NAME

smdb_dbf_create, smdb_dbf_open, smdb_dbf_free, smdb_dbf_create_table, smdb_dbf_free_table, smdb_dbf_sync, smdb_dbf_begin, smdb_dbf_end, smdb_dbf_rollback, smdb_dbf_get, smdb_dbf_get_next, smdb_dbf_first, smdb_dbf_next, smdb_dbf_free_record, smdb_dbf_put, smdb_dbf_erase

SYNOPSIS

#include <smdb-incl.h>

int *smdb_dbf_create*(**struct smdbxi_factory** * *fac*, **struct smdbxi_file** **bfile*,**struct smdb_db_config const** **dbcfg*, **struct smdb_dbfile_ctx** ** *pdfctx*);
int *smdb_dbf_open*(**struct smdbxi_factory** * *fac*, **struct smdbxi_file** **bfile*, **struct smdb_db_config const** **dbcfg*, **struct smdb_dbfile_ctx** ** *pdfctx*);
void *smdb_dbf_free*(**struct smdb_dbfile_ctx** **dfctx*);
int *smdb_dbf_create_table*(**struct smdb_dbfile_ctx** **dfctx*, **unsigned int** *tblid*, **unsigned int** *tblsize*);
int *smdb_dbf_free_table*(**struct smdb_dbfile_ctx** **dfctx*, **unsigned int** *tblid*);
int *smdb_dbf_sync*(**struct smdb_dbfile_ctx** **dfctx*);
int *smdb_dbf_begin*(**struct smdb_dbfile_ctx** **dfctx*);
int *smdb_dbf_end*(**struct smdb_dbfile_ctx** **dfctx*);
int *smdb_dbf_rollback*(**struct smdb_dbfile_ctx** **dfctx*);
int *smdb_dbf_get*(**struct smdb_dbfile_ctx** **dfctx*, **unsigned int** *tblid*, **struct smdb_db_ckey const** **key*, **struct smdb_db_record** **rec*,**struct smdb_db_kenum** **ken*);
int *smdb_dbf_get_next*(**struct smdb_dbfile_ctx** **dfctx*, **struct smdb_db_ckey const** **key*, **struct smdb_db_record** **rec*, **struct smdb_db_kenum** **ken*);
int *smdb_dbf_first*(**struct smdb_dbfile_ctx** **dfctx*, **unsigned int** *tblid*, **struct smdb_db_record** **rec*, **struct smdb_db_kenum** **ken*);
int *smdb_dbf_next*(**struct smdb_dbfile_ctx** **dfctx*, **struct smdb_db_record** **rec*, **struct smdb_db_kenum** **ken*);
void *smdb_dbf_free_record*(**struct smdb_dbfile_ctx** **dfctx*, **struct smdb_db_record** **rec*);
int *smdb_dbf_put*(**struct smdb_dbfile_ctx** **dfctx*, **unsigned int** *tblid*, **struct smdb_db_ckey const** **key*, **struct smdb_db_cdata** **data*);
int *smdb_dbf_erase*(**struct smdb_dbfile_ctx** **dfctx*, **unsigned int** *tblid*, **struct smdb_db_ckey const** **key*, **struct smdb_db_cdata const** **data*);

DESCRIPTION

The **SMDB** library exports a database API somehow similar to the one supplied by the GNU GDBM library, with a few differences. The **SMDB** database library is totally system independent, via the use of externally supplied interface pointers. The **SMDB** databse library does not try to implement any file locking, which is left to the caller, if the particular use requires it. This allows single instance users to not pay the the price (in terms of performance and portability) of the locking framework. The **SMDB** database library supports transactions, multiple tables inside the same DB file, and is less than half the size of GDBM. Finally, the **SMDB** database library is available under the loser LGPL license, instead of the GPL one. Conceptually though, both allows for simple storage and retrieval of KEY+DATA couples, inside a table.

Structures And Macros

The **SMDB** library uses externally supplied interfaces for every task that is related with the host system interaction. This allows for a totally portable implementation, that was one of the major objectives of its development. An interface is composed by an opaque data pointer *priv* and a set of methods (functions pointers) exported by the interface. All the interfaces exports the two basic methods *get* and *release*. The *get* method acquires an interface and makes sure it remains valid during the whole time it is used. Once an instance of an interface is no more used, the *release* method should be called. Two macros are available for this purpose:

SMDBXI_GET(*iface***)**

Get an instance of the interface pointed by *iface*.

SMDBXI_RELEASE(*iface***)**

Release an instance of the interface pointed by *iface*. An example implementation of the interfaces required by **SMDB** is supplied next in this document, and is also available inside the **SMDB** distribution package.

All the memory management used by the **SMDB** library is defined by the **struct smdbxi_mem** interface:

struct smdbxi_mem

```
struct smdbxi_mem
{
        void *priv;
        int (*get) (void *priv);
        int (*release) (void *priv);
        void *(*alloc) (void *priv, int size);
        void (*free) (void *priv, void *ptr);
};
```

The *alloc* method allows the caller to allocate a block of memory, whereas the method *free* allows the caller to free a block of memory returned by *alloc*. The following macros are available to access the **struct smdbxi_mem** methods:

SMDBXI_MM_ALLOC(*iface***,** *size***)**
Allocates a memory block of *size* bytes using the *iface* memory interface. The macro returns a pointer to the newly allocated memory block, or **NULL** in case of failure.

SMDBXI_MM_FREE(*iface***,** *ptr***)**
Frees a memory block pointed by *ptr* using the *iface* memory interface. It is allowed to call the **SMDBXI_MM_FREE** macro by passing **NULL** as *ptr* parameter.

The abstraction of a file is based on the **struct smdbxi_file** interface:

struct smdbxi_file

```
struct smdbxi_file
{
        void *priv;
        int (*get)(void *);
        int (*release)(void *);
        smdb_offset_t (*seek)(void *, smdb_offset_t, int);
        int (*read)(void *, void *, int);
        int (*write)(void *, void const *, int);
        int (*truncate)(void *, smdb_offset_t);
        int (*sync)(void *);
        char const *(*path)(void *);
};
```

Besides from the common *get* and *release* methods, the **struct smdbxi_file** interface defines functions to access the file API. Macros are availble to access the **struct smdbxi_file** interface:

SMDBXI_FL_SEEK(*iface***,** *offset***,** *whence***)**
Set the file position at *offset* bytes from the *whence* location. Valid constants for the *whence* location are:

SMDBXI_FL_SEEKSET
The *offset* is relative to the beginning of the file.

SMDBXI_FL_SEEKCUR
The *offset* is relative to the current file position.

SMDBXI_FL_SEEKEND
The *offset* is relative to the end of the file.

The **SMDBXI_FL_SEEK** returns the current file position from the beginning of the file, or -1 in case of error.

SMDBXI_FL_READ(*iface***,** *buffer***,** *size***)**
Read *size* bytes into the buffer pointer by *buffer* using the *iface* file interface. The function returns the number of bytes read, that can be lower than the requested one in case of EOF, or -1 in case of error.

SMDBXI_FL_WRITE(*iface***,** *buffer***,** *size***)**
Write *size* bytes from the buffer pointer by *buffer* using the *iface* file interface. The function returns the number of bytes written, or -1 in case of error.

SMDBXI_FL_TRUNCATE(*iface***,** *length***)**
Truncate the file identified by the *iface* interface to *length* bytes. The new length can be lower of higher than the current file size. The function returns 0 if succeeded, or -1 in case of error.

SMDBXI_FL_SYNC(*iface***)**
Sync the buffers content of the file identified by the *iface* interface to the underlying storage media. The function returns 0 if succeeded, or -1 in case of error.

SMDBXI_FL_PATH(*iface***)**
Returns the path of the file identified by the *iface* interface.

The **struct smdbxi_fs** interface provide access to some basic file system operations.

struct smdbxi_fs

```
struct smdbxi_fs {
        void *priv;
        int (*get)(void *);
        int (*release)(void *);
        struct smdbxi_file *(*open)(void *, char const *, int);
        int (*remove)(void *, char const *);
        int (*rename)(void *, char const *, char const *);
        int (*mkdir)(void *, char const *);
        int (*rmdir)(void *, char const *);
};
```

The following macros are availble to access the **struct smdbxi_file** interface (on top of the usual get/release):

SMDBXI_FS_OPEN(*iface***,** *path***,** *mode***)**
Opens the file specified in *path* using the *mode* parameter to request diffrent type of access. Valid values for the *mode* parameter are:

SMDBXI_FL_ROPEN
Opens an existing file in read-only mode.

SMDBXI_FL_RWOPEN
Opens an existing file in read-write mode.

SMDBXI_FL_CREATE
Tries to open an existing file in read-write mode, and creates it if the file specified by *path* does not exist.

SMDBXI_FL_CREATENEW
Tries to open an existing file in read-write mode and truncates it if exists. If the file specified by *path* does not exist, the underlying callback function creates it.

The **SMDBXI_FL_OPEN** returns the new file interface that can be used to manipulate the newly open file, or **NULL** in case of error.

SMDBXI_FS_REMOVE(*iface***,** *path***)**
Removes the file specified by the *path* parameter. Returns 0 in case of success, or -1 in case of error.

SMDBXI_FS_RENAME(*iface***,** *opath***,** *npath***)**
Renames the file/directory specified by the *opath* parameter, into the path specified in the *npath* parameter. Returns 0 in case of success, or -1 in case of error.

SMDBXI_FS_MKDIR(*iface***,** *path***)**
Creates the directory specified by the *path* parameter. Returns 0 in case of success, or -1 in case of error.

SMDBXI_FS_RMDIR(*iface***,** *path***)**
Removes the directory specified by the *path* parameter. Returns 0 in case of success, or -1 in case of error.

The **struct smdbxi_factory** interface is the factory of all the other interfaces:

struct smdbxi_factory

```
struct smdbxi_factory {
        void *priv;
        int (*get)(void *);
        int (*release)(void *);
        struct smdbxi_mem *(*mem)(void *);
        struct smdbxi_file *(*file)(void *);
        struct smdbxi_fs *(*fs)(void *);
};
```

Besides from the usual *get* and *release* methods, other function are available to instantiate other interfaces. The *mem* method allows the creation of a memory interface, whereas the *file* method creates a file interface. Macros are defined to access the **struct smdbxi_factory** functions:

SMDBXI_FC_MEM(*iface***)**
Returns a pointer to a memory interface (**struct smdbxi_mem**), or **NULL** in case of error.

SMDBXI_FC_FILE(*iface***)**
Returns a pointer to a file interface (**struct smdbxi_file**), or **NULL** in case of error. The file will be a temporary empty file, which will be removed once closed.

SMDBXI_FC_FS(*iface***)**
Returns a pointer to a file system interface (**struct smdbxi_fs**), or **NULL** in case of error.

Functions

int *smdb_dbf_create***(struct smdbxi_factory *** *fac***, struct smdbxi_file ***bfile***,struct smdb_db_config const ***dbcfg***, struct smdb_dbfile_ctx **** *pdfctx***);**

Creates a new database file, using the file interface supplied in *bfile* as storage. The *fac* parameter represent the interface factory, while the parameter *dbcfg* is used to configure the database creation. The *dbcfg* is a pointer to the following structure:

```
struct smdb_db_config {
        smdb_u32 blk_size;
        smdb_u32 blk_count;
        smdb_u32 cache_size;
        smdb_u32 num_tables;
};
```

The **blk_size** member is the requested block size, which cannot be smaller than 256 bytes. The

blk_count is the initial allocation for the database. The database file will automatically grow, but if you know beforehand what you estimated final size will be, guessing the **blk_count** parameter can reduce the number of database file grow operations. The **cache_size** is the amount of RAM (in bytes) to be used for caching the database file blocks. It will be rounded to the next block size. The **num_tables** parameter sets the allocation for the maximum number of tables which will be possible to create inside the database. If the *dbcfg* parameter is **NULL**, default values will be chosen. The *pdfctx* parameter is a pointer to the database accessory structure returned in case of success. The function returns 0 in case of success, and -1 in case of error.

int *smdb_dbf_open*(**struct smdbxi_factory** * *fac*, **struct smdbxi_file** **bfile*, **struct smdb_db_config const** **dbcfg*, **struct smdb_dbfile_ctx** ** *pdfctx*);

Opens an existing database file, using the file interface supplied in *bfile* as storage. The *fac* parameter represent the interface factory, while the parameter *dbcfg* is used to configure the database internals. The only parameter used during the open operation, is the **cache_size** one. The *pdfctx* parameter is a pointer to the database accessory structure returned in case of success. The function returns 0 in case of success, and -1 in case of error.

void *smdb_dbf_free*(**struct smdb_dbfile_ctx** **dfctx*);

Frees the database object pointed by the *dfctx* parameter, and releases all the associated resources.

int *smdb_dbf_create_table*(**struct smdb_dbfile_ctx** **dfctx*, **unsigned int** *tblid*, **unsigned int** *tblsize*);

Creates a table with ID *tblid* inside the database pointed by *dfctx*, with an initial hash allocation close to the value specified in the *tblsize* parameter. The *tblid* value must be lower than the number of tables allocated inside the database file during its creation. The function returns 0 in case of success, and -1 in case of error.

int *smdb_dbf_free_table*(**struct smdb_dbfile_ctx** **dfctx*, **unsigned int** *tblid*);

Frees the table specified by the *tblid* parameter, inside the database pointed by *dfctx*. All the allocation related to the specified table will be released. The function returns 0 in case of success, and -1 in case of error.

int *smdb_dbf_sync*(**struct smdb_dbfile_ctx** **dfctx*);

Syncs the database buffer cache onto the storage media. The *dfctx* represent the database to be sync. If transactions are used, the *smdb_dbf_sync*() function simply causes the dirty buffer cache blocks to be written onto the log. The function returns 0 in case of success, and -1 in case of error.

int *smdb_dbf_begin*(**struct smdb_dbfile_ctx** **dfctx*);

Starts a transation for the database pointed by the *dfctx* parameter. Transactions must not be nested, and must be always terminated by either a call to *smdb_dbf_end*() or *smdb_dbf_rollback*(). The function returns 0 in case of success, and -1 in case of error.

int *smdb_dbf_end*(**struct smdb_dbfile_ctx** **dfctx*);

Ends a transaction previously opened by a call to the *smdb_dbf_begin*() function. After a successful return from *smdb_dbf_end*(), the database operations performed inside the transaction are permanently visible inside the database pointed by *dfctx*. The function returns 0 in case of success, and -1 in case of error.

int *smdb_dbf_rollback***(struct smdb_dbfile_ctx ****dfctx***);**

Rollbacks the operations done after a previous call to *smdb_dbf_begin*() for the database pointed by *dfctx*. The function returns 0 in case of success, and -1 in case of error.

int *smdb_dbf_get***(struct smdb_dbfile_ctx ****dfctx***, unsigned int** *tblid***, struct smdb_db_ckey const ****key***, struct smdb_db_record ****rec***,struct smdb_db_kenum ****ken***);**

Looks up, inside the database pointed by *dfctx* and table *tblid*, the key specified by *key*. The *smdb_dbf_get*() function returns (if any found) the first record matching the passed *key* parameter. The caller can continue the enumeration of the records matching the given *key* using the *smdb_dbf_get_next*() function. The record is stored into the structure pointed by the *rec* parameter, while the enumeration context is stored inside the *ken* parameter (which should be considered a opaque structure by the caller). The caller must use the *smdb_dbf_free_record*() function to free the resources allocated in the *rec* structure, if a positive lookup happen. The function returns a number greater than 0 in case of positive lookup, 0 in case of missing lookup, and -1 in case of error.

int *smdb_dbf_get_next***(struct smdb_dbfile_ctx ****dfctx***, struct smdb_db_ckey const ****key***, struct smdb_db_record ****rec***, struct smdb_db_kenum ****ken***);**

Continue the enumeration initiated with the function *smdb_dbf_get*(), matching, inside the *dfctx* database, the next record with *key* value. The caller must use the *smdb_dbf_free_record*() function to free the resources allocated in the *rec* structure, if a positive lookup happen. The function returns a number greater than 0 in case of positive lookup, 0 in case of missing lookup, and -1 in case of error.

int *smdb_dbf_first***(struct smdb_dbfile_ctx ****dfctx***, unsigned int** *tblid***, struct smdb_db_record ****rec***, struct smdb_db_kenum ****ken***);**

Starts an enumeration of all records inside the *dfctx*, table *tblid*. The record data is stored in the supplied *rec* parameter, while the *ken* parameter is used to store the **SMDB** private enumeration context. The function returns a number greater than 0 in case of positive lookup, 0 in case of missing lookup, and -1 in case of error.

int *smdb_dbf_next***(struct smdb_dbfile_ctx ****dfctx***, struct smdb_db_record ****rec***, struct smdb_db_kenum ****ken***);**

Continues the enumeration of all records inside the *dfctx*, table *tblid*, initiated with a call to *smdb_dbf_first*(). The record data is stored in the supplied *rec* parameter, while the *ken* parameter is used to store the **SMDB** private enumeration context. The function returns a number greater than 0 in case of positive lookup, 0 in case of missing lookup, and -1 in case of error.

void *smdb_dbf_free_record***(struct smdb_dbfile_ctx ****dfctx***, struct smdb_db_record ****rec***);**

Frees the resources allocated for the *rec* record, inside the *dfctx* database. Failure to call *smdb_dbf_free_record*() after a successful lookup from *smdb_dbf_get*(),*smdb_dbf_get_next*(),*smdb_dbf_first*(), or *smdb_dbf_free_record*(), will cause resource leakage inside the system.

int *smdb_dbf_put***(struct smdb_dbfile_ctx ****dfctx***, unsigned int** *tblid***, struct smdb_db_ckey const ****key***, struct smdb_db_cdata ****data***);**

Stores the record specified by the *key* and *data* pair, inside the database *dfctx*, table *tblid*. The function returns 0 in case of success, and -1 in case of error.

int *smdb_dbf_erase***(struct smdb_dbfile_ctx ****dfctx***, unsigned int** *tblid***, struct smdb_db_ckey const ****key***, struct smdb_db_cdata const ****data***);**

Removes a record from inside the *dfctx*, table *tblid*. The *key* parameter specifies the key used to lookup and delete the record. If *data* is **NULL**, the first matching *key* is removed from the database. Otherwise, *data* is used for extra record matching before deletion. The function returns a number greater than 0 in case of positive elimination, 0 in case of missing lookup, and -1 in case of error.

Example Interface Implementation

Here is reported an example implementation for the external interfaces required by the **SMDB** functionality:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "smdb-incl.h"
#include "smdb-xif-posix.h"

#ifdef _WIN32
#include <windows.h>
#include <io.h>

#define O_RDONLY _O_RDONLY
#define O_CREAT _O_CREAT
#define O_RDWR _O_RDWR
#define O_TRUNC _O_TRUNC

#define open(f, m, p) _open(f, (m) | _O_BINARY)
#define read(f, d, n) _read(f, d, n)
#define write(f, d, n) _write(f, d, n)
#define lseek(f, o, w) _lseek(f, o, w)
#define fsync(f) _commit(f)
#define mkdir(n, p) _mkdir(n)
#define rmdir(n) _rmdir(n)

#endif


struct smdbxi_mem_px {
        struct smdbxi_mem ifc;
        long usecnt;
};
```

```
struct smdbxi_file_px {
	struct smdbxi_file ifc;
	long usecnt;
	int fd;
	int closefd;
	char *filename;
	int unlinkfile;
};

struct smdbxi_fs_px {
	struct smdbxi_fs ifc;
	long usecnt;
};

struct smdbxi_factory_px {
	struct smdbxi_factory ifc;
	long usecnt;
	long seqf;
};


#if defined(WIN32)

static int ftruncate(int fd, smdb_offset_t size)
{
	HANDLE hfile;
	LARGE_INTEGER cpos, lsize;

	hfile = (HANDLE) _get_osfhandle(fd);
	lsize.QuadPart = 0;
	if (!SetFilePointerEx(hfile, lsize, &cpos, FILE_CURRENT))
		return -1;
	lsize.QuadPart = size;
	if (!SetFilePointerEx(hfile, lsize, &cpos, FILE_BEGIN))
		return -1;
	if (!SetEndOfFile(hfile)) {
		lsize = cpos;
		SetFilePointerEx(hfile, lsize, &cpos, FILE_BEGIN);
		return -1;
	}
	if (cpos.QuadPart > size)
		cpos.QuadPart = size;
	SetFilePointerEx(hfile, cpos, NULL, FILE_BEGIN);

	return 0;
}

#endif

static int smdb_xif_mem__get(void *priv)
{
	struct smdbxi_mem_px *pif = (struct smdbxi_mem_px *) priv;

	pif->usecnt++;
```

```
        return 0;
}

static int smdb_xif_mem__release(void *priv)
{
        struct smdbxi_mem_px *pif = (struct smdbxi_mem_px *) priv;

        if (!--pif->usecnt) {

                free(pif);
        }

        return 0;
}

static void *smdb_xif_mem__alloc(void *priv, int size)
{
        return malloc(size);
}

static void smdb_xif_mem__free(void *priv, void *data)
{
        free(data);
}

static struct smdbxi_mem *smdb_xif_mem(void)
{
        struct smdbxi_mem_px *pif;

        if ((pif = (struct smdbxi_mem_px *)
            malloc(sizeof(struct smdbxi_mem_px))) == NULL)
                return NULL;
        pif->ifc.priv = pif;
        pif->ifc.get = smdb_xif_mem__get;
        pif->ifc.release = smdb_xif_mem__release;
        pif->ifc.alloc = smdb_xif_mem__alloc;
        pif->ifc.free = smdb_xif_mem__free;
        pif->usecnt = 1;

        return &pif->ifc;
}

static int smdb_xif_file__get(void *priv)
{
        struct smdbxi_file_px *pif = (struct smdbxi_file_px *) priv;

        pif->usecnt++;

        return 0;
}

static int smdb_xif_file__release(void *priv)
{
        struct smdbxi_file_px *pif = (struct smdbxi_file_px *) priv;
```

```
        if (!--pif->usecnt) {
                if (pif->closefd)
                        close(pif->fd);
                if (pif->filename != NULL) {
                        if (pif->unlinkfile)
                                remove(pif->filename);
                        free(pif->filename);
                }
                free(pif);
        }

        return 0;
}

static smdb_offset_t smdb_xif_file__seek(void *priv, smdb_offset_t off, int whence)
{
        struct smdbxi_file_px *pif = (struct smdbxi_file_px *) priv;

        return lseek(pif->fd, (off_t) off, whence);
}

static int smdb_xif_file__read(void *priv, void *buf, int n)
{
        struct smdbxi_file_px *pif = (struct smdbxi_file_px *) priv;

        return read(pif->fd, buf, n);
}

static int smdb_xif_file__write(void *priv, void const *buf, int n)
{
        struct smdbxi_file_px *pif = (struct smdbxi_file_px *) priv;

        return write(pif->fd, buf, n);
}

static int smdb_xif_file__truncate(void *priv, smdb_offset_t size)
{
        struct smdbxi_file_px *pif = (struct smdbxi_file_px *) priv;

        return ftruncate(pif->fd, (off_t) size);
}

static int smdb_xif_file__sync(void *priv)
{
        struct smdbxi_file_px *pif = (struct smdbxi_file_px *) priv;

        return fsync(pif->fd);
}

static char const *smdb_xif_file__path(void *priv)
{
        struct smdbxi_file_px *pif = (struct smdbxi_file_px *) priv;

        return pif->filename;
```

```
}

struct smdbxi_file *smdb_xif_file(int fd, int closefd, char const *filename,
                                  int flags, int unlinkfile)
{
        int lfd = -1;
        struct smdbxi_file_px *pif;

        if (fd < 0) {
                if (filename == NULL)
                        return NULL;
                switch (flags)
                {
                case SMDBXI_FL_ROPEN:
                        flags = O_RDONLY;
                        break;
                case SMDBXI_FL_RWOPEN:
                        flags = O_RDWR;
                        break;
                case SMDBXI_FL_CREATE:
                        flags = O_RDWR | O_CREAT;
                        break;
                case SMDBXI_FL_CREATENEW:
                        flags = O_RDWR | O_CREAT | O_TRUNC;
                        break;
                default:
                        return NULL;
                }
                if ((fd = lfd = open(filename, flags, 0644)) == -1)
                        return NULL;
                closefd = 1;
        }
        if ((pif = (struct smdbxi_file_px *)
            malloc(sizeof(struct smdbxi_file_px))) == NULL) {
                if (lfd != -1) {
                        close(lfd);
                        if (unlinkfile)
                                remove(filename);
                }
                return NULL;
        }
        pif->ifc.priv = pif;
        pif->ifc.get = smdb_xif_file__get;
        pif->ifc.release = smdb_xif_file__release;
        pif->ifc.seek = smdb_xif_file__seek;
        pif->ifc.read = smdb_xif_file__read;
        pif->ifc.write = smdb_xif_file__write;
        pif->ifc.truncate = smdb_xif_file__truncate;
        pif->ifc.sync = smdb_xif_file__sync;
        pif->ifc.path = smdb_xif_file__path;
        pif->usecnt = 1;
        pif->fd = fd;
        pif->closefd = closefd;
        pif->filename = filename != NULL ? strdup(filename): NULL;
```

```
        pif->unlinkfile = unlinkfile;

        return &pif->ifc;
}

static int smdb_xif_fs__get(void *priv)
{
        struct smdbxi_fs_px *pif = (struct smdbxi_fs_px *) priv;

        pif->usecnt++;

        return 0;
}

static int smdb_xif_fs__release(void *priv)
{
        struct smdbxi_fs_px *pif = (struct smdbxi_fs_px *) priv;

        if (!--pif->usecnt) {

                free(pif);
        }

        return 0;
}

static struct smdbxi_file *smdb_xif_fs__open(void *priv, char const *path,
                                              int flags)
{
        return smdb_xif_file(-1, 0, path, flags, 0);
}

static int smdb_xif_fs__remove(void *priv, char const *path)
{
        return remove(path);
}

static int smdb_xif_fs__rename(void *priv, char const *opath, char const *npath)
{
        return rename(opath, npath);
}

static int smdb_xif_fs__mkdir(void *priv, char const *path)
{
        return mkdir(path, 0775);
}

static int smdb_xif_fs__rmdir(void *priv, char const *path)
{
        return rmdir(path);
}

static struct smdbxi_fs *smdb_xif_fs(void)
{
```

```
        struct smdbxi_fs_px *pif;

        if ((pif = (struct smdbxi_fs_px *)
            malloc(sizeof(struct smdbxi_fs_px))) == NULL)
                return NULL;
        pif->ifc.priv = pif;
        pif->ifc.get = smdb_xif_fs__get;
        pif->ifc.release = smdb_xif_fs__release;
        pif->ifc.open = smdb_xif_fs__open;
        pif->ifc.remove = smdb_xif_fs__remove;
        pif->ifc.rename = smdb_xif_fs__rename;
        pif->ifc.mkdir = smdb_xif_fs__mkdir;
        pif->ifc.rmdir = smdb_xif_fs__rmdir;
        pif->usecnt = 1;

        return &pif->ifc;
}

static int smdb_xif_factory__get(void *priv)
{
        struct smdbxi_factory_px *pif = (struct smdbxi_factory_px *) priv;

        pif->usecnt++;

        return 0;
}

static int smdb_xif_factory__release(void *priv)
{
        struct smdbxi_factory_px *pif = (struct smdbxi_factory_px *) priv;

        if (!--pif->usecnt) {

                free(pif);
        }

        return 0;
}

static struct smdbxi_mem *smdb_xif_factory__mem(void *priv)
{
        return smdb_xif_mem();
}

static struct smdbxi_fs *smdb_xif_factory__fs(void *priv)
{
        return smdb_xif_fs();
}

static struct smdbxi_file *smdb_xif_factory__file(void *priv)
{
        struct smdbxi_factory_px *pif = (struct smdbxi_factory_px *) priv;
        char filename[256];
```

```
        sprintf(filename, "%p-%ld.tmp", priv, pif->seqf++);

        return smdb_xif_file(-1, 1, filename, O_CREAT | O_RDWR | O_TRUNC, 1);
}

struct smdbxi_factory *smdb_xif_factory(void)
{
        struct smdbxi_factory_px *pif;

        if ((pif = (struct smdbxi_factory_px *)
            malloc(sizeof(struct smdbxi_factory_px))) == NULL)
                return NULL;
        pif->ifc.priv = pif;
        pif->ifc.get = smdb_xif_factory__get;
        pif->ifc.release = smdb_xif_factory__release;
        pif->ifc.mem = smdb_xif_factory__mem;
        pif->ifc.file = smdb_xif_factory__file;
        pif->ifc.fs = smdb_xif_factory__fs;
        pif->usecnt = 1;
        pif->seqf = 0;

        return &pif->ifc;
}
```

EXAMPLE CODE

The following code (also included inside the **SMDB** distribution package) briefly shows how to use the **SMDB** library:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "smdb-incl.h"
#include "smdb-xif-posix.h"

#define MODE_PUT 1
#define MODE_GET 2
#define MODE_ERASE 3
#define MODE_CMP 4
#define MODE_DUMP 5
#define MODE_RMTABLE 6
#define MODE_MKTABLE 7

static void *load_file(char const *path, long *pfsize)
{
        long fsize;
        void *fdata;
        FILE *file;

        if ((file = fopen(path, "rb")) == NULL) {
                perror(path);
                return NULL;
        }
        fseek(file, 0, SEEK_END);
```

```
        fsize = ftell(file);
        rewind(file);
        if ((fdata = malloc(fsize + 1)) == NULL) {
                perror("allocating file data");
                fclose(file);
                return NULL;
        }
        if (fread(fdata, 1, (size_t) fsize, file) != (size_t) fsize) {
                perror(path);
                free(fdata);
                fclose(file);
                return NULL;
        }
        fclose(file);
        ((char *) fdata)[fsize] = 0;
        *pfsize = fsize;

        return fdata;
}

static char **get_flist(char **ifiles, int infiles, char const *flpath,
                        int *pnfiles)
{
        int i, nalloc;
        long fsize;
        char *fdata, *tok;
        char **flist, **nflist;

        if (flpath == NULL) {
                nalloc = infiles + 1;
                if ((flist = (char **) malloc(nalloc * sizeof(char *))) == NULL)
                        return NULL;
                for (i = 0; i < infiles; i++)
                        flist[i] = strdup(ifiles[i]);
                *pnfiles = infiles;
                return flist;
        }
        if ((fdata = (char *) load_file(flpath, &fsize)) == NULL) {
                perror(flpath);
                return NULL;
        }
        nalloc = infiles + (int) (fsize / 16) + 1;
        if ((flist = (char **) malloc(nalloc * sizeof(char *))) == NULL) {
                free(fdata);
                return NULL;
        }
        for (i = 0; i < infiles; i++)
                flist[i] = strdup(ifiles[i]);
        for (tok = strtok(fdata, " 0); tok != NULL;
           tok = strtok(NULL, " 0)) {
                if (i >= nalloc) {
                        nalloc = nalloc * 2 + 32;
                        if ((nflist = (char **)
                             realloc(flist, nalloc * sizeof(char *))) == NULL) {
```

```
                    for (i--; i >= 0; i--)
                        free(flist[i]);
                    free(flist);
                    free(fdata);
                    return NULL;
                }
            }
            flist[i++] = strdup(tok);
        }
        free(fdata);
        *pnfiles = i;

        return flist;
}

static void free_flist(char **flist, int n)
{
        if (flist != NULL) {
            for (; n > 0; n--)
                free(flist[n - 1]);
        }
        free(flist);
}

int main(int ac, char **av)
{
        int i, error, nfiles, mode = MODE_PUT, journal = 0;
        unsigned int tblsize = 16000, tblid = 0;
        long fsize, rcount;
        void *fdata;
        char **files;
        char const *path = NULL, *flpath = NULL;
        struct smdbxi_factory *fac;
        struct smdbxi_file *file;
        struct smdb_dbfile_ctx *dfctx;
        struct smdb_db_config dbcfg;
        struct smdb_db_ckey ckey;
        struct smdb_db_cdata cdata;
        struct smdb_db_record rec;
        struct smdb_db_kenum ken;

        MZERO(dbcfg);
        dbcfg.blk_size = 1024;
        dbcfg.blk_count = 100 * 1024;
        dbcfg.cache_size = 64 * 1024;
        dbcfg.num_tables = 16;
        for (i = 1; i < ac; i++) {
            if (strcmp(av[i], "-f") == 0) {
                if (++i < ac)
                    path = av[i];
            } else if (strcmp(av[i], "-l") == 0) {
                if (++i < ac)
                    flpath = av[i];
            } else if (strcmp(av[i], "-b") == 0) {
```

```
                        if (++i < ac)
                                dbcfg.blk_size = strtoul(av[i], NULL, 0);
                } else if (strcmp(av[i], "-c") == 0) {
                        if (++i < ac)
                                dbcfg.blk_count = strtoul(av[i], NULL, 0);
                } else if (strcmp(av[i], "-s") == 0) {
                        if (++i < ac)
                                dbcfg.cache_size = strtoul(av[i], NULL, 0);
                } else if (strcmp(av[i], "-x") == 0) {
                        if (++i < ac)
                                tblsize = strtoul(av[i], NULL, 0);
                } else if (strcmp(av[i], "-T") == 0) {
                        if (++i < ac)
                                tblid = strtoul(av[i], NULL, 0);
                } else if (strcmp(av[i], "-g") == 0)
                        mode = MODE_GET;
                else if (strcmp(av[i], "-e") == 0)
                        mode = MODE_ERASE;
                else if (strcmp(av[i], "-m") == 0)
                        mode = MODE_CMP;
                else if (strcmp(av[i], "-d") == 0)
                        mode = MODE_DUMP;
                else if (strcmp(av[i], "-R") == 0)
                        mode = MODE_RMTABLE;
                else if (strcmp(av[i], "-M") == 0)
                        mode = MODE_MKTABLE;
                else if (strcmp(av[i], "-j") == 0)
                        journal = 1;
                else
                        break;
        }
        if (path == NULL)
                return 1;

        if ((fac = smdb_xif_factory()) == NULL)
                return 2;
        if ((file = smdb_xif_file(-1, 0, path, SMDBXI_FL_RWOPEN, 0)) == NULL) {
                if ((file = smdb_xif_file(-1, 0, path, SMDBXI_FL_CREATENEW,
                                                 0)) == NULL)
                        return 3;
                if (smdb_dbf_create(fac, file, &dbcfg, &dfctx) < 0)
                        return 4;
                if (smdb_dbf_create_table(dfctx, tblid, tblsize) < 0)
                        return 5;
        } else {
                if (smdb_dbf_open(fac, file, &dbcfg, &dfctx) < 0)
                        return 4;
        }

        if ((files = get_flist(&av[i], ac - i, flpath, &nfiles)) == NULL)
                return 6;

        fprintf(stdout, "Number of files: %d0, nfiles);
```

```
if (journal && smdb_dbf_begin(dfctx) < 0)
        return 7;

if (mode == MODE_GET) {
        for (i = 0; i < nfiles; i++) {
                ckey.data = files[i];
                ckey.size = strlen(files[i]);
                if (smdb_dbf_get(dfctx, tblid, &ckey, &rec, &ken) > 0) {

                        smdb_dbf_free_record(dfctx, &rec);
                } else {
                        fprintf(stderr, "Record not found: '%s'0, files[i]);
                }
        }
} else if (mode == MODE_ERASE) {
        for (i = 0; i < nfiles; i++) {
                ckey.data = files[i];
                ckey.size = strlen(files[i]);
                if (smdb_dbf_erase(dfctx, tblid, &ckey, NULL) > 0) {

                } else {
                        fprintf(stderr, "Record not found: '%s'0, files[i]);
                }
        }
} else if (mode == MODE_PUT) {
        for (i = 0; i < nfiles; i++) {
                if ((fdata = load_file(files[i], &fsize)) == NULL) {
                        free_flist(files, nfiles);
                        return 8;
                }

                ckey.data = files[i];
                ckey.size = strlen(files[i]);
                cdata.data = fdata;
                cdata.size = fsize;
                if (smdb_dbf_put(dfctx, tblid, &ckey, &cdata) < 0) {
                        fprintf(stderr, "DD insert failed: '%s'0, files[i]);
                        free_flist(files, nfiles);
                        return 9;
                }
                free(fdata);
        }
} else if (mode == MODE_CMP) {
        for (i = 0; i < nfiles; i++) {
                if ((fdata = load_file(files[i], &fsize)) == NULL) {
                        free_flist(files, nfiles);
                        return 8;
                }

                ckey.data = files[i];
                ckey.size = strlen(files[i]);
                if (smdb_dbf_get(dfctx, tblid, &ckey, &rec, &ken) > 0) {
                        if (rec.data.size != (unsigned long) fsize ||
                            memcmp(rec.data.data, fdata, fsize) != 0)
```

```
                        fprintf(stderr, "Record in DB differs: '%s'0,
                            files[i]);

                    smdb_dbf_free_record(dfctx, &rec);
                } else {
                    fprintf(stderr, "Record not found: '%s'0, files[i]);
                }
                free(fdata);
            }
        } else if (mode == MODE_DUMP) {
            rcount = 0;
            if ((error = smdb_dbf_first(dfctx, tblid, &rec, &ken)) > 0) {
                do {
                    fprintf(stdout, "KEY (%ld)='", rec.key.size);
                    fwrite(rec.key.data, 1, rec.key.size, stdout);
                    fprintf(stdout, "'0);

                    fprintf(stdout, "DATA (%ld)0, rec.data.size);

                    smdb_dbf_free_record(dfctx, &rec);
                    rcount++;
                } while ((error = smdb_dbf_next(dfctx, &rec, &ken)) > 0);
            }
            if (error < 0) {
                fprintf(stderr, "Record enumeration failed!0);
                free_flist(files, nfiles);
                return 9;
            }
            fprintf(stdout, "0ound %ld records.0, rcount);
        } else if (mode == MODE_RMTABLE) {
            if (smdb_dbf_free_table(dfctx, tblid) < 0) {
                fprintf(stderr, "Table remove failed!0);
                return 11;
            }
        } else if (mode == MODE_MKTABLE) {
            if (smdb_dbf_create_table(dfctx, tblid, tblsize) < 0) {
                fprintf(stderr, "Table create failed!0);
                return 11;
            }
        }

        free_flist(files, nfiles);
        if (journal && smdb_dbf_end(dfctx) < 0)
            return 12;

        smdb_dbf_free(dfctx);
        SMDBXI_RELEASE(file);
        SMDBXI_RELEASE(fac);

        return 0;
    }
```

LICENSE

AUTHOR

Developed by Davide Libenzi **< davidel@xmailserver.org >**.

AVAILABILITY

The latest version of **SMDB** can be found at :

http://www.xmailserver.org/smdb-lib.html

BUGS

There are no known bugs. Bug reports and comments to Davide Libenzi **< davidel@xmailserver.org >**.