- database library - User Manual -

A. GENERAL INFORMATION

1. Product Description

The database library provides a software layer for accessing data using ISAM (Indexed Sequential Access Method). It has the following features :

. Several database files can be accessed at the same time.

. Database files contain tables. A table is in fact a collection of fixed-size records. The structure of a record is defined by its fields that can have types such as signed/unsigned integer, byte, floating-point, character or varying length string.

. Indexes can be created on these tables. An index is defined by specifying a list of one or more fields of the record. Indexes are typically used to search records through specific fields. Indexes can be created and deleted anytime, even with the table containing data. Index key values are compressed to minimize the number of disk accesses when searching a record.

. Insert, Update, Delete and Retrieve operations are used to access the records of a table using any index.

. Transactions can be used to execute a sequence of several commands either completely or not at all.

. Concurrent access of database files by multiple processes is supported through automatic file locking.

. The integrity of the database is guaranteed against power failures or process crashes by an automatic recovery mecanism.

. A logging file can optionally be associated with each database file. It will store all database modifications. In case of a disk failure, the database file can be re-created from the last database backup and the logging file.

2. Specific Product Limits

```
Maximum size of a database file : 2^63 bytes
Maximum size of a logging file  : 2 GB
Maximum size of a record        : 65.500 bytes
Maximum size of an index key    : 116 bytes
```

B. DATABASE MANAGEMENT

1. Management Commands

```
  db_create_database()         db_delete_database()
  db_create_logging()          db_recover()
  db_set_logging_filename()    db_get_logging_filename()
```

2. Creating a Database

A new database is created by the following call :

```
db_create_database ("c:\\mybase.db");
```

Attention: this will delete any existing file having the same filename.

3. Deleting a Database

A database and its associated logging file (if any) are deleted by the following call :

```
db_delete_database ("c:\\mybase.db");
```

4. Database Logging

Database Files can become damaged or lost in the event of a harddisk head crash. To overcome this problem, a database file can be stored with redundancy on two separate disks :

. one disk contains the (usually large) database file,
. the other contains the (usually small) logging file.

All updates done on the database file are recorded in the logging file. If the database file is lost, it can be restored by using the last full database file backup and re-doing all operations that were recorded in the logging file.

A new database file with logging file is created by the following two calls :

```
db_create_database ("c:\\mybase.db");
db_create_logging ("c:\\mybase.db",
                   "d:\\mybase.log");
```

5. Start Logging

Database Logging can also be started (or re-started) later while the database already contains data. The user should then perform the following actions :

1) stop all processes that access the database (if any),
2) take a full backup of the database file, (for example on tape),
3) create a logging file, by using the call :

```
db_create_logging ("c:\\mybase.db",
                   "d:\\mybase.log");
```

4) restart all processes (if any).

Taking a full backup of the database file is necessary because the recovery procedure will check that the first transaction of the logging file comes after the last transaction of the database file.

6. The Backup Procedure

At regular intervals (for example when the logging file becomes rather large), the user must perform the following actions :

1) stop all processes that access the database (if any),
2) take a backup of the database file, (for example on tape),
3) create a new logging file, by using the call :

```
           db_create_logging ("c:\\mybase.db",
                              "d:\\mybase.log");

4) restart all processes (if any).
```

7. Recovery Procedure after Database Disk Crash

If the disk containing the database file has crashed, the user must perform the following actions :

1) buy and install a new disk ('c:'),
2) restore the database file taken during the last full backup,
3) re-execute all actions stored in the current logging file upon the database file, by using the following call :

```
           db_recover ("c:\\mybase.db",
                       "d:\\mybase.log");
```

4) restart all processes (if any).

The step 2) can be skipped if this is the first logging file and a backup was not necessary until now. The database file need then not exist and the function db_recover() will automatically create a new database file.

8. Recovery Procedure after Logging Disk Crash

If the logging file's disk has crashed, the user must perform the following actions :

1) take a full backup of the database file,
2) buy and install a new disk ('d:'),
3) create a logging file on the new disk, by using the call :

```
           db_create_logging ("c:\\mybase.db",
                              "d:\\mybase.log");
```

4) restart all processes (if any).

Taking a full backup of the database file is necessary because the recovery procedure will check that the first transaction of the logging file comes after the last transaction of the database file.

9. Changing the Logging File Filename

The logging file filename is stored within the database file. It can be necessary to change it if the user decides to put the logging file on another disk.
For example, if the user decides to move the logging file from disk 'd:' to disk 'e:', the following actions should be performed :

1) stop all processes that access the database (if any),
2) move the logging file from disk 'd:' to disk 'e:'
3) inform the database about the new logging file location by using the following call :

```
           db_set_logging_filename ("c:\\mybase.db",
                                    "e:\\mybase.log");
```

4) restart all processes (if any).

```
10. Switching Off Database Logging
----------------------------------
Database Logging can be switched off by specifying an empty
logging filename, as follows :

           db_set_logging_filename ("c:\\mybase.db",
                                    "");

Attention: Recovery will no longer be possible after a disk
           crash.


11. Obtaining the current logging filename
------------------------------------------
The current logging file filename can be obtained by the
following call :

      char logging_filename[MAX_LOGGING_FILENAME_LENGTH+1];
       ...
      db_get_logging_filename ("c:\\mybase.db",
                               logging_filename);

An empty filename indicates that logging is not active.



C. DATABASE SESSION
===================

1. Database Session Commands
----------------------------
The Management Commands described in the previous chapter all
had the database filename as parameter.

All following commands are different : they are used within
a database session and require a database handle (or db_handle)
as parameter.

Working in a session has two effects :
. all database modifications are stored in the logging file
  (if any),
. transactions can be used to execute completely or not at all
  a series of commands in a session.

2. Opening a session
--------------------
db_open_database() opens a database session. If successful,
it returns a db_handle to be used for all following commands.

3. Closing a session
--------------------
db_close_database() must be used to close again the session.



D. TABLE MANAGEMENT
===================

There are two ways to create a database table : manual and
automatic.

1. Manual Table Creation
------------------------
You have to initialize manually a table definition variable
```

that has the following structure :

```
struct FIELD_DEFINITION
{
  uint  offset;                /* between 0 and MAX_RECORD_SIZE  */
  uint  size;                  /* between 1 and MAX_RECORD_SIZE  */
  byte  type;                  /* see TYPE_xxx constants above   */
  char  name[MAX_FIELD_NAME_LENGTH];  /* field name  */
}

struct TABLE_DEFINITION
{
  uint             record_size;         /* between 1 and MAX_RECORD_SIZE  */
  uint             nb_fields;           /* between 1 and MAX_TABLE_FIELDS */
  FIELD_DEFINITION field[MAX_TABLE_FIELDS];
}
```

The table definition must match exactly with the 'C' record
structure, as demonstrated in the following example :

```
      packed struct PERSON_RECORD
      {
        char  name[32];
        short age;
        char  address[200];
      }

      PERSON_RECORD    person;
      TABLE_DEFINITION t;
      int              rc;

      t.record_size = PERSON_RECORD'size;
      t.nb_fields = 3;

      t.field[0].offset = 0;
      t.field[0].size   = person.name'size;
      t.field[0].type   = TYPE_STRING;
      strcpy (out t.field[0].name, "name");

      t.field[1].offset = t.field[0].offset + t.field[0].size;
      t.field[1].size   = person.age'size;
      t.field[1].type   = TYPE_INT;
      strcpy (out t.field[1].name, "age");

      t.field[2].offset = t.field[1].offset + t.field[1].size;
      t.field[2].size   = person.address'size;
      t.field[2].type   = TYPE_STRING;
      strcpy (out t.field[2].name, "address");

      rc = db_create_table (db, "PERSON", t);
      if (rc < 0)
        ...
```

The table definition variable has the following constraints :

```
offset / size
-------------
Record fields must be given by increasing 'offset' order.
Holes in the record are allowed, but fields must not overlap.
All fields must be within the 'record_size'.


name
----
Field names must be unique within the table. They must be
```

```
non-null strings. Upper and lower case letters are distinct.
Only letters, digits and underscores are allowed (no spaces).
A name must not begin with a digit.

type
----
The following types are allowed :

const uint TYPE_INT       = 0x00;
const uint TYPE_FLOAT     = 0x01;
const uint TYPE_UINT      = 0x02;
const uint TYPE_CHAR      = 0x03;
const uint TYPE_BYTE      = 0x04;
const uint TYPE_STRING    = 0x05;

Numeric types are either :
. binary signed (TYPE_INT),
. binary unsigned (TYPE_UINT), or
. floating-point (TYPE_FLOAT).

Character types are either :
. optionally null-terminated (TYPE_STRING), or
. not null-terminated (TYPE_CHAR).

Fields containing undefined data should use (TYPE_BYTE).



Distinction between TYPE_STRING and TYPE_CHAR
---------------------------------------------
It is important not to confuse the types TYPE_STRING and
TYPE_CHAR because they are not interchangeable !

Comparison of string fields stops at the first null character
whereas character fields are compared up to the last character.

This has consequences for the uniqueness of index key values
and when searching an exactly matching record.

If you wrongly use the type (TYPE_CHAR) for null-terminated
strings, subtle problems can occur :
. a search for an exactly matching record can fail without
  apparent reason,
. when inserting duplicate key values, duplicate key errors
  will not always occur.

Similarly, if the type (TYPE_STRING) is wrongly used :
. a search for an exactly matching record will possibly
  return the wrong record,
. duplicate key errors will occur without apparent reason.

Furthermore, note that the compression techniques used for
TYPE_STRING and TYPE_CHAR are slighly different :
. TYPE_STRING : the string is only stored until the first null
                character.
. TYPE_CHAR   : identical trailing characters are compressed,
                for example spaces or zeroes.

Finally, note that string fields can be used up to the
last character (a terminating null character is only needed
if the string length is smaller than the field size).


2. Automatic Table Creation
---------------------------
You have to initialize a table definition variable using
```

the constructor function :

```
       db_construct_table_definition ()
```

This function requires a table definition string, as
demonstrated in the following example :

```
      packed struct PERSON_RECORD
      {
        char  name[32];
        short age;
        char  address[200];
      }

      PERSON_RECORD    person;
      TABLE_DEFINITION t;

      rc = db_construct_table_definition
                 (out t,
                  "string32  name    " +
                  "int2      age     " +
                  "string200 address ");
      if (rc < 0)
        ...

      rc = db_create_table (db, "PERSON", t);
      if (rc < 0)
        ...
```

The constructor function translates the table definition string
into a table definition variable.

Table Definition String Format

Newline characters '\n' are allowed within the string as
separators.

The types 'int', 'float', 'uint', 'char', 'byte' and
'string' are allowed.

The number following the type always denotes the size of the
field (size 1 is the default if no number is specified).

Holes cannot be specified.

It is the programmer's responsability that the table definition
string matches the actual 'Safe-C' structure. Using the constructor
function is therefore less secure than filling the table
definition variable manually.

3. Disk Space Allocation

Records are always allocated using entire disk blocks.
To compute the number of disk blocks allocated per record,
use the following algorithm :

```
  if record_size <= 508
    nb_blocks := 1
  else
    nb_blocks := 1 + (record_size - 9) / 500
```

Remark that even a 10-byte data record occupies at least
one disk block (disk blocks of 512 bytes are used).

note: records are not compressed, but indexes are (see below).

4. Renaming Tables

The name of a table can be changed using the function :

      db_rename_table()

No other process must have an open session with this table
via db_open_table().

5. Deleting Tables

Tables can be deleted using the function :

      db_delete_table()

No other process must have an open session with this table
via db_open_table().

6. Opening A Table

Before data operations (insert/delete/update/retrieve) can be
performed on a table, it must be opened by calling :

      db_open_table()

If successful, this function returns a table_handle that must
be used for all following functions operating on the table.

This function requires again the table definition variable
to check that it matches with the file structure (this
prevents file structure mismatch errors when a program
was not recompiled after the table structure was changed).

Note that when a table is open, indexes cannot be created,
renamed or deleted and the table cannot be renamed or deleted.

7. Closing A Table

To release the table_handle, the following call is appropriate :

      db_close_table()

E. INDEX MANAGEMENT
===================

1. Index Management Commands
----
  db_construct_index_definition ()
  db_create_index()   db_rename_index()   db_delete_index()
  db_set_index()

2. Manual Index Creation

To create an index manually, you have to initialize an index
definition variable that has the following structure :

```
struct INDEX_PART_DEFINITION
{
  char name[MAX_FIELD_NAME_LENGTH];   /* field name */
}

struct INDEX_DEFINITION
{
  int                   nb_parts;      /* between 1 and MAX_KEY_PARTS */
  INDEX_PART_DEFINITION part[MAX_KEY_PARTS];
}
```

The following example creates an index called "PRIMARY" on
the fields (name + age) of the table "PERSON" :

```
      INDEX_DEFINITION i;

      i.nb_parts = 2;
      strcpy (out i.part[0].name, "name");
      strcpy (out i.part[1].name, "age");

      rc = db_create_index (db, "PERSON", "PRIMARY", i);
      if (rc < 0)
        ...
```

3. Automatic Index Creation

Another method to create an index is to use an index constructor,
as shown in the following example :

```
      INDEX_DEFINITION i;

      rc = db_construct_index_definition (out i, "name, age");
      if (rc < 0)
        ...

      rc = db_create_index (db, "PERSON", "PRIMARY", i);
      if (rc < 0)
        ...
```

4. Purpose of an Index

An index is defined by a list of one or several record fields.
The key of an index is formed by the catenation of the
corresponding fields.

An index has three basic purposes :
. SEARCHING : a record can be searched by its index field(s),
. ORDERING : records can be read sequentially in index order,
. INTEGRITY : it guarantees that all records have different key
              values.

When several fields are specified in an index, then searching
occurs first on the first field. It continues on the second
field in case of identical values for the first field,
and so on until the last field.

5. No Duplicate Keys Allowed

An index must never contain duplicate key values.

When defining an index, the user must make sure that the fields
used in the key always form a unique combination for each record.

If this is not the case, the user should add a field in the
record that will contain a unique identifier like a serial
number. This field should then be appended in the key.

An error occurs if an insert or update operation is trying to
create a duplicate key value.

6. Dynamic Indexes

Indexes can be created and deleted anytime, even if the table
contains data. However, the table must not have been opened
by any process via db_open_table().

7. Number of Indexes

To optimize performance, the user should follow these
guidelines :

- Tables that do a lot of insertions and deletions should have
  as few indexes as possible (generally only one).
  Be aware of the fact that these operations work with all
  indexes because they have to be up-to-date at all times.

  Update operations also enter in this case if the key fields
  change.

  For non-realtime applications, an index can be created when
  needed and deleted afterwards.

- If most operations are retrieves and updates, then the number
  of indexes is not relevant for performance.

8. No Index Hierarchy

Unlike other ISAM implementations, there are no primary or
secondary indexes : all indexes are equivalent. There is however
a rule that must be followed strictly :

  At least one index must exist when the table contains data.

This has two consequences :
. you must create at least one index before inserting records.
. you cannot delete the last remaining index if the table
  contains data.

9. Renaming an index

The name of an index can be changed using the function :

                db_rename_index ()

No other process must have an open session with this table
via db_open_table().

10. Deleting an index

An index can be deleted using the function :

                db_delete_index ()

No other process must have an open session with this table

via db_open_table().

11. Index Compression

Three types of compression are automatically applied on key values :

- leading bytes compression : leading bytes that are common for one key value and the next key value are only stored once. This feature is enabled for keys that are larger than 4 bytes.

- trailing bytes compression : this compression method is used separately on each field of the key. It consists of compressing identical trailing characters of the field (for example, characters fields that are padded with spaces). It is enabled for character type fields that have a size larger than 3 bytes.

- string compression : this compression method is used separately on all string type fields of the key. It consists of storing strings only until their first null character. It is enabled for string type fields that have a size larger than 3 bytes.

Some of these compression features are nevertheless disabled if the maximum key size is close to MAX_KEY_SIZE (116 bytes).

The user has no control over these features but can take advantage of them.

12. Index Key Size

The maximum key size (the sum of all key fields) is 116 bytes.

To optimize performance, the user should define keys that are as small as possible. It is important that as many keys as possible can be stored in a disk block.

Compression techniques (as described earlier) try to reduce the size of the actual key value. Note therefore that keys formed of large character or string fields need not be inefficient if the actual values can be compressed strongly.

F. QUERY
===

1. Query Commands
---
  db_query_table_name () db_query_table_definition ()
  db_query_index_name () db_query_index_definition ()

2. Query Table Information

The function db_query_table_name() can be used to obtain information about all the table names of the database.

The function db_query_table_definition() can be used to obtain the structure of a table. It uses a table definition variable (see db_create_table())

3. Query Index Information

The function db_query_table_index() can be used to obtain
information about all the index names of the table.

The function db_query_index_definition() can be used to
obtain the structure of an index. It uses an index definition
variable (see db_create_index())

4. Concurrent Access

The number of tables or the number of indexes can change
dynamically while querying the database, resulting in
incoherences between successive db_query_xx() calls.
To avoid this, the user should enclose all query statements
within a transaction when working in a multiprocess environment.

G. RECORD OPERATIONS
====================

db_insert() db_delete() db_update() db_retrieve()

1. Insertion

The function db_insert() is used to insert records.
The supplied record value must be completely initialized.
All indexes are automatically updated.

2. Deletion

The function db_delete() is used to delete records.

The function first searches the record using the specified index
and then deletes it.

The supplied record value must be partly initialized (at least
the fields that are part of the specified index).

All indexes are automatically updated.

3. Update

The function db_update() is used to update records.

The function first searches the record using the current index
(see db_set_index), and then updates all fields that are not
part of the index using the supplied record value.

The supplied record value must be completely initialized.

If the table has more than one index and the key value of
any of these other indexes is changed by the update, then
the corresponding index is automatically updated.

4. Retrieve

The function db_retrieve() is used to retrieve records.

The function searches the record using the specified index
and reads it into the supplied record parameter.

A retrieve_mode parameter further refines the search :

DB_FIRST
  Retrieve the first record using index order.

```
  The supplied record need not be initialized.

DB_LAST
  Retrieve the last record using index order.
  The supplied record need not be initialized.

DB_EQUAL
  Retrieve a record with a key matching the supplied record,
  using the specified index.
  The supplied record value must be partly initialized, at least
  the fields that are part of the specified index.

DB_SMALLER
  Retrieve a record that is just smaller than the supplied record,
  using the specified index.
  The supplied record value must be partly initialized, at least
  the fields that are part of the specified index.

DB_LARGER
  Retrieve a record that is just larger than the supplied record,
  using the specified index.
  The supplied record value must be partly initialized, at least
  the fields that are part of the specified index.

DB_EQUAL_OR_SMALLER
  Retrieve a record that is either equal or, if not found,
  a record that is just smaller than the supplied record,
  using the specified index.
  The supplied record value must be partly initialized, at least
  the fields that are part of the specified index.

DB_EQUAL_OR_LARGER
  Retrieve a record that is either equal or, if not found,
  a record that is just larger than the supplied record,
  using the specified index.
  The supplied record value must be partly initialized, at least
  the fields that are part of the specified index.

Unlike other ISAM implementations, there is no "current record
position" and consequently no "next" or "previous" retrieve
modes.

When searching a record using a character or string field,
please note this important difference :
. for a character field, all characters must match exactly,
. for a string field, comparison stops at the first null
  character.

5. Errors
---------
A "record not found" error occurs for delete/update/retrieve
  operations if no corresponding record could be found.

A "duplicate index" error occurs for insert operations if
  a record with this key already exists. It can also occur in
  update operations if the table has more than one index
  and the key value of any of these other indexes is changed.

A "locking" error occurs if the database is currently in use
  by other processes for a too long time. Locking errors
  cannot occur when executing commands within transactions,
  but they can occur when beginning a transaction.

A "no index" error occurs in insert operations if the table
  has no indexes. At least one index must be created before
  inserting records.
```

H. TRANSACTIONS

1. Transaction Commands

```
  db_begin_transaction()
  db_end_transaction()
  db_rollback_transaction()
```

2. Transaction Characteristics

Transactions are used to execute "atomic" command sequences,
i.e. the sequence is executed either entirely or not at all.

Furthermore, commands executed within transactions are guaranteed
never to be "disturbed" by other processes, which allows typical
bank transfer operations (decrease amount of record A and
increase amount of record B).

A transaction is started by calling db_begin_transaction(), and
it is terminated by calling either db_end_transaction() to
confirm it or db_rollback_transaction() to cancel it.

The transaction will also be cancelled if one of the following
events happens :

. db_close_database() is called without closing the transaction,
. db_end_transaction() returns an error, for example disk full,
. the program stops, or a power failure occurs.

The following example shows how transaction errors are handled
properly :

```
      int test_transaction (int db)
      {
         rc = db_begin_transaction (db);
         if (rc < 0)
           return rc;

         /* .. modify the database here ... */

         if (some error occured during the modifications)
         {
           rc = db_rollback_transaction (db);
           if (rc < 0)  /* either E_NO_BEGIN_TRANS or E_BAD_HANDLE */
             return rc;
           return (some error)
         }
         else    /* everything all right */
         {
           rc = db_end_transaction (db);
           if (rc < 0)
             return rc;
           return 0;
         }
      }
```

3. Commands that can/cannot be used in transactions

The following commands can be used in transactions :

```
 db_create_table    db_rename_table   db_delete_table
 db_create_index    db_rename_index   db_delete_index
 db_insert          db_delete         db_update
```

The following commands can also be used in transactions;
their rollback has no effect because they don't modify
the database :

```
 db_open_database               db_construct_table_definition
 db_close_database              db_construct_index_definition

 db_query_table_name            db_query_table_definition
 db_query_index_name            db_query_index_definition

 db_open_table                  db_set_index
 db_close_table                 db_retrieve
```

The following commands cannot be used in transactions :

```
 db_create_database             db_create_logging
 db_delete_database             db_recover
 db_set_logging_filename        db_get_logging_filename
```

Finally, these commands cannot be used in a transaction
because nested transactions are not allowed :

```
 db_begin_transaction  db_end_transaction  db_rollback_transaction
```

4. Efficiency Considerations

All database commands (i.e. db_insert) are executed "atomically",
which means that they are executed either completely or not at all.

To achieve this, the system opens an implicit transaction at the
start of each command, and closes it at the end of the command.

As a result, database commands are slowed down terribly because
the system opens and closes a transaction for each command,
which does not allow an efficient use of the disk's cache memory.

If you have a sequence of commands to execute, you might want to
enclose it between db_begin_transaction() and db_end_transaction() :
the system will then open and close a transaction only once instead
of doing it for each command, what will enormously improve the speed
of the application (data blocks can be kept in the cache instead
of being written physically to the disk at each command).

Enclosing a sequence of commands between db_begin_transaction() and
db_end_transaction() has however one disadvantage in the case of
multiple processes working concurrently on the database :
when a process executes a transaction, it acquires exclusive access
rights on the database and all other processes must wait until
the transaction terminates.

As a summary, we can recommend to have only short transactions
when working in a multiprocess environment and to have long
transactions when running a single process.

5. Thread Support

All database commands are thread-safe.

A transaction started by a thread blocks indefinitely any other database command until the transaction has terminated.

When a database file is used by several processes engaged in transactions, one should be aware that a locking error occurs at the beginning of the implicit or explicit transaction after a 10 second delay.