is the size of the database file or 0 if the object does not connect to any database file.

Example Code

The following code is an example to use a hash database.

```
#include <tcutil.h>
#include <tcfdb.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>

int main(int argc, char **argv){
  TCFDB *fdb;
  int ecode;
  char *key, *value;

  /* create the object */
  fdb = tcfdbnew();

  /* open the database */
  if(!tcfdbopen(fdb, "casket.tcf", FDBOWRITER | FDBOCREAT)){
    ecode = tcfdbecode(fdb);
    fprintf(stderr, "open error: %s\n", tcfdberrmsg(ecode));
  }

  /* store records */
  if(!tcfdbput3(fdb, "1", "one") ||
     !tcfdbput3(fdb, "12", "twelve") ||
     !tcfdbput3(fdb, "144", "one forty four")){
    ecode = tcfdbecode(fdb);
    fprintf(stderr, "put error: %s\n", tcfdberrmsg(ecode));
  }

  /* retrieve records */
  value = tcfdbget3(fdb, "1");
  if(value){
    printf("%s\n", value);
    free(value);
  } else {
    ecode = tcfdbecode(fdb);
    fprintf(stderr, "get error: %s\n", tcfdberrmsg(ecode));
  }

  /* traverse records */
  tcfdbiterinit(fdb);
  while((key = tcfdbiternext3(fdb)) != NULL){
    value = tcfdbget3(fdb, key);
    if(value){
      printf("%s:%s\n", key, value);
      free(value);
    }
    free(key);
```

```
  }

  /* close the database */
  if(!tcfdbclose(fdb)){
    ecode = tcfdbecode(fdb);
    fprintf(stderr, "close error: %s\n", tcfdberrmsg(ecode));
  }

  /* delete the object */
  tcfdbdel(fdb);

  return 0;
}
```

CLI

To use the fixed-length database API easily, the commands `**tcftest**', `**tcfmttest**', and `**tcfmgr**' are provided.

The command `**tcftest**' is a utility for facility test and performance test. This command is used in the following format. `*path*' specifies the path of a database file. `*rnum*' specifies the number of iterations. `*width*' specifies the width of the value of each record. `*limsiz*' specifies the limit size of the database file.

tcftest write [-mt] [-nl|-nb] [-rnd] ***path rnum*** **[*width* [*limsiz*]]**
: Store records with keys of 8 bytes. They change as `00000001', `00000002'...

tcftest read [-mt] [-nl|-nb] [-wb] [-rnd] ***path***
: Retrieve all records of the database above.

tcftest remove [-mt] [-nl|-nb] [-rnd] ***path***
: Remove all records of the database above.

tcftest rcat [-mt] [-nl|-nb] [-pn ***num***] **[-dai|-dad|-rl]** ***path rnum*** **[*limsiz*]]**
: Store records with partway duplicated keys using concatenate mode.

tcftest misc [-mt] [-nl|-nb] ***path rnum***
: Perform miscellaneous test of various operations.

tcftest wicked [-mt] [-nl|-nb] ***path rnum***
: Perform updating operations selected at random.

Options feature the following.

-mt : call the function `tcfdbsetmutex'.
-nl : enable the option `FDBNOLCK'.
-nb : enable the option `FDBLCKNB'.
-rnd : select keys at random.
-wb : use the function `tcfdbget4' instead of `tcfdbget2'.
-pn ***num*** : specify the number of patterns.
-dai : use the function `tcfdbaddint' instead of `tcfdbputcat'.
-dad : use the function `tcfdbadddouble' instead of `tcfdbputcat'.
-rl : set the length of values at random.

This command returns 0 on success, another on failure.

The command `**tcfmttest**' is a utility for facility test under multi-thread situation. This command is used in

the following format. `*path*' specifies the path of a database file. `*tnum*' specifies the number of running threads. `*rnum*' specifies the number of iterations. `*width*' specifies the width of the value of each record. `*limsiz*' specifies the limit size of the database file.

tcfmttest write [-nl|-nb] [-rnd] *path tnum rnum* [*width* [*limsiz*]]
Store records with keys of 8 bytes. They change as `00000001', `00000002'...
tcfmttest read [-nl|-nb] [-wb] [-rnd] *path tnum*
Retrieve all records of the database above.
tcfmttest remove [-nl|-nb] [-rnd] *path tnum*
Remove all records of the database above.
tcfmttest wicked [-nl|-nb] [-nc] *path tnum rnum*
Perform updating operations selected at random.
tcfmttest typical [-nl|-nb] [-nc] [-rr *num*] *path tnum rnum* [*width* [*limsiz*]]
Perform typical operations selected at random.

Options feature the following.

-nl : enable the option `FDBNOLCK'.
-nb : enable the option `FDBLCKNB'.
-rnd : select keys at random.
-wb : use the function `tcfdbget4' instead of `tcfdbget2'.
-nc : omit the comparison test.
-rr *num* : specify the ratio of reading operation by percentage.

This command returns 0 on success, another on failure.

The command `**tcfmgr**' is a utility for test and debugging of the fixed-length database API and its applications. `*path*' specifies the path of a database file. `*width*' specifies the width of the value of each record. `*limsiz*' specifies the limit size of the database file. `*key*' specifies the key of a record. `*value*' specifies the value of a record. `*file*' specifies the input file.

tcfmgr create *path* [*width* [*limsiz*]]
Create a database file.
tcfmgr inform [-nl|-nb] *path*
Print miscellaneous information to the standard output.
tcfmgr put [-nl|-nb] [-sx] [-dk|-dc|-dai|-dad] *path key value*
Store a record.
tcfmgr out [-nl|-nb] [-sx] *path key*
Remove a record.
tcfmgr get [-nl|-nb] [-sx] [-px] [-pz] *path key*
Print the value of a record.
tcfmgr list [-nl|-nb] [-m *num*] [-pv] [-px] [-rb *lkey ukey*] [-ri *str*] *path*
Print keys of all records, separated by line feeds.
tcfmgr optimize [-nl|-nb] *path* [*width* [*limsiz*]]
Optimize a database file.
tcfmgr importtsv [-nl|-nb] [-sc] *path* [*file*]
Store records of TSV in each line of a file.

tcfmgr version

Print the version information of Tokyo Cabinet.

Options feature the following.

- **-nl** : enable the option `FDBNOLCK'.
- **-nb** : enable the option `FDBLCKNB'.
- **-sx** : the input data is evaluated as a hexadecimal data string.
- **-dk** : use the function `tcfdbputkeep' instead of `tcfdbput'.
- **-dc** : use the function `tcfdbputcat' instead of `tcfdbput'.
- **-dai** : use the function `tcfdbaddint' instead of `tcfdbput'.
- **-dad** : use the function `tcfdbadddouble' instead of `tcfdbput'.
- **-px** : the output data is converted into a hexadecimal data string.
- **-pz** : do not append line feed at the end of the output.
- **-m** ***num*** : specify the maximum number of the output.
- **-pv** : print values of records also.
- **-rb** ***lkey ukey*** : specify the range of keys.
- **-ri** ***str*** : specify the interval notation of keys.
- **-sc** : normalize keys as lower cases.

This command returns 0 on success, another on failure.

The Table Database API

Table database is a file containing records composed of the primary keys and arbitrary columns and is handled with the table database API. See `**tctdb.h**' for the entire specification.

Description

To use the table database API, include `**tcutil.h**', `**tctdb.h**', and related standard header files. Usually, write the following description near the front of a source file.

```
#include <tcutil.h>
#include <tctdb.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
```

Objects whose type is pointer to `**TCTDB**' are used to handle table databases. A table database object is created with the function `**tctdbnew**' and is deleted with the function `**tctdbdel**'. To avoid memory leak, it is important to delete every object when it is no longer in use.

Before operations to store or retrieve records, it is necessary to open a database file and connect the table database object to it. The function `**tctdbopen**' is used to open a database file and the function `**tctdbclose**' is used to close the database file. To avoid data missing or corruption, it is important to close every database file when it is no longer in use. It is forbidden for multible database objects in a process to open the same database at the same time.

API

The function `tctdberrmsg' is used in order to get the message string corresponding to an error code.

const char *tctdberrmsg(int *ecode*);

`*ecode*' specifies the error code.
The return value is the message string of the error code.

The function `tctdbnew' is used in order to create a table database object.

TCTDB *tctdbnew(void);

The return value is the new table database object.

The function `tctdbdel' is used in order to delete a table database object.

void tctdbdel(TCTDB **tdb*);

`*tdb*' specifies the table database object.
If the database is not closed, it is closed implicitly. Note that the deleted object and its derivatives can not be used anymore.

The function `tctdbecode' is used in order to get the last happened error code of a table database object.

int tctdbecode(TCTDB *_tdb_);

`_tdb_' specifies the table database object.
The return value is the last happened error code.
The following error codes are defined: `TCESUCCESS' for success, `TCETHREAD' for threading error, `TCEINVALID' for invalid operation, `TCENOFILE' for file not found, `TCENOPERM' for no permission, `TCEMETA' for invalid meta data, `TCERHEAD' for invalid record header, `TCEOPEN' for open error, `TCECLOSE' for close error, `TCETRUNC' for trunc error, `TCESYNC' for sync error, `TCESTAT' for stat error, `TCESEEK' for seek error, `TCEREAD' for read error, `TCEWRITE' for write error, `TCEMMAP' for mmap error, `TCELOCK' for lock error, `TCEUNLINK' for unlink error, `TCERENAME' for rename error, `TCEMKDIR' for mkdir error, `TCERMDIR' for rmdir error, `TCEKEEP' for existing record, `TCENOREC' for no record found, and `TCEMISC' for miscellaneous error.

The function `tctdbsetmutex' is used in order to set mutual exclusion control of a table database object for threading.

bool tctdbsetmutex(TCTDB *_tdb_);

`_tdb_' specifies the table database object which is not opened.
If successful, the return value is true, else, it is false.
Note that the mutual exclusion control is needed if the object is shared by plural threads and this function should be called before the database is opened.

The function `tctdbtune' is used in order to set the tuning parameters of a table database object.

bool tctdbtune(TCTDB *_tdb_, int64_t _bnum_, int8_t _apow_, int8_t _fpow_, uint8_t _opts_);

`_tdb_' specifies the table database object which is not opened.
`_bnum_' specifies the number of elements of the bucket array. If it is not more than 0, the default value is specified. The default value is 131071. Suggested size of the bucket array is about from 0.5 to 4 times of the number of all records to be stored.
`_apow_' specifies the size of record alignment by power of 2. If it is negative, the default value is specified. The default value is 4 standing for 2^4=16.
`_fpow_' specifies the maximum number of elements of the free block pool by power of 2. If it is negative, the default value is specified. The default value is 10 standing for 2^10=1024.
`_opts_' specifies options by bitwise-or: `TDBTLARGE' specifies that the size of the database can be larger than 2GB by using 64-bit bucket array, `TDBTDEFLATE' specifies that each record is compressed with Deflate encoding, `TDBTBZIP' specifies that each record is compressed with BZIP2 encoding, `TDBTTCBS' specifies that each record is compressed with TCBS encoding.
If successful, the return value is true, else, it is false.
Note that the tuning parameters should be set before the database is opened.

The function `tctdbsetcache' is set the caching parameters of a table database object.

bool tctdbsetcache(TCTDB *_tdb_, int32_t _rcnum_, int32_t _lcnum_, int32_t _ncnum_);

`_tdb_' specifies the table database object which is not opened.
`_rcnum_' specifies the maximum number of records to be cached. If it is not more than 0, the record cache is disabled. It is disabled by default.
`_lcnum_' specifies the maximum number of leaf nodes to be cached. If it is not more than 0, the default

value is specified. The default value is 4096.
`*ncnum*' specifies the maximum number of non-leaf nodes to be cached. If it is not more than 0, the default value is specified. The default value is 512.
If successful, the return value is true, else, it is false.
Note that the caching parameters should be set before the database is opened. Leaf nodes and non-leaf nodes are used in column indices.

The function `tctdbsetxmsiz' is used in order to set the size of the extra mapped memory of a table database object.

bool tctdbsetxmsiz(TCTDB ****tdb*, int64_t** ***xmsiz***);**

`*tdb*' specifies the table database object which is not opened.
`*xmsiz*' specifies the size of the extra mapped memory. If it is not more than 0, the extra mapped memory is disabled. The default size is 67108864.
If successful, the return value is true, else, it is false.
Note that the mapping parameters should be set before the database is opened.

The function `tctdbsetdfunit' is used in order to set the unit step number of auto defragmentation of a table database object.

bool tctdbsetdfunit(TCTDB ****tdb*, int32_t** ***dfunit***);**

`*tdb*' specifies the table database object which is not opened.
`*dfunit*' specifie the unit step number. If it is not more than 0, the auto defragmentation is disabled. It is disabled by default.
If successful, the return value is true, else, it is false.
Note that the defragmentation parameters should be set before the database is opened.

The function `tctdbopen' is used in order to open a database file and connect a table database object.

bool tctdbopen(TCTDB ****tdb*, const char ****path***, int** ***omode***);**

`*tdb*' specifies the table database object which is not opened.
`*path*' specifies the path of the database file.
`*omode*' specifies the connection mode: `TDBOWRITER' as a writer, `TDBOREADER' as a reader. If the mode is `TDBOWRITER', the following may be added by bitwise-or: `TDBOCREAT', which means it creates a new database if not exist, `TDBOTRUNC', which means it creates a new database regardless if one exists, `TDBOTSYNC', which means every transaction synchronizes updated contents with the device. Both of `TDBOREADER' and `TDBOWRITER' can be added to by bitwise-or: `TDBONOLCK', which means it opens the database file without file locking, or `TDBOLCKNB', which means locking is performed without blocking.
If successful, the return value is true, else, it is false.

The function `tctdbclose' is used in order to close a table database object.

bool tctdbclose(TCTDB ****tdb*);**

`*tdb*' specifies the table database object.
If successful, the return value is true, else, it is false.
Update of a database is assured to be written when the database is closed. If a writer opens a database but does not close it appropriately, the database will be broken.

The function `tctdbput' is used in order to store a record into a table database object.

```
bool tctdbput(TCTDB *tdb, const void *pkbuf, int pksiz, TCMAP *cols);
```

`*tdb*' specifies the table database object connected as a writer.
`*pkbuf*' specifies the pointer to the region of the primary key.
`*pksiz*' specifies the size of the region of the primary key.
`*cols*' specifies a map object containing columns.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tctdbput2' is used in order to store a string record into a table database object with a zero separated column string.

```
bool tctdbput2(TCTDB *tdb, const void *pkbuf, int pksiz, const void *cbuf, int csiz);
```

`*tdb*' specifies the table database object connected as a writer.
`*pkbuf*' specifies the pointer to the region of the primary key.
`*pksiz*' specifies the size of the region of the primary key.
`*cbuf*' specifies the pointer to the region of the zero separated column string where the name and the value of each column are situated one after the other.
`*csiz*' specifies the size of the region of the column string.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tctdbput3' is used in order to store a string record into a table database object with a tab separated column string.

```
bool tctdbput3(TCTDB *tdb, const char *pkstr, const char *cstr);
```

`*tdb*' specifies the table database object connected as a writer.
`*pkstr*' specifies the string of the primary key.
`*cstr*' specifies the string of the the tab separated column string where the name and the value of each column are situated one after the other.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tctdbputkeep' is used in order to store a new record into a table database object.

```
bool tctdbputkeep(TCTDB *tdb, const void *pkbuf, int pksiz, TCMAP *cols);
```

`*tdb*' specifies the table database object connected as a writer.
`*pkbuf*' specifies the pointer to the region of the primary key.
`*pksiz*' specifies the size of the region of the primary key.
`*cols*' specifies a map object containing columns.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tctdbputkeep2' is used in order to store a new string record into a table database object with a zero separated column string.

```
bool tctdbputkeep2(TCTDB *tdb, const void *pkbuf, int pksiz, const void *cbuf, int
```

***csiz*);**

`*tdb*' specifies the table database object connected as a writer.
`*pkbuf*' specifies the pointer to the region of the primary key.
`*pksiz*' specifies the size of the region of the primary key.
`*cbuf*' specifies the pointer to the region of the zero separated column string where the name and the value of each column are situated one after the other.
`*csiz*' specifies the size of the region of the column string.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tctdbputkeep3' is used in order to store a new string record into a table database object with a tab separated column string.

bool tctdbputkeep3(TCTDB *tdb, const char *pkstr, const char *cstr);

`*tdb*' specifies the table database object connected as a writer.
`*pkstr*' specifies the string of the primary key.
`*cstr*' specifies the string of the the tab separated column string where the name and the value of each column are situated one after the other.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tctdbputcat' is used in order to concatenate columns of the existing record in a table database object.

bool tctdbputcat(TCTDB *tdb, const void *pkbuf, int pksiz, TCMAP *cols);

`*tdb*' specifies the table database object connected as a writer.
`*pkbuf*' specifies the pointer to the region of the primary key.
`*pksiz*' specifies the size of the region of the primary key.
`*cols*' specifies a map object containing columns.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tctdbputcat2' is used in order to concatenate columns in a table database object with a zero separated column string.

bool tctdbputcat2(TCTDB *tdb, const void *pkbuf, int pksiz, const void *cbuf, int csiz);

`*tdb*' specifies the table database object connected as a writer.
`*pkbuf*' specifies the pointer to the region of the primary key.
`*pksiz*' specifies the size of the region of the primary key.
`*cbuf*' specifies the pointer to the region of the zero separated column string where the name and the value of each column are situated one after the other.
`*csiz*' specifies the size of the region of the column string.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tctdbputcat3' is used in order to concatenate columns in a table database object with with a tab separated column string.

bool tctdbputcat3(TCTDB *_tdb_, const char *_pkstr_, const char *_cstr_);

`_tdb_' specifies the table database object connected as a writer.
`_pkstr_' specifies the string of the primary key.
`_cstr_' specifies the string of the the tab separated column string where the name and the value of each column are situated one after the other.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tctdbout' is used in order to remove a record of a table database object.

bool tctdbout(TCTDB *_tdb_, const void *_pkbuf_, int _pksiz_);

`_tdb_' specifies the table database object connected as a writer.
`_pkbuf_' specifies the pointer to the region of the primary key.
`_pksiz_' specifies the size of the region of the primary key.
If successful, the return value is true, else, it is false.

The function `tctdbout2' is used in order to remove a string record of a table database object.

bool tctdbout2(TCTDB *_tdb_, const char *_pkstr_);

`_tdb_' specifies the table database object connected as a writer.
`_pkstr_' specifies the string of the primary key.
If successful, the return value is true, else, it is false.

The function `tctdbget' is used in order to retrieve a record in a table database object.

TCMAP *tctdbget(TCTDB *_tdb_, const void *_pkbuf_, int _pksiz_);

`_tdb_' specifies the table database object.
`_pkbuf_' specifies the pointer to the region of the primary key.
`_pksiz_' specifies the size of the region of the primary key.
If successful, the return value is a map object of the columns of the corresponding record. `NULL' is returned if no record corresponds.
Because the object of the return value is created with the function `tcmapnew', it should be deleted with the function `tcmapdel' when it is no longer in use.

The function `tctdbget2' is used in order to retrieve a record in a table database object as a zero separated column string.

char *tctdbget2(TCTDB *_tdb_, const void *_pkbuf_, int _pksiz_, int *_sp_);

`_tdb_' specifies the table database object.
`_pkbuf_' specifies the pointer to the region of the primary key.
`_pksiz_' specifies the size of the region of the primary key.
`_sp_' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the column string of the corresponding record. `NULL' is returned if no record corresponds.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tctdbget3' is used in order to retrieve a string record in a table database object as a tab

separated column string.

char *tctdbget3(TCTDB *_tdb_, const char *_pkstr_);

`_tdb_' specifies the table database object.
`_pkstr_' specifies the string of the primary key.
If successful, the return value is the tab separated column string of the corresponding record. `NULL' is returned if no record corresponds.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tctdbvsiz' is used in order to get the size of the value of a record in a table database object.

int tctdbvsiz(TCTDB *_tdb_, const void *_pkbuf_, int _pksiz_);

`_tdb_' specifies the table database object.
`_kbuf_' specifies the pointer to the region of the primary key.
`_ksiz_' specifies the size of the region of the primary key.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tctdbvsiz2' is used in order to get the size of the value of a string record in a table database object.

int tctdbvsiz2(TCTDB *_tdb_, const char *_pkstr_);

`_tdb_' specifies the table database object.
`_kstr_' specifies the string of the primary key.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tctdbiterinit' is used in order to initialize the iterator of a table database object.

bool tctdbiterinit(TCTDB *_tdb_);

`_tdb_' specifies the table database object.
If successful, the return value is true, else, it is false.
The iterator is used in order to access the primary key of every record stored in a database.

The function `tctdbiternext' is used in order to get the next primary key of the iterator of a table database object.

void *tctdbiternext(TCTDB *_tdb_, int *_sp_);

`_tdb_' specifies the table database object.
`_sp_' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the next primary key, else, it is `NULL'. `NULL' is returned when no record is to be get out of the iterator.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. It is possible to access every record by iteration of calling this function. It is allowed to update or remove records whose keys are fetched while the iteration. However, it is not assured if updating the database is occurred while the iteration. Besides, the order of this traversal access method is arbitrary, so it is not assured that the order of storing matches the one of the traversal access.

The function `tctdbiternext2' is used in order to get the next primary key string of the iterator of a table database object.

`char *tctdbiternext2(TCTDB *tdb);`

`*tdb*' specifies the table database object.

If successful, the return value is the string of the next primary key, else, it is `NULL'. `NULL' is returned when no record is to be get out of the iterator.

Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. It is possible to access every record by iteration of calling this function. However, it is not assured if updating the database is occurred while the iteration. Besides, the order of this traversal access method is arbitrary, so it is not assured that the order of storing matches the one of the traversal access.

The function `tctdbiternext3' is used in order to get the columns of the next record of the iterator of a table database object.

`TCMAP *tctdbiternext3(TCTDB *tdb);`

`*tdb*' specifies the table database object.

If successful, the return value is a map object of the columns of the next record, else, it is `NULL'. `NULL' is returned when no record is to be get out of the iterator. The primary key is added into the map as a column of an empty string key.

Because the object of the return value is created with the function `tcmapnew', it should be deleted with the function `tcmapdel' when it is no longer in use. It is possible to access every record by iteration of calling this function. However, it is not assured if updating the database is occurred while the iteration. Besides, the order of this traversal access method is arbitrary, so it is not assured that the order of storing matches the one of the traversal access.

The function `tctdbfwmkeys' is used in order to get forward matching primary keys in a table database object.

`TCLIST *tctdbfwmkeys(TCTDB *tdb, const void *pbuf, int psiz, int max);`

`*tdb*' specifies the table database object.

`*pbuf*' specifies the pointer to the region of the prefix.

`*psiz*' specifies the size of the region of the prefix.

`*max*' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.

The return value is a list object of the corresponding keys. This function does never fail. It returns an empty list even if no key corresponds.

Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use. Note that this function may be very slow because every key in the database is scanned.

The function `tctdbfwmkeys2' is used in order to get forward matching string primary keys in a table database object.

`TCLIST *tctdbfwmkeys2(TCTDB *tdb, const char *pstr, int max);`

`*tdb*' specifies the table database object.

`*pstr*' specifies the string of the prefix.

`*max*' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.

The return value is a list object of the corresponding keys. This function does never fail. It returns an

empty list even if no key corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use. Note that this function may be very slow because every key in the database is scanned.

The function `tctdbaddint' is used in order to add an integer to a column of a record in a table database object.

```
int tctdbaddint(TCTDB *tdb, const void *pkbuf, int pksiz, int num);
```

`*tdb*' specifies the table database object connected as a writer.
`*kbuf*' specifies the pointer to the region of the primary key.
`*ksiz*' specifies the size of the region of the primary key.
`*num*' specifies the additional value.
If successful, the return value is the summation value, else, it is `INT_MIN'.
The additional value is stored as a decimal string value of a column whose name is "_num". If no record corresponds, a new record with the additional value is stored.

The function `tctdbadddouble' is used in order to add a real number to a column of a record in a table database object.

```
double tctdbadddouble(TCTDB *tdb, const void *pkbuf, int pksiz, double num);
```

`*tdb*' specifies the table database object connected as a writer.
`*kbuf*' specifies the pointer to the region of the primary key.
`*ksiz*' specifies the size of the region of the primary key.
`*num*' specifies the additional value.
If successful, the return value is the summation value, else, it is Not-a-Number.
The additional value is stored as a decimal string value of a column whose name is "_num". If no record corresponds, a new record with the additional value is stored.

The function `tctdbsync' is used in order to synchronize updated contents of a table database object with the file and the device.

```
bool tctdbsync(TCTDB *tdb);
```

`*tdb*' specifies the table database object connected as a writer.
If successful, the return value is true, else, it is false.
This function is useful when another process connects to the same database file.

The function `tctdboptimize' is used in order to optimize the file of a table database object.

```
bool tctdboptimize(TCTDB *tdb, int64_t bnum, int8_t apow, int8_t fpow, uint8_t opts);
```

`*tdb*' specifies the table database object connected as a writer.
`*bnum*' specifies the number of elements of the bucket array. If it is not more than 0, the default value is specified. The default value is two times of the number of records.
`*apow*' specifies the size of record alignment by power of 2. If it is negative, the current setting is not changed.
`*fpow*' specifies the maximum number of elements of the free block pool by power of 2. If it is negative, the current setting is not changed.
`*opts*' specifies options by bitwise-or: `TDBTLARGE' specifies that the size of the database can be

larger than 2GB by using 64-bit bucket array, `TDBTDEFLATE' specifies that each record is compressed with Deflate encoding, `TDBTBZIP' specifies that each record is compressed with BZIP2 encoding, `TDBTTCBS' specifies that each record is compressed with TCBS encoding. If it is `UINT8_MAX', the current setting is not changed.
If successful, the return value is true, else, it is false.
This function is useful to reduce the size of the database file with data fragmentation by successive updating.

The function `tctdbvanish' is used in order to remove all records of a table database object.

```
bool tctdbvanish(TCTDB *tdb);
```

`*tdb*' specifies the table database object connected as a writer.
If successful, the return value is true, else, it is false.

The function `tctdbcopy' is used in order to copy the database file of a table database object.

```
bool tctdbcopy(TCTDB *tdb, const char *path);
```

`*tdb*' specifies the table database object.
`*path*' specifies the path of the destination file. If it begins with `@', the trailing substring is executed as a command line.
If successful, the return value is true, else, it is false. False is returned if the executed command returns non-zero code.
The database file is assured to be kept synchronized and not modified while the copying or executing operation is in progress. So, this function is useful to create a backup file of the database file.

The function `tctdbtranbegin' is used in order to begin the transaction of a table database object.

```
bool tctdbtranbegin(TCTDB *tdb);
```

`*tdb*' specifies the table database object connected as a writer.
If successful, the return value is true, else, it is false.
The database is locked by the thread while the transaction so that only one transaction can be activated with a database object at the same time. Thus, the serializable isolation level is assumed if every database operation is performed in the transaction. Because all pages are cached on memory while the transaction, the amount of referred records is limited by the memory capacity. If the database is closed during transaction, the transaction is aborted implicitly.

The function `tctdbtrancommit' is used in order to commit the transaction of a table database object.

```
bool tctdbtrancommit(TCTDB *tdb);
```

`*tdb*' specifies the table database object connected as a writer.
If successful, the return value is true, else, it is false.
Update in the transaction is fixed when it is committed successfully.

The function `tctdbtranabort' is used in order to abort the transaction of a table database object.

```
bool tctdbtranabort(TCTDB *tdb);
```

`*tdb*' specifies the table database object connected as a writer.
If successful, the return value is true, else, it is false.
Update in the transaction is discarded when it is aborted. The state of the database is rollbacked to before transaction.

The function `tctdbpath' is used in order to get the file path of a table database object.

```
const char *tctdbpath(TCTDB *tdb);
```

`*tdb*' specifies the table database object.
The return value is the path of the database file or `NULL' if the object does not connect to any database file.

The function `tctdbrnum' is used in order to get the number of records ccccof a table database object.

```
uint64_t tctdbrnum(TCTDB *tdb);
```

`*tdb*' specifies the table database object.
The return value is the number of records or 0 if the object does not connect to any database file.

The function `tctdbfsiz' is used in order to get the size of the database file of a table database object.

```
uint64_t tctdbfsiz(TCTDB *tdb);
```

`*tdb*' specifies the table database object.
The return value is the size of the database file or 0 if the object does not connect to any database file.

The function `tctdbsetindex' is used in order to set a column index to a table database object.

```
bool tctdbsetindex(TCTDB *tdb, const char *name, int type);
```

`*tdb*' specifies the table database object connected as a writer.
`*name*' specifies the name of a column. If the name of an existing index is specified, the index is rebuilt. An empty string means the primary key.
`*type*' specifies the index type: `TDBITLEXICAL' for lexical string, `TDBITDECIMAL' for decimal string, `TDBITTOKEN' for token inverted index, `TDBITQGRAM' for q-gram inverted index. If it is `TDBITOPT', the index is optimized. If it is `TDBITVOID', the index is removed. If `TDBITKEEP' is added by bitwise-or and the index exists, this function merely returns failure.
If successful, the return value is true, else, it is false.
Note that the setting indices should be set after the database is opened.

The function `tctdbgenuid' is used in order to generate a unique ID number of a table database object.

```
int64_t tctdbgenuid(TCTDB *tdb);
```

`*tdb*' specifies the table database object connected as a writer.
The return value is the new unique ID number or -1 on failure.

The function `tctdbqrynew' is used in order to create a query object.

```
TDBQRY *tctdbqrynew(TCTDB *tdb);
```

`*tdb*' specifies the table database object.
The return value is the new query object.

The function `tctdbqrydel' is used in order to delete a query object.

```
void tctdbqrydel(TDBQRY *qry);
```

`*qry*' specifies the query object.

The function `tctdbqryaddcond' is used in order to add a narrowing condition to a query object.

void tctdbqryaddcond(TDBQRY *_qry_, const char *_name_, int _op_, const char *_expr_);

`_qry_' specifies the query object.
`_name_' specifies the name of a column. An empty string means the primary key.
`_op_' specifies an operation type: `TDBQCSTREQ' for string which is equal to the expression, `TDBQCSTRINC' for string which is included in the expression, `TDBQCSTRBW' for string which begins with the expression, `TDBQCSTREW' for string which ends with the expression, `TDBQCSTRAND' for string which includes all tokens in the expression, `TDBQCSTROR' for string which includes at least one token in the expression, `TDBQCSTROREQ' for string which is equal to at least one token in the expression, `TDBQCSTRRX' for string which matches regular expressions of the expression, `TDBQCNUMEQ' for number which is equal to the expression, `TDBQCNUMGT' for number which is greater than the expression, `TDBQCNUMGE' for number which is greater than or equal to the expression, `TDBQCNUMLT' for number which is less than the expression, `TDBQCNUMLE' for number which is less than or equal to the expression, `TDBQCNUMBT' for number which is between two tokens of the expression, `TDBQCNUMOREQ' for number which is equal to at least one token in the expression, `TDBQCFTSPH' for full-text search with the phrase of the expression, `TDBQCFTSAND' for full-text search with all tokens in the expression, `TDBQCFTSOR' for full-text search with at least one token in the expression, `TDBQCFTSEX' for full-text search with the compound expression. All operations can be flagged by bitwise-or: `TDBQCNEGATE' for negation, `TDBQCNOIDX' for using no index.
`_expr_' specifies an operand exression.

The function `tctdbqrysetorder' is used in order to set the order of a query object.

void tctdbqrysetorder(TDBQRY *_qry_, const char *_name_, int _type_);

`_qry_' specifies the query object.
`_name_' specifies the name of a column. An empty string means the primary key.
`_type_' specifies the order type: `TDBQOSTRASC' for string ascending, `TDBQOSTRDESC' for string descending, `TDBQONUMASC' for number ascending, `TDBQONUMDESC' for number descending.

The function `tctdbqrysetlimit' is used in order to set the limit number of records of the result of a query object.

void tctdbqrysetlimit(TDBQRY *_qry_, int _max_, int _skip_);

`_qry_' specifies the query object.
`_max_' specifies the maximum number of records of the result. If it is negative, no limit is specified.
`_skip_' specifies the number of skipped records of the result. If it is not more than 0, no record is skipped.

The function `tctdbqrysearch' is used in order to execute the search of a query object.

TCLIST *tctdbqrysearch(TDBQRY *_qry_);

`_qry_' specifies the query object.
The return value is a list object of the primary keys of the corresponding records. This function does never fail. It returns an empty list even if no record corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tctdbqrysearchout' is used in order to remove each record corresponding to a query object.

bool tctdbqrysearchout(TDBQRY *qry);

`qry' specifies the query object of the database connected as a writer.
If successful, the return value is true, else, it is false.

The function `tctdbqryproc' is used in order to process each record corresponding to a query object.

bool tctdbqryproc(TDBQRY *qry, TDBQRYPROC proc, void *op);

`qry' specifies the query object of the database connected as a writer.
`proc' specifies the pointer to the iterator function called for each record. It receives four parameters. The first parameter is the pointer to the region of the primary key. The second parameter is the size of the region of the primary key. The third parameter is a map object containing columns. The fourth parameter is the pointer to the optional opaque object. It returns flags of the post treatment by bitwise-or: `TDBQPPUT' to modify the record, `TDBQPOUT' to remove the record, `TDBQPSTOP' to stop the iteration.
`op' specifies an arbitrary pointer to be given as a parameter of the iterator function. If it is not needed, `NULL' can be specified.
If successful, the return value is true, else, it is false.

The function `tctdbqryhint' is used in order to get the hint string of a query object.

const char *tctdbqryhint(TDBQRY *qry);

`qry' specifies the query object.
The return value is the hint string.

The function `tctdbmetasearch' is used in order to retrieve records with multiple query objects and get the set of the result.

TCLIST *tctdbmetasearch(TDBQRY **qrys, int num, int type);

`qrys' specifies an array of the query objects.
`num' specifies the number of elements of the array.
`type' specifies a set operation type: `TDBMSUNION' for the union set, `TDBMSISECT' for the intersection set, `TDBMSDIFF' for the difference set.
The return value is a list object of the primary keys of the corresponding records. This function does never fail. It returns an empty list even if no record corresponds.
If the first query object has the order setting, the result array is sorted by the order. Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

Example Code

The following code is an example to use a table database.

```
#include <tcutil.h>
#include <tctdb.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>

int main(int argc, char **argv){
  TCTDB *tdb;
  int ecode, pksiz, i, rsiz;
  char pkbuf[256];
```

```
const char *rbuf, *name;
TCMAP *cols;
TDBQRY *qry;
TCLIST *res;

/* create the object */
tdb = tctdbnew();

/* open the database */
if(!tctdbopen(tdb, "casket.tct", TDBOWRITER | TDBOCREAT)){
  ecode = tctdbecode(tdb);
  fprintf(stderr, "open error: %s\n", tctdberrmsg(ecode));
}

/* store a record */
pksiz = sprintf(pkbuf, "%ld", (long)tctdbgenuid(tdb));
cols = tcmapnew3("name", "mikio", "age", "30", "lang", "ja,en,c", NULL);
if(!tctdbput(tdb, pkbuf, pksiz, cols)){
  ecode = tctdbecode(tdb);
  fprintf(stderr, "put error: %s\n", tctdberrmsg(ecode));
}
tcmapdel(cols);

/* store a record in a naive way */
pksiz = sprintf(pkbuf, "12345");
cols = tcmapnew();
tcmapput2(cols, "name", "falcon");
tcmapput2(cols, "age", "31");
tcmapput2(cols, "lang", "ja");
if(!tctdbput(tdb, pkbuf, pksiz, cols)){
  ecode = tctdbecode(tdb);
  fprintf(stderr, "put error: %s\n", tctdberrmsg(ecode));
}
tcmapdel(cols);

/* store a record with a TSV string */
if(!tctdbput3(tdb, "abcde", "name\tjoker\tage\t19\tlang\ten,es")){
  ecode = tctdbecode(tdb);
  fprintf(stderr, "put error: %s\n", tctdberrmsg(ecode));
}

/* search for records */
qry = tctdbqrynew(tdb);
tctdbqryaddcond(qry, "age", TDBQCNUMGE, "20");
tctdbqryaddcond(qry, "lang", TDBQCSTROR, "ja,en");
tctdbqrysetorder(qry, "name", TDBQOSTRASC);
tctdbqrysetlimit(qry, 10, 0);
res = tctdbqrysearch(qry);
for(i = 0; i < tclistnum(res); i++){
  rbuf = tclistval(res, i, &rsiz);
  cols = tctdbget(tdb, rbuf, rsiz);
  if(cols){
    printf("%s", rbuf);
    tcmapiterinit(cols);
    while((name = tcmapiternext2(cols)) != NULL){
      printf("\t%s\t%s", name, tcmapget2(cols, name));
    }
    printf("\n");
    tcmapdel(cols);
  }
}
tclistdel(res);
tctdbqrydel(qry);

/* close the database */
```

```
  if(!tctdbclose(tdb)){
    ecode = tctdbecode(tdb);
    fprintf(stderr, "close error: %s\n", tctdberrmsg(ecode));
  }

  /* delete the object */
  tctdbdel(tdb);

  return 0;
}
```

CLI

To use the table database API easily, the commands `**tcttest**', `**tctmttest**', and `**tctmgr**' are provided.

The command `**tcttest**' is a utility for facility test and performance test. This command is used in the following format. `*path*' specifies the path of a database file. `*rnum*' specifies the number of iterations. `*bnum*' specifies the number of buckets. `*apow*' specifies the power of the alignment. `*fpow*' specifies the power of the free block pool.

tcttest write [-mt] [-tl] [-td|-tb|-tt|-tx] [-rc *num*] [-lc *num*] [-nc *num*] [-xm *num*] [-df *num*] [-ip] [-is] [-in] [-it] [-if] [-ix] [-nl|-nb] [-rnd] *path* *rnum* [*bnum* [*apow* [*fpow*]]]
: Store records with columns "str", "num", "type", and "flag".

tcttest read [-mt] [-rc *num*] [-lc *num*] [-nc *num*] [-xm *num*] [-df *num*] [-nl|-nb] [-rnd] *path*
: Retrieve all records of the database above.

tcttest remove [-mt] [-rc *num*] [-lc *num*] [-nc *num*] [-xm *num*] [-df *num*] [-nl|-nb] [-rnd] *path*
: Remove all records of the database above.

tcttest rcat [-mt] [-tl] [-td|-tb|-tt|-tx] [-rc *num*] [-lc *num*] [-nc *num*] [-xm *num*] [-df *num*] [-ip] [-is] [-in] [-it] [-if] [-ix] [-nl|-nb] [-pn *num*] [-dai|-dad|-rl|-ru] *path* *rnum* [*bnum* [*apow* [*fpow*]]]
: Store records with partway duplicated keys using concatenate mode.

tcttest misc [-mt] [-tl] [-td|-tb|-tt|-tx] [-nl|-nb] *path* *rnum*
: Perform miscellaneous test of various operations.

tcttest wicked [-mt] [-tl] [-td|-tb|-tt|-tx] [-nl|-nb] *path* *rnum*
: Perform updating operations selected at random.

Options feature the following.

- **-mt** : call the function `tctdbsetmutex'.
- **-tl** : enable the option `TDBTLARGE'.
- **-td** : enable the option `TDBTDEFLATE'.
- **-tb** : enable the option `TDBTBZIP'.
- **-tt** : enable the option `TDBTTCBS'.
- **-tx** : enable the option `TDBTEXCODEC'.
- **-rc *num*** : specify the number of cached records.
- **-lc *num*** : specify the number of cached leaf pages.

-nc *num* : specify the number of cached non-leaf pages.
-xm *num* : specify the size of the extra mapped memory.
-df *num* : specify the unit step number of auto defragmentation.
-ip : create the number index for the primary key.
-is : create the string index for the column "str".
-in : create the number index for the column "num".
-it : create the string index for the column "type".
-if : create the token inverted index for the column "flag".
-ix : create the q-gram inverted index for the column "text".
-nl : enable the option `TDBNOLCK'.
-nb : enable the option `TDBLCKNB'.
-rnd : select keys at random.
-pn *num* : specify the number of patterns.
-dai : use the function `tctdbaddint' instead of `tctdbputcat'.
-dad : use the function `tctdbadddouble' instead of `tctdbputcat'.
-rl : set the length of values at random.
-ru : select update operations at random.

This command returns 0 on success, another on failure.

The command `**tctmttest**' is a utility for facility test under multi-thread situation. This command is used in the following format. `*path*' specifies the path of a database file. `*tnum*' specifies the number of running threads. `*rnum*' specifies the number of iterations. `*bnum*' specifies the number of buckets. `*apow*' specifies the power of the alignment. `*fpow*' specifies the power of the free block pool.

tctmttest write [-tl] [-td|-tb|-tt|-tx] [-rc *num*] [-lc *num*] [-nc *num*] [-xm *num*] [-df *num*] [-ip] [-is] [-in] [-it] [-if] [-ix] [-nl|-nb] [-rnd] *path tnum rnum* [*bnum* [*apow* [*fpow*]]]
Store records with columns "str", "num", "type", and "flag".
tctmttest read [-rc *num*] [-lc *num*] [-nc *num*] [-xm *num*] [-df *num*] [-nl|-nb] [-rnd] *path tnum*
Retrieve all records of the database above.
tctmttest remove [-rc *num*] [-lc *num*] [-nc *num*] [-xm *num*] [-df *num*] [-nl|-nb] [-rnd] *path tnum*
Remove all records of the database above.
tctmttest wicked [-tl] [-td|-tb|-tt|-tx] [-nl|-nb] *path tnum rnum*
Perform updating operations selected at random.
tctmttest typical [-tl] [-td|-tb|-tt|-tx] [-rc *num*] [-lc *num*] [-nc *num*] [-xm *num*] [-df *num*] [-nl|-nb] [-rr *num*] *path tnum rnum* [*bnum* [*apow* [*fpow*]]]
Perform typical operations selected at random.

Options feature the following.

-tl : enable the option `TDBTLARGE'.
-td : enable the option `TDBTDEFLATE'.
-tb : enable the option `TDBTBZIP'.

-tt : enable the option `TDBTTCBS'.
-tx : enable the option `TDBTEXCODEC'.
-rc ***num*** : specify the number of cached records.
-lc ***num*** : specify the number of cached leaf pages.
-nc ***num*** : specify the number of cached non-leaf pages.
-xm ***num*** : specify the size of the extra mapped memory.
-df ***num*** : specify the unit step number of auto defragmentation.
-ip : create the number index for the primary key.
-is : create the string index for the column "str".
-in : create the number index for the column "num".
-it : create the string index for the column "type".
-if : create the token inverted index for the column "flag".
-ix : create the q-gram inverted index for the column "text".
-nl : enable the option `TDBNOLCK'.
-nb : enable the option `TDBLCKNB'.
-rnd : select keys at random.
-nc : omit the comparison test.
-rr ***num*** : specify the ratio of reading operation by percentage.

This command returns 0 on success, another on failure.

The command `**tctmgr**' is a utility for test and debugging of the table database API and its applications. `*path*' specifies the path of a database file. `*bnum*' specifies the number of buckets. `*apow*' specifies the power of the alignment. `*fpow*' specifies the power of the free block pool. `*pkey*' specifies the primary key of a record. `*cols*' specifies the names and the values of a record alternately. `*name*' specifies the name of a column. `*op*' specifies an operator. `*expr*' specifies the condition expression. `*file*' specifies the input file.

tctmgr create [-tl] [-td|-tb|-tt|-tx] ***path*** **[***bnum*** [***apow*** [***fpow***]]]**
Create a database file.
tctmgr inform [-nl|-nb] ***path***
Print miscellaneous information to the standard output.
tctmgr put [-nl|-nb] [-sx] [-dk|-dc|-dai|-dad] ***path pkey*** **[***cols*** ...]**
Store a record.
tctmgr out [-nl|-nb] [-sx] ***path pkey***
Remove a record.
tctmgr get [-nl|-nb] [-sx] [-px] [-pz] ***path pkey***
Print the value of a record.
tctmgr list [-nl|-nb] [-m ***num***] **[-pv] [-px] [-fm** ***str***] ***path***
Print the primary keys of all records, separated by line feeds.
tctmgr search [-nl|-nb] [-ord ***name type***] **[-m** ***num***] **[-sk** ***num***] **[-kw] [-pv] [-px] [-ph] [-bt** ***num***] **[-rm] [-ms** ***type***] ***path*** **[***name op expr*** ...]**
Print records matching conditions, separated by line feeds.
tctmgr optimize [-tl] [-td|-tb|-tt|-tx] [-tz] [-nl|-nb] [-df] ***path*** **[***bnum*** [***apow*** [***fpow***]]]**
Optimize a database file.
tctmgr setindex [-nl|-nb] [-it ***type***] ***path name***

Set the index of a column.

tctmgr importtsv [-nl|-nb] [-sc] ***path*** **[*file*]**

Store records of TSV in each line of a file.

tctmgr version

Print the version information of Tokyo Cabinet.

Options feature the following.

- **-tl** : enable the option `TDBTLARGE'.
- **-td** : enable the option `TDBTDEFLATE'.
- **-tb** : enable the option `TDBTBZIP'.
- **-tt** : enable the option `TDBTTCBS'.
- **-tx** : enable the option `TDBTEXCODEC'.
- **-nl** : enable the option `TDBNOLCK'.
- **-nb** : enable the option `TDBLCKNB'.
- **-sx** : the input data is evaluated as a hexadecimal data string.
- **-dk** : use the function `tctdbputkeep' instead of `tctdbput'.
- **-dc** : use the function `tctdbputcat' instead of `tctdbput'.
- **-dai** : use the function `tctdbaddint' instead of `tctdbput'.
- **-dad** : use the function `tctdbadddouble' instead of `tctdbput'.
- **-px** : the output data is converted into a hexadecimal data string.
- **-pz** : do not append line feed at the end of the output.
- **-m** ***num*** : specify the maximum number of the output.
- **-pv** : print values of records also.
- **-fm** ***str*** : specify the prefix of keys.
- **-ord** ***name type*** : specify the order of the result.
- **-sk** ***num*** : specify the number of skipped records.
- **-kw** : print KWIC string.
- **-ph** : print hint information also.
- **-bt** : specify the number of benchmark tests.
- **-rm** : remove every record in the result.
- **-ms** ***type*** : specify the set operation of meta search.
- **-tz** : enable the option `UINT8_MAX'.
- **-df** : perform defragmentation only.
- **-it** ***type*** : specify the index type among "lexical", "decimal", "token", "qgram", and "void".
- **-cd** : create the number index instead of the string index.
- **-cv** : remove the existing index.
- **-sc** : normalize keys as lower cases.

The operator of the `search' subcommand is one of "STREQ", "STRINC", "STRBW", "STREW", "STRAND", "STROR", "STROREQ", "STRRX", "NUMEQ", "NUMGT", "NUMGE", "NUMLT", "NUMLE", "NUMBT", "NUMOREQ", "FTSPH", "FTSAND", "FTSOR", and "FTSEX". If "~" preposes each operator, the logical meaning is reversed. If "+" preposes each operator, no index is used for the operator. The type of the `-ord' option is one of "STRASC", "STRDESC", "NUMASC", and "NUMDESC". The type of the `-ms' option is one of "UNION", "ISECT", and "DIFF". This command returns 0 on success, another on failure.

The Abstract Database API

Abstract database is a set of interfaces to use on-memory hash database, on-memory tree database, hash database, B+ tree database, fixed-length database, and table database with the same API. See `**tcadb.h**' for the entire specification.

Description

To use the abstract database API, include `**tcutil.h**', `**tcadb.h**', and related standard header files. Usually, write the following description near the front of a source file.

```
#include <tcutil.h>
#include <tcadb.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
```

Objects whose type is pointer to `**TCADB**' are used to handle abstract databases. An abstract database object is created with the function `**tcadbnew**' and is deleted with the function `**tcadbdel**'. To avoid memory leak, it is important to delete every object when it is no longer in use.

Before operations to store or retrieve records, it is necessary to connect the abstract database object to the concrete one. The function `**tcadbopen**' is used to open a concrete database and the function `**tcadbclose**' is used to close the database. To avoid data missing or corruption, it is important to close every database instance when it is no longer in use. It is forbidden for multible database objects in a process to open the same database at the same time.

API

The function `tcadbnew' is used in order to create an abstract database object.

TCADB *tcadbnew(void);

The return value is the new abstract database object.

The function `tcadbdel' is used in order to delete an abstract database object.

void tcadbdel(TCADB *_adb_);

`*adb*' specifies the abstract database object.

The function `tcadbopen' is used in order to open an abstract database.

bool tcadbopen(TCADB *_adb_, const char *_name_);

`*adb*' specifies the abstract database object.
`*name*' specifies the name of the database. If it is "*", the database will be an on-memory hash database. If it is "+", the database will be an on-memory tree database. If its suffix is ".tch", the database will be a

hash database. If its suffix is ".tcb", the database will be a B+ tree database. If its suffix is ".tcf", the database will be a fixed-length database. If its suffix is ".tct", the database will be a table database. Otherwise, this function fails. Tuning parameters can trail the name, separated by "#". Each parameter is composed of the name and the value, separated by "=". On-memory hash database supports "bnum", "capnum", and "capsiz". On-memory tree database supports "capnum" and "capsiz". Hash database supports "mode", "bnum", "apow", "fpow", "opts", "rcnum", "xmsiz", and "dfunit". B+ tree database supports "mode", "lmemb", "nmemb", "bnum", "apow", "fpow", "opts", "lcnum", "ncnum", "xmsiz", and "dfunit". Fixed-length database supports "mode", "width", and "limsiz". Table database supports "mode", "bnum", "apow", "fpow", "opts", "rcnum", "lcnum", "ncnum", "xmsiz", "dfunit", and "idx".

If successful, the return value is true, else, it is false.

The tuning parameter "capnum" specifies the capacity number of records. "capsiz" specifies the capacity size of using memory. Records spilled the capacity are removed by the storing order. "mode" can contain "w" of writer, "r" of reader, "c" of creating, "t" of truncating, "e" of no locking, and "f" of non-blocking lock. The default mode is relevant to "wc". "opts" can contains "l" of large option, "d" of Deflate option, "b" of BZIP2 option, and "t" of TCBS option. "idx" specifies the column name of an index and its type separated by ":". For example, "casket.tch#bnum=1000000#opts=ld" means that the name of the database file is "casket.tch", and the bucket number is 1000000, and the options are large and Deflate.

The function `tcadbclose' is used in order to close an abstract database object.

```
bool tcadbclose(TCADB *adb);
```

`*adb*' specifies the abstract database object.

If successful, the return value is true, else, it is false.

Update of a database is assured to be written when the database is closed. If a writer opens a database but does not close it appropriately, the database will be broken.

The function `tcadbput' is used in order to store a record into an abstract database object.

```
bool tcadbput(TCADB *adb, const void *kbuf, int ksiz, const void *vbuf, int vsiz);
```

`*adb*' specifies the abstract database object.

`*kbuf*' specifies the pointer to the region of the key.

`*ksiz*' specifies the size of the region of the key.

`*vbuf*' specifies the pointer to the region of the value.

`*vsiz*' specifies the size of the region of the value.

If successful, the return value is true, else, it is false.

If a record with the same key exists in the database, it is overwritten.

The function `tcadbput2' is used in order to store a string record into an abstract object.

```
bool tcadbput2(TCADB *adb, const char *kstr, const char *vstr);
```

`*adb*' specifies the abstract database object.

`*kstr*' specifies the string of the key.

`*vstr*' specifies the string of the value.

If successful, the return value is true, else, it is false.

If a record with the same key exists in the database, it is overwritten.

The function `tcadbputkeep' is used in order to store a new record into an abstract database object.

bool tcadbputkeep(TCADB *_adb_, const void *_kbuf_, int _ksiz_, const void *_vbuf_, int _vsiz_);

`_adb_' specifies the abstract database object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_vbuf_' specifies the pointer to the region of the value.
`_vsiz_' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tcadbputkeep2' is used in order to store a new string record into an abstract database object.

bool tcadbputkeep2(TCADB *_adb_, const char *_kstr_, const char *_vstr_);

`_adb_' specifies the abstract database object.
`_kstr_' specifies the string of the key.
`_vstr_' specifies the string of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tcadbputcat' is used in order to concatenate a value at the end of the existing record in an abstract database object.

bool tcadbputcat(TCADB *_adb_, const void *_kbuf_, int _ksiz_, const void *_vbuf_, int _vsiz_);

`_adb_' specifies the abstract database object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_vbuf_' specifies the pointer to the region of the value.
`_vsiz_' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tcadbputcat2' is used in order to concatenate a string value at the end of the existing record in an abstract database object.

bool tcadbputcat2(TCADB *_adb_, const char *_kstr_, const char *_vstr_);

`_adb_' specifies the abstract database object.
`_kstr_' specifies the string of the key.
`_vstr_' specifies the string of the value.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tcadbout' is used in order to remove a record of an abstract database object.

bool tcadbout(TCADB *_adb_, const void *_kbuf_, int _ksiz_);

`_adb_' specifies the abstract database object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
If successful, the return value is true, else, it is false.

The function `tcadbout2' is used in order to remove a string record of an abstract database object.

```
bool tcadbout2(TCADB *adb, const char *kstr);
```

`*adb*' specifies the abstract database object.
`*kstr*' specifies the string of the key.
If successful, the return value is true, else, it is false.

The function `tcadbget' is used in order to retrieve a record in an abstract database object.

```
void *tcadbget(TCADB *adb, const void *kbuf, int ksiz, int *sp);
```

`*adb*' specifies the abstract database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the value of the corresponding record. `NULL' is returned if no record corresponds.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcadbget2' is used in order to retrieve a string record in an abstract database object.

```
char *tcadbget2(TCADB *adb, const char *kstr);
```

`*adb*' specifies the abstract database object.
`*kstr*' specifies the string of the key.
If successful, the return value is the string of the value of the corresponding record. `NULL' is returned if no record corresponds.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcadbvsiz' is used in order to get the size of the value of a record in an abstract database object.

```
int tcadbvsiz(TCADB *adb, const void *kbuf, int ksiz);
```

`*adb*' specifies the abstract database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tcadbvsiz2' is used in order to get the size of the value of a string record in an abstract database object.

```
int tcadbvsiz2(TCADB *adb, const char *kstr);
```

`*adb*' specifies the abstract database object.
`*kstr*' specifies the string of the key.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tcadbiterinit' is used in order to initialize the iterator of an abstract database object.

```
bool tcadbiterinit(TCADB *adb);
```

`*adb*' specifies the abstract database object.
If successful, the return value is true, else, it is false.
The iterator is used in order to access the key of every record stored in a database.

The function `tcadbiternext' is used in order to get the next key of the iterator of an abstract database object.

```
void *tcadbiternext(TCADB *adb, int *sp);
```

`*adb*' specifies the abstract database object.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the next key, else, it is `NULL'. `NULL' is returned when no record is to be get out of the iterator.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. It is possible to access every record by iteration of calling this function. It is allowed to update or remove records whose keys are fetched while the iteration. However, it is not assured if updating the database is occurred while the iteration. Besides, the order of this traversal access method is arbitrary, so it is not assured that the order of storing matches the one of the traversal access.

The function `tcadbiternext2' is used in order to get the next key string of the iterator of an abstract database object.

```
char *tcadbiternext2(TCADB *adb);
```

`*adb*' specifies the abstract database object.
If successful, the return value is the string of the next key, else, it is `NULL'. `NULL' is returned when no record is to be get out of the iterator.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. It is possible to access every record by iteration of calling this function. However, it is not assured if updating the database is occurred while the iteration. Besides, the order of this traversal access method is arbitrary, so it is not assured that the order of storing matches the one of the traversal access.

The function `tcadbfwmkeys' is used in order to get forward matching keys in an abstract database object.

```
TCLIST *tcadbfwmkeys(TCADB *adb, const void *pbuf, int psiz, int max);
```

`*adb*' specifies the abstract database object.
`*pbuf*' specifies the pointer to the region of the prefix.
`*psiz*' specifies the size of the region of the prefix.
`*max*' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.
The return value is a list object of the corresponding keys. This function does never fail. It returns an empty list even if no key corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use. Note that this function may be very slow because every key in the database is scanned.

The function `tcadbfwmkeys2' is used in order to get forward matching string keys in an abstract database object.

```
TCLIST *tcadbfwmkeys2(TCADB *adb, const char *pstr, int max);
```

`*adb*' specifies the abstract database object.
`*pstr*' specifies the string of the prefix.
`*max*' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.
The return value is a list object of the corresponding keys. This function does never fail. It returns an empty list even if no key corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use. Note that this function may be very slow because every key in the database is scanned.

The function `tcadbaddint' is used in order to add an integer to a record in an abstract database object.

```
int tcadbaddint(TCADB *adb, const void *kbuf, int ksiz, int num);
```

`*adb*' specifies the abstract database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*num*' specifies the additional value.
If successful, the return value is the summation value, else, it is `INT_MIN'.
If the corresponding record exists, the value is treated as an integer and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tcadbadddouble' is used in order to add a real number to a record in an abstract database object.

```
double tcadbadddouble(TCADB *adb, const void *kbuf, int ksiz, double num);
```

`*adb*' specifies the abstract database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*num*' specifies the additional value.
If successful, the return value is the summation value, else, it is Not-a-Number.
If the corresponding record exists, the value is treated as a real number and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tcadbsync' is used in order to synchronize updated contents of an abstract database object with the file and the device.

```
bool tcadbsync(TCADB *adb);
```

`*adb*' specifies the abstract database object.
If successful, the return value is true, else, it is false.

The function `tcadboptimize' is used in order to optimize the storage of an abstract database object.

```
bool tcadboptimize(TCADB *adb, const char *params);
```

`*adb*' specifies the abstract database object.
`*params*' specifies the string of the tuning parameters, which works as with the tuning of parameters the function `tcadbopen'. If it is `NULL', it is not used.
If successful, the return value is true, else, it is false.
This function is useful to reduce the size of the database storage with data fragmentation by successive updating.

The function `tcadbvanish' is used in order to remove all records of an abstract database object.

```
bool tcadbvanish(TCADB *adb);
```

`*adb*' specifies the abstract database object.
If successful, the return value is true, else, it is false.

The function `tcadbcopy' is used in order to copy the database file of an abstract database object.

```
bool tcadbcopy(TCADB *adb, const char *path);
```

`*adb*' specifies the abstract database object.
`*path*' specifies the path of the destination file. If it begins with `@', the trailing substring is executed as a command line.
If successful, the return value is true, else, it is false. False is returned if the executed command returns non-zero code.
The database file is assured to be kept synchronized and not modified while the copying or executing operation is in progress. So, this function is useful to create a backup file of the database file.

The function `tcadbtranbegin' is used in order to begin the transaction of an abstract database object.

```
bool tcadbtranbegin(TCADB *adb);
```

`*adb*' specifies the abstract database object.
If successful, the return value is true, else, it is false.
The database is locked by the thread while the transaction so that only one transaction can be activated with a database object at the same time. Thus, the serializable isolation level is assumed if every database operation is performed in the transaction. All updated regions are kept track of by write ahead logging while the transaction. If the database is closed during transaction, the transaction is aborted implicitly.

The function `tcadbtrancommit' is used in order to commit the transaction of an abstract database object.

```
bool tcadbtrancommit(TCADB *adb);
```

`*adb*' specifies the abstract database object.
If successful, the return value is true, else, it is false.
Update in the transaction is fixed when it is committed successfully.

The function `tcadbtranabort' is used in order to abort the transaction of an abstract database object.

```
bool tcadbtranabort(TCADB *adb);
```

`*adb*' specifies the abstract database object.
If successful, the return value is true, else, it is false.
Update in the transaction is discarded when it is aborted. The state of the database is rollbacked to before transaction.

The function `tcadbpath' is used in order to get the file path of an abstract database object.

```
const char *tcadbpath(TCADB *adb);
```

`*adb*' specifies the abstract database object.
The return value is the path of the database file or `NULL' if the object does not connect to any database. "*" stands for on-memory hash database. "+" stands for on-memory tree database.

The function `tcadbrnum' is used in order to get the number of records of an abstract database object.

`uint64_t tcadbrnum(TCADB *adb);`

`*adb*' specifies the abstract database object.

The return value is the number of records or 0 if the object does not connect to any database instance.

The function `tcadbsize' is used in order to get the size of the database of an abstract database object.

`uint64_t tcadbsize(TCADB *adb);`

`*adb*' specifies the abstract database object.

The return value is the size of the database or 0 if the object does not connect to any database instance.

The function `tcadbmisc' is used in order to call a versatile function for miscellaneous operations of an abstract database object.

`TCLIST *tcadbmisc(TCADB *adb, const char *name, const TCLIST *args);`

`*adb*' specifies the abstract database object.

`*name*' specifies the name of the function. All databases support "put", "out", "get", "putlist", "outlist", "getlist", and "getpart". "put" is to store a record. It receives a key and a value, and returns an empty list. "out" is to remove a record. It receives a key, and returns an empty list. "get" is to retrieve a record. It receives a key, and returns a list of the values. "putlist" is to store records. It receives keys and values one after the other, and returns an empty list. "outlist" is to remove records. It receives keys, and returns an empty list. "getlist" is to retrieve records. It receives keys, and returns keys and values of corresponding records one after the other. "getpart" is to retrieve the partial value of a record. It receives a key, the offset of the region, and the length of the region.

`*args*' specifies a list object containing arguments.

If successful, the return value is a list object of the result. `NULL' is returned on failure.

Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

Example Code

The following code is an example to use an abstract database.

```
#include <tcutil.h>
#include <tcadb.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>

int main(int argc, char **argv){
  TCADB *adb;
  char *key, *value;

  /* create the object */
  adb = tcadbnew();

  /* open the database */
  if(!tcadbopen(adb, "casket.tch")){
    fprintf(stderr, "open error\n");
  }

  /* store records */
  if(!tcadbput2(adb, "foo", "hop") ||
```

```
    !tcadbput2(adb, "bar", "step") ||
    !tcadbput2(adb, "baz", "jump")){
   fprintf(stderr, "put error\n");
 }

 /* retrieve records */
 value = tcadbget2(adb, "foo");
 if(value){
   printf("%s\n", value);
   free(value);
 } else {
   fprintf(stderr, "get error\n");
 }

 /* traverse records */
 tcadbiterinit(adb);
 while((key = tcadbiternext2(adb)) != NULL){
   value = tcadbget2(adb, key);
   if(value){
     printf("%s:%s\n", key, value);
     free(value);
   }
   free(key);
 }

 /* close the database */
 if(!tcadbclose(adb)){
   fprintf(stderr, "close error\n");
 }

 /* delete the object */
 tcadbdel(adb);

 return 0;
}
```

CLI

To use the abstract database API easily, the commands `**tcatest**', `**tcamttest**' and `**tcamgr**' are provided.

The command `**tcatest**' is a utility for facility test and performance test. This command is used in the following format. `*name*' specifies the database name. `*rnum*' specifies the number of iterations. `*tnum*' specifies the number of transactions.

tcatest write *name rnum*
: Store records with keys of 8 bytes. They change as `00000001', `00000002'...

tcatest read *name*
: Retrieve all records of the database above.

tcatest remove *name*
: Remove all records of the database above.

tcatest rcat *name rnum*
: Store records with partway duplicated keys using concatenate mode.

tcatest misc *name rnum*
: Perform miscellaneous test of various operations.

tcatest wicked *name rnum*
: Perform updating operations of list and map selected at random.

tcatest compare ***name tnum rnum***
: Perform comparison test of database schema.

This command returns 0 on success, another on failure.

The command `**tcamttest**' is a utility for facility test under multi-thread situation. This command is used in the following format. `*name*' specifies the database name. `*tnum*' specifies the number of running threads. `*rnum*' specifies the number of iterations.

tcamttest write ***name tnum rnum***
: Store records with keys of 8 bytes. They change as `00000001', `00000002'...

tcamttest read ***name tnum***
: Retrieve all records of the database above.

tcamttest remove ***name tnum***
: Remove all records of the database above.

This command returns 0 on success, another on failure.

The command `**tcamgr**' is a utility for test and debugging of the abstract database API and its applications. `*name*' specifies the name of a database. `*key*' specifies the key of a record. `*value*' specifies the value of a record. `*params*' specifies the tuning parameters. `*func*' specifies the name of a function. `*arg*' specifies the arguments of the function. `*dest*' specifies the path of the destination file.

tcamgr create ***name***
: Create a database file.

tcamgr inform ***name***
: Print miscellaneous information to the standard output.

tcamgr put [-sx] [-sep ***chr***] **[-dk|-dc|-dai|-dad]** ***name key value***
: Store a record.

tcamgr out [-sx] [-sep ***chr***] ***name key***
: Remove a record.

tcamgr get [-sx] [-sep ***chr***] **[-px] [-pz]** ***name key***
: Print the value of a record.

tcamgr list [-sep ***chr***] **[-m** ***num***] **[-pv] [-px] [-fm** ***str***] ***name***
: Print keys of all records, separated by line feeds.

tcamgr optimize ***name params***
: Optimize a database file.

tcamgr misc [-sx] [-sep ***chr***] **[-px]** ***name func*** **[*arg*...]**
: Call a versatile function for miscellaneous operations.

tcamgr map [-fm ***str***] ***name dest***
: Map records into another B+ tree database.

tcamgr version
: Print the version information of Tokyo Cabinet.

Options feature the following.

-sx : the input data is evaluated as a hexadecimal data string.

-sep ***chr*** : specify the separator of the input data.
-dk : use the function `tcadbputkeep' instead of `tcadbput'.
-dc : use the function `tcadbputcat' instead of `tcadbput'.
-dai : use the function `tcadbaddint' instead of `tcadbput'.
-dad : use the function `tcadbadddouble' instead of `tcadbput'.
-px : the output data is converted into a hexadecimal data string.
-pz : do not append line feed at the end of the output.
-m ***num*** : specify the maximum number of the output.
-pv : print values of records also.
-fm ***str*** : specify the prefix of keys.

This command returns 0 on success, another on failure.

CGI

To use the abstract database API easily, the CGI script `**tcawmgr.cgi**' is provided.

The CGI script `**tcawmgr.cgi**' is a utility to browse and edit an abstract database by Web interface. The database should be placed in the same directory of the CGI script and named as "**casket.tch**", "**casket.tcb**", or "**casket.tcf**". And, its permission should allow reading and writing by the user executing the CGI script. Install the CGI script in a public directory of your Web server then you can start to use the CGI script by accessing the assigned URL.

File Format

This section describes the format of the database files of Tokyo Cabinet.

File Format of Hash Database

There are four sections in the file managed by the hash database; the header section, the bucket section, the free block pool section, and the record section. Numeric values in the file are serialized in the little endian order or in the variable length format. The latter format is delta encoding based on the 128-radix numbering.

The header section is from the top of the file and its length is 256 bytes. There are the following information.

name	offset	length	feature
magic number	0	32	identification of the database. Begins with "ToKyO CaBiNeT"
database type	32	1	hash (0x01) / B+ tree (0x02) / fixed-length (0x03) / table (0x04)
additional flags	33	1	logical union of open (1<<0) and fatal (1<<1)
alignment power	34	1	the alignment size, by power of 2
free block pool power	35	1	the number of elements in the free block pool, by power of 2
options	36	1	logical union of large (1<<0), Deflate (1<<1), BZIP2 (1<<2), TCBS (1<<3), extra codec (1<<4)
bucket number	40	8	the number of elements of the bucket array
record number	48	8	the number of records in the database
file size	56	8	the file size of the database
first record	64	8	the offset of the first record
opaque region	128	128	users can use this region arbitrarily

The bucket section trails the header section and its size is defined by the bucket number. Each element of the bucket array indicates the offset of the first record of the hash chain. The format of each element is the fixed length number and its size is 4 bytes in the normal mode or 8 bytes in the large mode. The offset is recorded as the quotient by the alignment.

The free block pool section trails the bucket section and its size is defined by the free block pool number. Each element of the free block pool indicates the offset and the size of each free block. The offset is recorded as the difference of the former free block and as the quotient by the alignment. The offset and the size are serialized in the variable length format.

The record section trails the free block pool section and occupies the rest region to the end of the file. Each element has the following information. The region of each record begins at the offset of the multiple of the alignment.

name	offset	length	feature
magic number	0	1	identification of record block. always 0xC8

hash value	1	1	the hash value to decide the path of the hash chain
left chain	2	4	the alignment quotient of the destination of the left chain
right chain	6	4	the alignment quotient of the destination of the right chain
padding size	10	2	the size of the padding
key size	12	vary	the size of the key
value size	vary	vary	the size of the value
key	vary	vary	the data of the key
value	vary	vary	the data of the value
padding	vary	vary	useless data

However, regions of free blocks contain the following information.

name	offset	length	feature
magic number	0	1	identification of record block. always 0xB0
block size	1	4	size of the block

The transaction log is recorded in the file whose name is composed of the database name and the suffix ".wal". The top eight bytes indicate the file size of the beginning of the transaction. After that, there are the following information.

name	offset	length	feature
offset	0	8	the offset of the updated region
size	8	4	the size of the updated region
data	12	vary	the data before update

File Format of B+ Tree Database

All data managed by the B+ tree database are recorded in the hash database. Recorded data are classified into meta data and logical pages. Logical pages are classified into leaf nodes and non-leaf nodes. The formats of the fixed length number and the variable length number are the same as with the hash database.

Meta data are recorded in the opaque region in the header of the hash database and have the following information.

name	offset	length	feature
comparison function	0	1	tccmplexical (0x00), tccmpdecimal (0x01), tccmpint32 (0x02), tccmpint64 (0x03), other (0xff)
reserved region	1	7	not used
record number of leaf node	8	4	the maximum number of records in a leaf node
index number of non-leaf node	12	4	the maximum number of indices in a leaf node
root node ID	16	8	the page ID of the root node of B+ tree
first leaf ID	24	8	the page ID of the first leaf node
last leaf ID	32	8	the page ID of the last leaf node
leaf number	40	8	the number of the leaf nodes
non-leaf number	48	8	the number of the non-leaf nodes
record number	56	8	the number of records in the database

Each leaf node contains a list of records. Each non-leaf node contains a list of indices to child nodes. Though each record is a logical unit of user data, records with the same key are integrated into one record physically. Each physical record has the following information.

name	offset	length	feature
key size	0	vary	the size of the key
value size	vary	vary	the size of the value
duplication number	vary	vary	the number of values with the same key
key	vary	vary	the data of the key
value	vary	vary	the data of the value
duplicated records	vary	vary	a list of value sizes and value data

Each leaf node is a physical unit of a set of records. Each leaf node is identified by the sequential ID number from 1. Each leaf node is recorded in the hash database. The key is a string in the hexadecimal numbering. The value has the following information. Records are kept in the ascending order of keys.

name	offset	length	feature
previous leaf	0	vary	the ID number of the previous leaf node
next leaf	vary	vary	the ID number of the next leaf node
record list	vary	vary	the serialized data of all records in the node

Each index is a logical unit of pointer to the child node. Each index has the following information.

name	offset	length	feature
page ID	0	vary	the ID number of the referred page
key size	vary	vary	the size of the key
key	vary	vary	the data of the key

Each non-leaf node is a physical unit of a set of indices. Each non-leaf node is identified by the sequential number from 281474976710657. Each non-leaf node is recorded in the hash database. The key is a string begins with "#" and is trailed by the hexadecimal number of the ID number subtracted by 281474976710657. The value has the following information. Indices are kept in the ascending order of keys.

name	offset	length	feature
accession ID	0	vary	the ID number of the first child node
index list	vary	vary	the serialized data of all indices in the node

File Format of Fixed-length Database

There are two sections in the file managed by the fixed-length database; the header section, and the record section. Numeric values in the file are serialized in the little endian order.

The header section is from the top of the file and its length is 256 bytes. There are the following information.

name	offset	length	feature
magic number	0	32	identification of the database. Begins with "ToKyO CaBiNeT"
database type	32	1	always 0x03

additional flags	33	1	logical union of open (1<<0) and fatal (1<<1)
record number	48	8	the number of records in the database
file size	56	8	the file size of the database
record width	64	8	the width of each record
limit size	72	8	the limit size of the database
least ID	80	8	the least ID number of records
greatest ID	88	8	the greatest ID number of records
opaque region	128	128	users can use this region arbitrarily

The record section trails the header section and occupies the rest region to the end of the file. Each element has the following information. The size region takes 1 byte if the record width is less than 256 bytes, or takes 2 bytes if the record width is less than 65536, else takes 4 bytes. The size of each record is the summation of the size of the width region and the record width. So, the region of each record begins at the offset generated by the ID number subtracted by 1 and multiplied by the record width and the added by 256.

name	offset	length	feature
value size	0	vary	the size of the value
value	vary	vary	the data of the value
padding	vary	vary	padding. If the size of the value is 0, the first byte indicates whether the record exists or not

The naming convention and the file format of the transaction log file is the same as the one of the hash database.

Note

Because database files are not sparse, you can copy them as with normal files. Moreover, the database formats don't depend on the byte order of the running environment, you can migrate the database files between environments with different byte orders.

If possible, set the MIME type `**application/x-tokyocabinet-hash**' when sending files of the hash database. The suffix of the file name should be `**.tch**'. As for the B+ tree database, `**application/x-tokyocabinet-btree**' and `**.tcb**'. As for the fixed-length database, `**application/x-tokyocabinet-fixed**' and `**.tcf**'. As for the table database, `**application/x-tokyocabinet-btree**' and `**.tct**'.

To make the `**file**' command identify the database formats, append the following lines to the `**magic**' file.

```
# Tokyo Cabinet magic data
0       string    ToKyO\ CaBiNeT\n   Tokyo Cabinet
>14     string    x                  \b (%s)
>32     byte      0                  \b, Hash
!:mime  application/x-tokyocabinet-hash
>32     byte      1                  \b, B+ tree
!:mime  application/x-tokyocabinet-btree
>32     byte      2                  \b, Fixed-length
!:mime  application/x-tokyocabinet-fixed
>32     byte      3                  \b, Table
!:mime  application/x-tokyocabinet-table
>33     byte      &1                 \b, [open]
```

```
>33     byte      &2                \b, [fatal]
>34     byte      x                 \b, apow=%d
>35     byte      x                 \b, fpow=%d
>36     byte      &1                \b, [large]
>36     byte      &2                \b, [deflate]
>36     byte      &4                \b, [bzip]
>36     byte      &8                \b, [tcbs]
>36     byte      &16               \b, [excodec]
>40     lequad    x                 \b, bnum=%lld
>48     lequad    x                 \b, rnum=%lld
>56     lequad    x                 \b, fsiz=%lld
```

License

Tokyo Cabinet is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License or any later version.

Tokyo Cabinet is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with Tokyo Cabinet (See the file `**COPYING**'); if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

Tokyo Cabinet was written by FAL Labs. You can contact the author by e-mail to `**info@fallabs.com**'.