Transbase
Programming Interface TBX

Transaction Software GmbH
Willy-Brandt-Allee 2
D-81829 München
Germany
Phone: +49-89-62709-0
Fax: +49-89-62709-11
Email: info@transaction.de
http://www.transaction.de

Version 6.8.1.40
November 02, 2010

Contents

1 Overview **3**

1.1 Note to the TBX Interface Functions 3

1.2 Databases and User Authorization 4

1.3 Transactions 4

1.4 Dynamic Queries 6

1.4.1 One call Evaluation of Queries with TbxRun 7

1.4.2 Updatability of SELECT queries 7

1.4.3 Scrolling Cursor 8

1.4.4 Stored Queries with Parameters 8

1.5 DDL Statements 9

1.6 Control Information 9

2 Application Development **11**

2.1 Interfaces 11

2.2 Include-Files 11

2.2.1 Transbase Type-Coding 11

2.2.2 ANSI-C Support 12

2.2.3 C++ Support 12

2.3 Compilation of an Application Program on Unix Platforms 12

2.4 Application Development on PC Platforms 13

2.4.1 Process Architecture Topics 13

2.4.1.1 Scheduling 13

2.4.1.2 Abort-Callback 14

2.4.1.3 Unloading of DLLs 16
2.4.2 Extension of tbx-Interface 16
2.4.3 Application Development for MS/Windows 17
2.4.3.1 Choosing the Memory Model 17
2.4.3.2 Configuring the Stacksize 17
2.4.4 Application Development for WindowsNT, Win32S 18
2.4.4.1 Configuring the Stacksize 18
2.5 Description of Data Types 18
2.5.1 The Datatype Query_descr 19
2.5.2 The Components of Query_descr 21
2.5.3 The Datatype Result 25
2.5.4 The Components of Result 25
2.5.5 The Datatype Parameters for Stored Queries 26
2.6 Field Types 27
2.7 The Interface Routines of tbx 29
2.7.1 TbxConnect / CONTACTK / ACCEPTK - Connect to a database 30
2.7.2 Login 32
2.7.3 TbxDisconnect - Disconnect from a database 33
2.7.4 TbxBt - Begin a Transaction 34
2.7.5 TbxCt - Commit a Transaction 36
2.7.6 TbxAt - Abort a Transaction 37
2.7.7 TbxRun - Run a Statement 38
2.7.8 TbxDml/TbxCursorOpen - Activate a Cursor or Scrolling Cursor 42
2.7.9 TbxEval - Evaluate a non-scrolling cursor 44
2.7.10 TbxCursorFetch - Evaluate a scrolling cursor 46
2.7.11 TbxClose - Close a Query 48
2.7.12 TbxUpdPos - Perform an Update Positioned Statement . . 49
2.7.13 TbxDelPos - Perform a Delete Positioned Statement 51
2.7.14 TbxStore Statement - Store a Statement 52

2.7.15 TbxRunStored - Run a Stored Statement 54
2.7.16 TbxAddBatchParams - Add batch parameters to a stored INSERT/UPDATE/DELETE statement 56
2.7.17 TbxClearBatchParams - Clear pending batch parameters from the applications batch buffer 59
2.7.18 TbxRunStoredBatch - Run a stored INSERT/UPDATE/DELETE statement in batch mode 61
2.7.19 TbxOpenStored - Activate a Stored Statement 63
2.7.20 TbxCursorOpenStored - Open Scrolling Cursor on a Stored Statement 65
2.7.21 TbxUpdPosStored - Run a Stored Update Positioned Statement 67
2.7.22 TbxDelPosStored - Run a Stored Delete Positioned Statement 69
2.7.23 TbxDropStored - Drop a Stored Statement 71
2.7.24 TbxDropAllStored - Drop all Stored Statement 72
2.7.25 TbxSetSortOrder - Setting a User Sortorder 73
2.7.26 TbxGetSortOrder - Retrieving the Actual Sortorder 74
2.7.27 TbxTbmode - Run a Tbmode Statement 75
2.7.28 TbxSetDatDir - Define a Directory for Spool Files 76
2.7.29 TbxSetTimeOut Statement 77
2.7.30 TbxSetConsistency Statement 78
2.7.31 TbxGetTaState - Get Transaction State 79
2.7.32 TbxGetDbState - Get Database State 81
2.7.33 TbxSendEvent - Send an Application Event to the Database 82
2.7.34 TbxGetPlan - Get Evaluation Plan from Database 82

3 Tuples and Fields 85

3.1 Access to a Field 86
3.2 Number of Fields 86
3.3 Length of Tuple 87
3.4 Length of Field 87
3.5 End of Result 87

4 BLOBs at the TBX Programming Interface 89

4.1 Type and Type Encoding of BLOBs 89

4.2 Fetching BLOB Objects with TBX 90

4.2.1 TbxGetBlob 90

4.3 Inserting and Updating BLOB Objects with TBX 93

4.3.1 TbxMakeBlob 93

5 Signal Handling 95

5.1 Asynchronous Abort: TbxInterrupt 96

6 Error Codes and Messages 98

6.1 Hard and Soft Errors 99

A Transbase Interface Functions 100

B TBX Interface under MS/Windows 102

Chapter 1

Overview

The relational database system Transbase offers a relational data model together with the set oriented language TB/SQL and several other utilities.

For interactive applications, especially for ad-hoc queries, application programs like UFI or TBI are a sufficient and comfortable way to use Transbase.

For users who want to write their own application programs, a program linkage module called tbx is provided. It offers a interface between an application program and Transbase.

The interface tbx hides all communication details between the application program and the Transbase kernel. It is thus transparent to the application whether Transbase is linked together with the application or runs as a separate process on the same machine or even on a remote host. In any case, the application programmer sees the procedural interface as described by this manual.

The following sections describe syntax and semantics of the procedural interface provided by tbx.

1.1 Note to the TBX Interface Functions

The TBX application programming interface is the oldest Transbase interface. Due to the evolving standards in languages and compilers, most interface functions have been replaced by semantically equivalent functions which are more secure w.r.t. parameter type checking.

In detail, in the first TBX versions, one single interface function "tbx" existed which was called with a parameter determining the requested service. For example, tbx(DML,...,) compiled a query and tbx(EVAL,...) evaluated the query by delivering the first tuple.

Further parameters of the 2 services of course are different such that a reasonable parameter type checking was not possible.

The current version provides a separate function for each service, e.g. TbxDml(..) and TbxEval(..).

However, for compatibility, also the old concept still is supported. Thus, for each service, this manual offers 2 different variants of interface functions, one for existing applications and one recommended version for writing new applications.

1.2 Databases and User Authorization

Each application program can access one or more databases on local or remote hosts at a time. Before accessing a database the application must connect to each database. As a result of a successful `TbxConnect` operation the application program is delivered a database identifier to be used for subsequent identification of a database.

Databases are identified by a string of the form `<ldb>@<host>`. `<ldb>` is the logical name of a database, defined at creation time, `<host>` is the name of the host computer where the database resides. If the database is sitting on the local host, it may simply be identified by the string `<ldb>`.

A connection to a database is terminated by an explicit `TbxDisconnect` call specifying the database by its identifier.

After connecting to a database the application program has to login into the database, i.e. to perform the authorization procedure by specifying a user identification (the user name) and a password to `tbx`. Note that the authorization procedure has to be performed for each database connected to (i.e. one can connect as a different user to each database). After connection, but before authorization the application is not logged in and thus has no privileges to use the database.

By having a separate `TbxLogin` request, it is possible to change the user identification while still being connected to a particular database. Any unsuccessful `TbxLogin` request does not change the current user identification.

The application program can easily determine, in what state it is with respect to connections by calling `TbxGetDbState`. The states are `DB_DCONN` (no connection), `DB_CONN` (successfully connected), `DB_LOGGED` (successfully logged in), and `DB_KILLED` (the database kernel process has been killed).

1.3 Transactions

An application program can run transactions against databases. A transaction is the unit of consistency. Each transaction can either be committed or aborted by

the application program. A transaction may be distributed over some or all of the connected databases.

Committing a transaction protects all its effects against further crashes and makes any changes visible to other transactions (Note that intermediate states are not shown to other transactions). An abort undoes all effects of the current transaction and restores the database state as of the last commit point. This "all-or-nothing" property is guaranteed by Transbase even in case of a distributed transaction by a built-in 2-phase-commit protocol.

Whereas there are no restrictions in how transactions may be distributed over databases, there is the following restriction for multi-database applications:

For each application, a database may only participate in a single transaction at a time.

By this rule it is forbidden for an application to have two or more active transactions on the same database. In contrast, it is possible to have a set of independent transactions each referring to a different database.

If an application program does not commit its transaction, but simply leaves or exits the program, the transaction will be automatically aborted by Transbase, either immediately (if the premature exit is signalled via the communication link) or later when the database is connected again (by any application program).

A transaction consists of a sequence of DML and/or DDL statements. A statement addresses exactly one of the connected databases. However, more than one database can be addressed within a transaction by different statements.

As a rule, the application program has both to connect and login to the database addressed and to start a transaction before executing any DML or DDL statements on the database. The sequence of those two actions is arbitrary, i.e. the application may first connect and login and then start a transaction or equivalently first start a transaction and then connect and login. Note that by the latter mechanism it is thus possible to dynamically extend a running transaction over further databases determined at runtime of the transaction.

The application program can easily determine, in what state it is with respect to transactions by the call `TbxGetTaState`.

The transaction states and their meaning are as follows (defined in `tbx.h`):

`TA_NOT_ACTIVE` initial state; no transaction has been opened yet.

`TA_ACTIVE` a transaction has been opened and not yet ended (i.e. not yet committed or aborted).

`TA_COMMITTED` the last transaction run has been committed.

`TA_ABORTED` the last transaction run has been aborted. Note that committing a transaction can lead to an TA_ABORTED state e.g. if communication problems arise during the commit operation.

`TA_UNDEF` the state of the last transaction run cannot be determined (e.g. due to communication failures).

The state transitions are shown in the table State Transitions of Transactions.

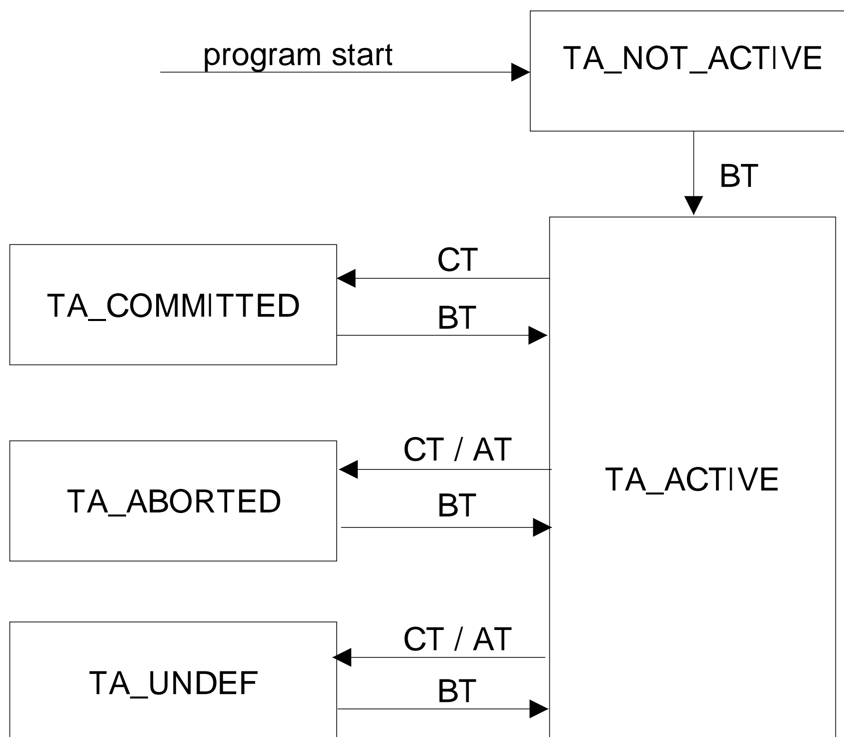

Figure 1.1: State Transitions of Transactions

1.4 Dynamic Queries

DML ("Data Manipulation Language") statements fall into two categories, namely retrieval queries (`SELECT ...`) and modification queries (`INSERT / UPDATE / DELETE ...`). The execution of a query consists of several phases shown below. This multi-phase-structure is reflected in corresponding kinds of `tbx`calls, namely a `TbxDml` call followed by a number of `TbxEval` calls followed by a `TbxClose` call.

A `TbxDml` call compiles a query given as a character string into an internal representation, called an operator tree. This operator tree then is optimized for

high-performance execution. After compilation the query is active i.e. ready for evaluation. A query description is returned to identify an active query. A query compiled by a `TbxDml` call is called a dynamic query (because it is immediately ready for evaluation and also completely disappears after evaluation).

The compiled query is evaluated by `TbxEval` calls. A modification query has to be evaluated by exactly one `TbxEval` call. A retrieval query, however, in general has a set of tuples as result and therefore the evaluation has to be performed by a number of `TbxEval` calls to get the result one tuple at a time. The normal way for an application program to fetch all tuples of a retrieval query is to loop through the result tuples until the "end of data" condition is signalled. However, it is possible to evaluate a query only partially or even not at all.

The active query has to be closed `TbxClose` in any case to make the query inactive. A query can be closed even if it is not evaluated or only partially evaluated (but it must be ative). For example to check for syntax errors, a query can be compiled and closed afterwards without evaluating it. After a dynamic query has been closed, it is not retained in any form in Transbase.

A transaction can be committed or aborted in any state of the query evaluation with the effect of `TbxClose`ing all active queries automatically at that time.

An application program may have active several queries at a time. So-called query descriptors are used to distinguish different queries. However, the number of concurrently active queries should be kept small since an active query may be expensive in terms of main memory requirements.

1.4.1 One call Evaluation of Queries with TbxRun

For convenience, the special sequence of a `TbxDml` call followed by exactly one `TbxEval` call followed by a `TbxClose` call can be abbreviated by the `TbxRun` call. The `TbxRun` call is of special interest for `UPDATE, INSERT or DELETE` queries and for `SELECT` queries which retrieve at most one tuple (e.g. an aggregate query or a tuple specified by its primary key) or when only the first result tuple is of interest. Note that using the `TbxRun` call instead of the sequence `TbxDml-TbxEval-TbxClose` not only saves program code but also executes much faster.

1.4.2 Updatability of SELECT queries

A `SELECT` query either is updatable or not updatable (see TB/SQL Reference Manual). The evaluation phase of an updatable `SELECT` query allows two additional calls, namely a `UPDATE` positioned (`TbxUpdPos` or `TbxUpdPosStored` call) and a `DELETE` positioned (`TbxDelPos` or `TbxDelPosStored` call). A call of these kinds only touches the current tuple. For the definition of "current" tuple, see TbxUpdPos 2.7.12 or TbxDelPos 2.7.13.

1.4.3 Scrolling Cursor

SELECT queries whioh have been opened with TbxDml also are called cursors. By definition, they are non-scrollable cursors, i.e. they only provide for sequential result fetching. On the contrary, scrolling cursors can be positioned at any position of its result. The call TbxCursorOpen creates scrolling cursors and TbxCursorFetch provides for absolute and relative positioning.

Note that scrolling cursors are slower than non-scrolling cursors because all result tuples are stored at kernel side.

1.4.4 Stored Queries with Parameters

A query can be compiled and stored with a `TbxStore` call. The `TbxStore` call returns a statement identifier which identifies the stored query. With a subsequent `TbxOpenStored` call, the stored query can be made active for subsequent evaluation (the evaluation itself via TbxEval calls and a terminating `TbxClose` call proceeds as with dynamic queries). Stored queries can be opened arbitrarily often within an application program without the overhead of compilation and optimization. A stored query is not bound to the transaction which stores it (in fact, it can be stored outside any transaction) but remains stored until the application disconnects (`TbxDisconnect`). After the `TbxDisconnect` of the application, the stored queries are not retained in any form in the database.

Stored queries can be parameterized: the supplied query contains formal parameters in the query string (see syntax below), and the `TbxOpenStored` call must supply actual parameters. The syntax of a formal parameter is ? or `#i(datatype)` where i is a natural number (greater than or equal to 1) which identifies the parameter, and datatype is one of the Transbase data types in SQL notation (e.g. `INTEGER, NUMERIC(6,2), CHAR(*)`).

Analogously to dynamic queries there is the possibility to run a stored query with one call provided it is an `UPDATE, INSERT or DELETE` query or a `SELECT` query where at most one result tuple is of interest. However, in contrast to a dynamic `TbxRun` call a special `TbxRunStored` call must be used for a stored query because the statement identifier of the stored query and (possibly) parameters must be supplied.

Moreover parameterized stored `UPDATE, INSERT or DELETE` queries can be executed in batch mode. Here a set of parameters is successively added to the batch by using TbxAddBatchParams. Batch parameters are buffered locally in the application until a certain size is reached (MAXTUPLESIZE) then the batch will be executed and results are retrieved. Afterwards the application can continue adding batch parameters. Finally TbxRunStoredBatch will transfer pending parameter batches to the database server and thereby complete the batch job. TbxClearBatchParams clears the local batch buffer.

A stored SELECT query can also be opened as a scrolling cursor by using the call TbxCursorOpenStored. This means that the property of scrollable need not be decided at STORE time but is dynamically choosable at activation time of the stored query.

`TbxUpdPos` and `TbxDelPos` calls may work on dynamic queries as well as on stored queries with no difference. It is also possible to store `UPDATE` positioned and `DELETE` positioned statements (the `UPDATE` positioned statement may have formal parameters). They are also stored with the `TbxStore` call. However, they must be run with the special calls `TbxUpdPosStored` and `TbxDelPosStored` resp. They may work on dynamic as well as on stored `SELECT` queries.

1.5 DDL Statements

In contrast to DML statements, DDL statements (”Data Definition Language”, see TB/SQL Reference Manual) are not translated, evaluated and closed phase by phase but executed directly in a single step. To execute a DDL statement, the TbxRun call is used.

Note that DDL statements are subject to the transaction concept as are all other statements, i.e. they can be aborted or committed by the application program at will. As with `TbxDml` statements, the DDL statement is passed as parameter of type string to Transbase.

Since DDL statements modify the data dictionary which is an exclusive resource of the database, the execution of a DDL statement may have to wait some period of time until accessing the data dictionary or even might lead to a deadlock. The first case is covered by a timeout mechanism, the second case results in an error-signalling return code.

1.6 Control Information

In this section an overview about control information is given. Detailed C definitions are given in the following sections.

The calls `TbxDml`, `TbxOpenStored`, `TbxRun`, `TbxRunStored` `TbxAddBatchParam` `TbxRunStoredBatch` return a query description. This query description is a structure which includes a query identifier for the active query (to distinguish between several queries active at a time) and a flag to distinguish the query type. In case of a `SELECT` query, also the number of fields of the result tuples, their types and names are delivered in the query description. Each name either is the simple field name in the `SELECT` clause or the explicitly specified name in an AS clause or (in case of expressions or aggregate functions) a unique dummy name.

Information delivered by an `TbxEval` call depends on the query type. A `TbxEval` call referring to a `SELECT` query returns either the next result tuple or a special endmarker tuple (called the empty tuple) when the result set is exhausted. Both events can easily be controlled by an additional `eod` variable which is set to the value `ONE_TUPLE` or `NO_TUPLE`, resp.

An TbxEval call refering to a modification query returns two integer values called `ntuples` and `tried`. The value `ntuples` describes the number of tuples touched by the query, i.e. the number of tuples inserted in case of an `INSERT` query, the number of tuples deleted in case of a `DELETE` query or the number of tuples updated in case of an `UPDATE` query.

In case of an `INSERT` query, an additional value `tried` delivers the number of tuples which were specified to be inserted by the statement. The value `tried` may be greater than `ntuples` since insertion of a tuple t into a relation which already contains a duplicate of t (i.e. a tuple identical on all fields) is not an error but simply has no effect. Thus, a difference between `tried` and `ntuples` tells the application program that not all tuples specified have been inserted since they had already been members of the relation.

Since a `TbxRun` call, `TbxRunStored`, `TbxAddBatchParam` and `TbxRunStoredBatch` call is a combination of several calls (as described in the sections above) it also returns a combination of the control information.

The same result information as for `INSERT` queries is available for the `SPOOL FROM` statement which reads data from external files into the database. The `SPOOL FROM` statement is executed by the `TbxRun` call. Remember that `TbxRun` calls deliver both a query description and a result.

A `SPOOL INTO` statement which writes data into an external file is also executed by the `TbxRun` call. Thus it delivers both a query description and a result where the component `ntuples` gives the number of tuples spooled into the file.

Note that for both `SPOOL` statements a data directory where the files are placed can be specified by the `TbxSetDatDir` call; if no directory is explicitly specified, data files will be placed in the directory which is the current directory of the application program at the moment of executing the `SPOOL` statement.

Chapter 2

Application Development

Prerequisite for application development is the Transbase Developer Toolkit. This is a licenced package which contains preprocessors, header files and libraries. Note that this package is not a software development environment. The latter can be chosen arbitrarily (e.g. Gnu CC, MS Visual C++, Borland C, Watcom, etc.).

2.1 Interfaces

Transbase applications can be developped for 2 different target interfaces:

C-Call-Interface: Function Calls like TbxDml or `tbx(<Service>, <par1>, ...)`

ESQL-Interface: Preprocessor Statements: `EXEC SQL ...`

2.2 Include-Files

Any application must include the file `tbx.h` header file.

2.2.1 Transbase Type-Coding

Transbase type coding used so far caused conflicts with code names in the file `WINDOWS.H` (e.g. code name `BOOL`). Therefore all Transbase type names are now prefixed with `TB__`. The same holds for the selector operators of the types `DATETIME` and `TIMESPAN`.

Compatibility to older programs can be achieved by inclusion of an adaptation include file named `TBCOMPAT.H` (after including `tbx.h`).

The ESQL-Preprocessor has been adapted to the new type coding.

2.2.2 ANSI-C Support

The include file `tbx.h` contains all prototypes for the TBX interface. This gives the necessary information to the prototype checks of ANSIC-C as far as Transbase calls are concerned. The prototype definitions are inside a switch

```
#if  USE_PROTOTYPES == 1
```

(which is the default) and can be switched off (if desired) by a definition

```
#define USE_PROTOTYPES 0
```

inside the application before `tbx.h` is included .

To make full use of ANSI-C checking, it is recommended to avoid the generic `TBX(...)` call and instead to use the specialized functions (see Transbase Interface Functions which have fixed parameter types and thus enable a complete parameter check by the compiler (also available under UNIX).

2.2.3 C++ Support

By definition of the macro `_cplusplus` in a C++ development environment all functions are declared as `extern "C"`.

2.3 Compilation of an Application Program on Unix Platforms

The tbx interface provides several functions, e.g. TbxConnect, TbxDml, some auxiliary routines, and a library of functions which handle the values of special field types like `Numeric` etc.

An include file named `tbx.h` has to be included in the application program (after `stdio.h`). It provides for type definitions (e.g. query description), constant definitions, and a number of macro definitions.

The library `tbx.a` as well as the include file `tbx.h` are located in the Transbase directory defined by the `TRANSBASE` environment variable.

The include statement necessary in each C program might look like one of the following:

```
#include "/usr/transbase/tbx.h"

#include "tbx.h"
```

The first possibility specifies an absolute pathname and thus is not invariant to location changes of the TRANSBASE directory. The second possibility specifies no directory at all and relies on the directory search path of the C preprocessor which can be set by the -I option on the command line, e.g.

```
cc -c appl.c -I$TRANSBASE
```

To link an application program and the tbx library (which is also located in the TRANSBASE directory and which itself needs the math library) the following command has to be issued:

```
cc appl.o $TRANSBASE/tbx.a -lm
```

Both command lines can be merged into one:

```
cc -I$TRANSBASE appl.c $TRANSBASE/tbx.a -lm
```

Another alternate method to compile a source program would be to define the following rule in the makefile:

```
cc -I$(TRANSBASE) $< $(TRANSBASE)/tbx.a -lm
```

2.4 Application Development on PC Platforms

For the development of applications for MS Windows, Windows NT the Microsoft Visual C++ Compiler or any compatible one is recommended.

2.4.1 Process Architecture Topics

2.4.1.1 Scheduling

With the exception of Windows NT, the Windows platforms use non-preemptive scheduling. This means that the programs are responsible to release the CPU to enable a task switch. Without appropriate additional mechanisms at the TBX interface and in the kernel the following inacceptable properties would hold:

- The processing of possibly very long actions (queries) would block the system and avoid (quasi)parallel other actions.
- Manual interruption of time consuming queries would not be possible.

To avoid these drawbacks, the TBX interface as well as the kernel periodically call a Callback function and thus transfer control to Windows to enable a scheduling. The callback situation is visualized in the diagram Transbase Architecture under Windows below.

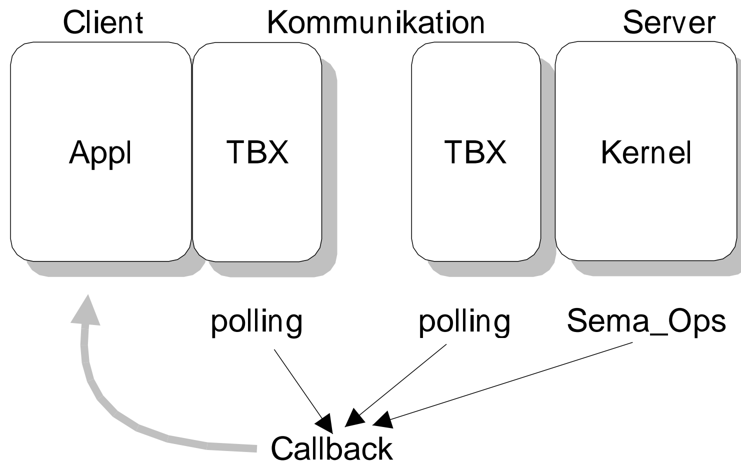

Figure 2.1: Transbase Architecture under Windows

Note:

- In a linked-in version, it is sufficient that only the kernel performs the callbacks.
- Problems arise for an application which accesses a local database (i.e. application and server on the same machine) under PC/NFS (`HOSTACCESSxx=tbnetnfs.dll, LINKED_IN=OFF`).

2.4.1.2 Abort-Callback

A Callback function can be defined with the following methods:

Default Callback: Unless an explicit callback function has been defined (see below) a standard callback function with the following functionality is activated:

CTRL + C triggers the abort of the transaction.

ALT + F4 is caught and transmitted to the application.

Userdefined Callback: By a special TBX interface function call: `TbxAbortCallback, tbxAbortCallback)` it is possible to install a callback function which is defined in the application program (see also 2.2.5.2).

For example, the callback function could present a (modeless) dialog box to abort a long transaction (see example below):

Example:

```
int far _pascal _export tbxCallback()
{
  MSG msg;
  State st;

  while(PeekMessage(&msg,0,0,0,PM_REMOVE))
    switch(msg.message)
    {
      case WM_ABORTTBX:         // user-defined message
         TbxInterrupt(&st);  // SIG_HANDLE mechanism, see. tbx-Manual
         // NO  break;

      default:
         TranslateMessage(&msg);
         DispatchMessage(&msg);
    }
  return 0;
}
```

Deactivate Callback: The call `TbxAbortCallback(NULL)` deactivates the explicitly defined or default callback, i.e. no task switching is performed inside a TBX or ESQL call!

Note:

- Disabling the task switch is unwise in a client/server scenario (not linked-in) where the client accesses a local database because the server would never be activated after a client's request,
- Disabling the task switch is also unwise if client/server communication is specified with DDE because data exchange across several DLLs needs activation of several tasks. This holds for network DDE as well as for Local-DDE!.
- Transbase database access via MS-Access/ODBC: With activated task switching, MS-Access would not await the completion of a DBX action but send the next action after reactivation. This causes an erroneous situation inside TBX because each call must come to its end before the next can be accepted (error message 'tbx already active'). As MS-Access is not configurable with respect to this behaviour, the Transbase ODBC driver does not permit task switching. By this fact it follows that *only non-local databases can be accessed via the ODBC driver.*

The implementation of Transbase application must consider the task switch problem. If callback is activated, the TBX calls must be synchronized to avoid the entrance of an already active TBX. If the blocking of the system during TBX calls are not a problem, then the disactivation of callback is the simplest solution.

2.4.1.3 Unloading of DLLs

Some of the Transbase DLLs are dynamically loaded at `TbxConnect` time and unloaded at `TbxDisconnect` time. If the `TbxDisconnect` call is not executed due to an uncontrolled end of the program, the DLLs remain loaded and the next `TbxConnect` call delivers an error (Transbase error message).

One possibility to circumvent this problem is to implement and bind a module for unloading Transbase DLLs (see also chapter Trouble Shooting).

Example:

```
    // unload of Transbase libraries:
{
      // turn off 'File not found' error box
  SetErrorMode(SEM_NOOPENFILEERRORBOX);
      // load Transbase dll
  HANDLE hLib=LoadLibrary("tbwsoc16.dll");
      // valid library handle?
  if(hLib >= HINSTANCE_ERROR)
      // while usage_count of library greater 0
  while(GetModuleUsage(hLib)>0)
      // free library, i.e. decrement usage_count
  FreeLibrary(hLib);
      // if usage_count equals 0, it will be freed
}
```

DLLs to be inloaded are:

- communication protocol DLL defined via variable `HOSTACCESS16` or `HOSTACCES32`
- kernel DLL `TBKER16.DLL` or `TBKER32.DLL` if used (i.e. `HOSTACCESSxx=TBBUNxx.DLL` or `LINKED_IN=ON`)

2.4.2 Extension of tbx-Interface

The tbx-interface provides a special function for Windows:

```
TbxAbortCallback(tbxCallback);
      // Pointer to Callback-function
PTbxCallback  tbxCallback;
```

A call of this function supports a callback function to tbx, which then is periodically called during the processing of tbx requests. This mechanism periodically transfers control to the application which is suitable to trigger task switching or asynchronous abort during long kernel actions. Note that the tbxCallback function must be exported in your applications module definition file.

2.4.3 Application Development for MS/Windows

To choose the 16 Bit target system make sure that the compiler macro `_WINDOWS` is defined. If you are using Microsoft C this is automatically done.

Transbase datatypes are compiled with 1-Byte packing. Therefore the `#pragma pack(1)` statement is defined in `tbx.h`. Optionally the `/Zp` option can be set during compilation.

Link your application against library tbx.lib which is part of the Transbase Developers Toolkit.

2.4.3.1 Choosing the Memory Model

Applications can be compiled and linked using various memory models (Tiny / Small / Compact / Medium / Large / Huge). Of course, prototypes and datatypes must be suitably defined (Near-/Far-Pointer) and number and size of program and data segments must fit the restrictions of the model. Independent of the properties induced by the memory models (not discussed here), Transbase DLLs cannot be multiply instantiated on all platforms, and parallel Transbase applications cannot run on one machine on all platforms (see Appendix).

2.4.3.2 Configuring the Stacksize

A Transbase application in version linked-in should have a stack of at least 50 KBytes. A fundamental problem is induced by the restriction of the stacksize on the 16 bit platforms.

In general, global variables, stack (function call stack and stack variables) and local heap are all placed in the standard data segment (DGROUP). The size of a segment is restricted on 64 Kbytes.

It is therefore important to arrange application data such that enough stack space remains for the kernel. To achieve this one can use the compiler option `/Gtn`

which places data with space requirement `>n` in other segments. In this context, the option `/Gx-` is of importance. Note that these options are only available with the large and huge memory models.

If you cannot configure the stacksize you have to access the databases using the DDE or the TCPIP communication libraries (See Transbase System and Installation Guide).

2.4.4 Application Development for WindowsNT, Win32S

To choose the 32 Bit target system make sure that the compiler macro `_WIN32` is defined. If you are using Microsoft C this is automatically done. Link your application against library `tbx32.lib` which is part of the Transbase Developers Toolkit for WindowsNT.

2.4.4.1 Configuring the Stacksize

Configuration of the stacksize is not available on the 32 Bit platforms. Please note that the stacksize limit for Win32S is 128K.

2.5 Description of Data Types

The include file `tbx.h` defines data types and constants which are used for parameters of the interface functions. It is recommended to use these type definitions in the application program in order to define variables which are type compatible to database types. As an example a type `Integer` is defined which is compatible to the database type INTEGER. Using the C-type int instead of the defined type Integer may result in reduced portability since int is not guaranteed to be a 4-byte-integer.

Note: Throughout the Transbase database system, for mnemonic reasons every data type name starts with an upper case followed by lower case letters whereas a variable or function name always is lowercase. Symbolic constants are always uppercase. For example, in the `tbx.h` file you will find lines like:

```
#define INTEGER 1

#define MAXSTRINGSIZE  4000

typedef char    String[MAXSTRINGSIZE];
typedef double  Double;
typedef Int4    Integer;
```

To transfer control information at the tbx interface the following three data types are used.

Query_descr is used to describe the property of a query. To each active query, a unique structure of type Query_descr is attached. For SELECT queries, there is also the `eod` variable provided and space for pointers to the result fields of one result tuple.

Result is used for result information such as the number of tuples inserted or changed by an INSERT or UPDATE statement, resp. Also one whole result tuple of a SELECT query can be supported if the splitting into single result fields is not desired. As a matter of fact, the `Result` variable is seldomly used.

Parameters is a structure used for delivering actual parameters to a stored query when it is OPENed.

2.5.1 The Datatype Query_descr

The datatype `Query_descr` is defined in `tbx.h` as follows:

```
typedef struct
{
   Id  query_id;        /* identifier of the open mql-query */
   short eod;
   short qtype;
   short updatable;     /* updatability of select queries */
   short qattr_no;      /* number of attrs in  result-tuple */
   short param_no;      /* number of params in  query */
   Field *field;    /* for each result-attribute */
   Par_descr *params;   /* for each parameter */
} Query_descr;          /* Description of an open query */

typedef struct
{
   char unnamed;    /*  BOOLesch: fieldname is generated by DB ? */
   short  fieldtype;    /* code of type */
   Tspec tspec;     /* fine type specification */
   char fieldname[MAXIDENTSIZE+1];
   char *fieldpointer;
} Field;

typedef struct {
```

```
    short  fieldtype;
    Tspec tspec;
} Par_descr;

typedef union {
    struct {
        char prec;  /* precision */
        char scale;  /* scale */
    } ps; /* for NUMERIC */
    struct {
        char lowf;  /* YY/MO/DD/HH/MI/SS/MS */
        char highf;  /* same */
    } lh;/* for DATETIME/TIMESPAN [highf:lowf] */
    short strprec;  /* for (BIN)CHAR/BITS */
} Tspec; /* fine type specifiers */
```

A structure of type `Query_descr` has to be defined by the application program for each TB/SQL statement. As a result of compilation, a structure of type `Query_descr` will be filled by Transbase and is then available to the application program. A pointer to a variable of type `Query_descr` has to be passed to Transbase for this purpose:

Example:

```
Query_descr qd;
char * statement = "SELECT * FROM suppliers";
TbxDml(dbid, taid, statement, & qd);
TbxEval(&qd, NULL);
```

Upon return from the TbxDml call, the query description qd contains a unique query identifier in component `query_id`, the type of the query in component `qtype` and additional information, depending on the type of the query. Note that the component `field` is allocated and freed on heap by Transbase and is only valid until the query is closed in case of a TbxDml call or in case of a `TbxRun`, `TbxRunStored`, `TbxAddBatchParam`, or `TbxRunStoredBatch` call until the next statement of one of these classes, a `TbxDml` or a `TbxClearBatchParams` call. For all subsequent calls corresponding to this query (e.g. `TbxEval` call, `TbxClose` call), the application program has to supply the pointer to qd as input parameter in order to identify the query. This is also shown in the example above (`TbxEval` call).

All components of the query description are accessible in reading mode by the application program. They are described in detail in the following section. Note that unpredictable errors may occur if the application changes some of the components of query description.

2.5.2 The Components of Query_descr

Note that all components are filled by Transbase. `query_id` is a number which is unique among all currently active queries in the application program.

eod may have one of the following 3 values: `NO_TUPLE`, `ONE_TUPLE`, `MORE_TUPLE`; It has a defined value after an `TbxEval`, `TbxRun`, or `TbxRunStored` call against a `SELECT` query.

In case of an `TbxEval` against a SELECT query, eod is set to `ONE_TUPLE` if a result tuple has been delivered and to `NO_TUPLE` if the result set is exhausted. In case of a `TbxRun` or `TbxRunStored` call against a SELECT query, eod is set to `NO_TUPLE`, `ONE_TUPLE`, `MORE_TUPLE` if the SELECT query produces no result tuple, 1 result tuple or more than 1 result tuple, resp.

In all other cases (e.g. `TbxEval` call against INSERT query or TbxDml call) eod is undefined.

qtype is a variable which contains the type of the compiled query. The value of `qtype` is one of the following constants (divided into classes) defined in `tbx.h`:

SEL class	SEL_TYPE SEL_FOR_UPD
DML class	INS_TYPE DEL_TYPE UPD_TYPE
DDL class	CREATETABLE DROPTABLE CREATEINDEX DROPINDEX CREATEVIEW DROPVIEW GRANT_ACCESS REVOKE_ACCESS GRANT_PRIVILEGE REVOKE_PRIVILEGE PASSWORD
SPOOL class	SPOOL_FILE SPOOL_RELATION
LOCK class	LOCK UNLOCK
LOAD class	LOAD_TABLE LOAD_INDEX LOAD_BY_QUERY LOAD_SWITCH_ON LOAD_SWITCH_OFF UNLOAD_TABLE UNLOAD_INDEX UNLOAD_ALL

The constant `SEL_FOR_UPD` describes a SELECT query where the FOR UPDATE clause has been specified; . Several macros for qtype are also defined in `tbx.h`, namely:

```
sel_class(qtype)
dml_class(qtype)
ddl_class(qd.qtype)
spool_class(qd.qtype)
lock_class(qd.qtype)
load_class(qd.qtype)
```

which return true if the query belongs to the corresponding class.

updatable is a flag which is defined if the query is a SEL class query (SELECT query); it is 0 if the query is not updatable and 1 if the query is updatable;

qattr_no is the number of fields of the tuples which are delivered by the query. qattr_no is defined for statements of class SEL.

param_no is the number of parameters which are contained in the query. If param_no > 0 then component params contains an array of length param_no where params[i] describes the SQL type of the i-th parameter (Par_descr).

field is an array on heap; it is dynamically allocated and freed by Transbase; do not malloc or free on this pointer.

field[i].fieldtype In case of a SEL class query, for each field i ($0 <= i < qattr_no$) `field[i].fieldtype` contains the data type of the result field. This data type is encoded as a symbolic constant defined in the include file tbx.h. The following table shows the names of the constants and the corresponding data type in SQL syntax. It is one of the following encoded values:

Symbolic constant of type	type in SQL syntax
TB__TINYINT	TINYINT
TB__SMALLINT	SMALLINT
TB__INTEGER	INTEGER
TB__BIGINT	BIGINT
TB__NUMERIC	NUMERIC(p,s)
TB__FLOAT	FLOAT
TB__DOUBLE	DOUBLE
TB__CHAR(p)	(VAR)CHAR)*/p)
TB__BINCHAR(p)	BINCHAR(*/p)
TB__BITSS	BITS(p/*)
TB__BITSS2	BITS2(p/*)
TB__BOOL	BOOL
TB__DATETIME	DATETIME all ranges
TB__TIMESPAN	TIMESPAN all ranges
TB__BLOB	Blob

Table 2.1: Types in Query Description

field[i].tspec is a finer description of the above described field[i].fieldtype. It is only defined if the type is one of the following:

TB__NUMERIC	the precision and scale
TB__CHAR	p for CHAR(p), 0 for CHAR(*)
TB__BINCHAR	analogous
TB__BITSS	analogous, strprec in bits
TB__BITSS2	analogous, strprec in bits
TB__DATETIME	range: symbolic constants YY to MS
TB__TIMESPAN	same

field[i].unnamed is a Boolean value describing whether the i-th component of the SELECT list is a named field (simple field or contains AS clause) or is a unnamed field (in the latter case, component fieldname contains a unique system generated name).

field[i].fieldname contains the name of the i-th field in case of a SEL class query if the i-th field is named otherwise it contains the symbolic name `Column_i`

field[i].fieldpointer points to the i-th field value of the delivered tuple after an `TbxEval` call or `TbxRun` call or `TbxRunStored` call for a SEL class query where a tuple has been evaluated (`eod != NO_TUPLE`). If the i-th field value is the SQL NULL value, the entry is set to the language C NULL pointer. If the call has not delivered a tuple (result exhausted) all entries have undefined values. Note that the type of the pointer is (char*) so it must be casted to the appropriate field type before dereferencing it (see example in 2.7.9).

params[i] Describes the type of the i-th parameter in the query. If the ? notation has been used for the parameter, then its type has been derived by the kernel according to derivation rules for the parameter's environment (e.g. " WHERE field = ?" attaches the type of ¡field¿ to the parameter).

Note that this parameter type need not be identical to the type of the actually supplied parameter value parameter type but need only be compatible to it.

Note: The component `field` points to an array which is allocated on heap by the runtime system of Transbase. In case of a `TbxDml` call or `TbxOpenStored` call the life time of this array is until the `TbxClose` call. In case of a `TbxRun`, `TbxRunStored`, `TbxAddBatchParams`, or `TbxRunStoredBatch` its life time is until the next statement of one of these classes, a `TbxDml` or a `TbxClearBatchParams` call.

Note: Unfortunately the names of the macros sel_class and dml_class are inconsistent with the use of the term `DML` throughout this and all Transbase Manuals. Note that whenever the term `DML` is used, SELECT queries as well as INSERT, UPDATE, DELETE queries are meant, i.e. subsumed under `DML`.

2.5.3 The Datatype Result

The datatype Result is defined in `tbx.h` as follows:

```
typedef struct {
    short qtype;
    short eod;
    union {
        char tuple [MAXTUPLESIZE];
        Count_result count;
    } _var;
} Result;

typedef struct {
    Int4  ntuples;
    Int4  tried;
} Count_result;
```

A structure of type Result may be passed by the application program for `TbxEval`, `TbxRun`, `TbxRunStored`, `TbxAddBatchParam`, `TbxRunStoredBatch` calls. The result variable is passed by pointer. Alternatively, a NULL value can be passed (see note at the end of the next section).

Example:

```
Query_descr qd;
char* statement = "SELECT * FROM suppliers";
Result res;

TbxDml( statement, & qd);
TbxEval( & qd, & res); /*** or alternatively:
TbxEval( & qd, NULL);  ***/
```

The components of Result are shown in detail in the following section.

2.5.4 The Components of Result

Note that all components are filled by Transbase.

qtype has the same meaning and the same value as `qtype` in `Query_descr`.

eod has the same meaning and the same value as `eod` in `Query_descr`.

res._var.tuple is a character array which contains the actual result tuple after an `TbxEval` or `TbxRun` or `TbxRunStored` call against a SELECT query. The tuple is delivered in its internal form with a header and a list of tuple fields. A number of macros are available to access tuple fields, the number or size of fields and the size of tuple as a whole. These macros are described in Chapter "Tuples and Fields".

res._var.count.ntuples is the number of tuples touched by a modifying DML (INSERT,UPDATE,DELETE) or a SPOOL statement.

res._var.count.tried is the number (of type Int4) of tuples presented for a modifying DML or a SPOOL statement. res._var.count.tried and res._var.count.ntuples are identical in case of UPDATE, DELETE and SPOOL INTO statements.

In case of an INSERT or a SPOOL FROM statement, the (non-negative) difference between res._var.count.tried and res._var.count.ntuples is the number of duplicates which were simply ignored by the insertion process. A tuple is considered a duplicate if exactly the same tuple (with exactly the same field values) is already contained in the particular table.

Note: For the evaluation of SELECT queries via a loop of `TbxEval` calls it is important to note that each result tuple is copied into the `Result` structure if the application supplies a pointer to a `Result` structure. However, it is very rare that an application needs the whole tuple in its internal form as described in Chapter "Tuples and Fields". In most cases the application wants to directly extract the field values. In this case it is useful to supply a NULL pointer and to use the fieldpointers of the structure `Query_descr` (which are set in any case) to read the field values. This results in better performance because then the copy process into `Result` is saved.

2.5.5 The Datatype Parameters for Stored Queries

The datatype Parameter is defined in `tbx.h` as follows:

```
typedef struct {
    short param_no; /* number of params */
    Param *param;   /* array of Param   */
}  Parameters;

typedef struct {
    short type;     /* type of   param */
    char  *value;   /* pointer to  value */
} Param;
```

The type Parameters is used to support actual parameters to the evaluation of a stored parameterized query. It is used in the `TbxOpenStored`, `TbxRunStored`, `TbxUpdPosStored`, `TbxAddBatchParam` and `TbxRunStoredBatch` call.

The component `param_no` gives the number of parameters, i.e. the number of elements in field `param`.

The array element `param[i]` refers to the i-th formal parameter of the corresponding query. Note that the numbering of the formal parameters `#1`, `#2`, ... in the SELECT query starts with 1, thus `param[0]` refers to the formal parameter `#1` and so on.

The component `param[i].type` is the encoding of the data type of the parameter. The same symbolic constants for type encoding are used as those in `Query_descr` (see The Datatype Query_descr.

The component `param[i].value` is a (char*) pointer to the parameter value. To provide space for the parameter value one can e.g. use C variables of the corresponding type (see type correspondences in Chapter "Field Types") or dynamic heap memory. To supply a SQL NULL value as parameter, one sets param[i].value to the C NULL pointer. A complete example for a parameterized query is given in the description of the `TbxOpenStored` call.

2.6 Field Types

To process field values, e.g. to store field values of a tuple into application program variables or to support actual parameters to a parameterized query, the following type mapping scheme is of importance.

All the above listed types in the middle column which are used as variable types in the application are defined in `tbx.h`. Examples of variable definitions in the application program are:

```
Integer i;
Double d;
Numeric n;
String s;
```

Fields of type CHAR(p) and CHAR(*) are represented within the tuple according to C conventions, i.e. they are ended with a NULL(0x0) byte. Therefore standard C-library functions such as 'strcpy', 'strcmp' etc. can be used to access string fields.

Fields of type BINCHAR(p), BINCHAR(*) are represented by a structure named Binchar (defined in `tbx.h`), which have a length component and an array of characters (inside the tuple this array really is of length p or variable sized, resp.). No

field type in DDL	variable type in application	corresponding type in C
TINYINT	Tinyint	char
SMALLINT	Smallint	short
INTEGER	Integer	Int4
BIGINT	Bigint	Int8
NUMERIC(p,s)	Numeric	- structure -
FLOAT	Float	float
DOUBLE	Double	double
CHAR(p)	char[p+1]	char[p+1]
CHAR(*)	String	char[MAXSTRINGSIZE]
BINCHAR(p)	Binchar	- structure -
BINCHAR(*)	Binchar	- structure -
BITS(p/*)	Bits	- structure -
BITS2(p/*)	Bits	- structure -
BOOL	Bool	char
DATETIME	Datetime	- structure -
TIMESPAN	Timespan	- structure -
BLOB	Blob	- structure -

Table 2.2: Datatype Mapping

NULL byte is used to mark the end because such a byte may also occur inside the data. By use of the length field also standard C-library functions such as 'strncpy', 'strncmp' etc. can be used to access those fields.

Fields of type BITS(p), BITS(*), BITS2(p), BITS2(*) are represented by a structure named Bits (defined in `tbx.h`), which have a length component and an array of characters. Furthermore, macros BITSARR and BITSLEN (and BITS2ARR and BITS2LEN for the type Bits2) are defined which serve to access the 2 components. Note that BITSLEN and BITS2LEN describes the length of the field value in bits not in byte.

For the field types NUMERIC, DATETIME and TIMESPAN more complex structures are used. A set of functions is supplied in the library `tbx.a`. See the Chapters "Routines for Numerics" and "DATETIME and TIMESPAN Conversion Routines".

For the type BLOB see BLOBs at the TBX Programming Interface.

Note: Do not confuse the DDL type notations of Transbase as noted in the above table with the symbolic C constants for types defined in `tbx.h`. The latter are used as encodings of types in the structure `Query_descr` and Parameters.

2.7 The Interface Routines of tbx

An overview of all interface functions is given in the table.

All interface functions return an error code of type int. An error code 0 means that no error has ocurred. Error codes are defined in the file `tberror.h` as constants with symbolic names, as e.g.

```
#define NO_ERROR          0
#define NO_TA_ACTIVE   9933
```

A typical call sequence for tbx would be (see also Error Codes and Messages).

```
int rc;
if (rc = TbxConnect(...)) {
    printf ("Error %d occurred: %s\n", rc, tb_errtxt);
    exit (1);
}
```

The following paragraphs discuss each call to tbx in full detail, by listing the input parameters, result parameters, call syntax and the effect of the call. Note that all parameters used by tbx have to be declared by the application program, i.e. tbx itself does not provide space for result parameters!

For some calls to tbx, a non exhaustive list of the most typical error return codes is given. For special error treatment see also (see also Error Codes and Messages).

2.7.1 TbxConnect / CONTACTK / ACCEPTK - Connect to a database

Input Parameter:

```
char * dbname;
```

Result Parameter:

```
Id dbid;
```

Call Syntax:

```
tbx (CONNECT, dbname, & dbid);
TbxConnect( dbname, & dbid);
tbx (CONTACTK, dbname, & dbid);
tbx (ACCEPTK, dbid);
```

Effect: Establishes a connection to the database specified by its logical name (given as input parameter "dbname"). This can be done either one-phase by `TbxConnect` or two-phase by the calls CONTACTK and ACCEPTK.

If successful, the result parameter `dbid` contains a valid database identifier in the range `[0 .. MAX_DB]`. This identifier has to be used in subsequent calls refering to this database.

An application program can connect to more than a database at a time. Connecting to the same database more than once is considered to be idempotent, i.e. a second `TbxConnect` call returns the same dbid that has been returned by the first `TbxConnect` call for the same database (assuming that no `TbxDisconnect` call interfered). The maximum number of databases an application can connect to at a time, is contained in `tbx.h` as a compile time constant.

A logical dbname is a character string consisting of a local dbname and optionally a host name, separated by the ampersand character `@`. If no host name is given, the database is assumed to reside on the local host. If a remote host is given, `TbxConnect` tries to address the host in order to establish a remote connection to the required database.

Instead of issueing a `TbxConnect` call the application may parallelize this (sometimes time-consuming) operation into two actions `CONTACTK` and `ACCEPTK`. `CONTACTK` only initiates the `TbxConnect` request but does not wait until the `TbxConnect` has succeeded. A later `ACCEPTK` operation waits for the `TbxConnect` operation to be completed. The `CONTACTK` and the `ACCEPTK` call deliver the same error codes that a one-phase `TbxConnect` call delivers.

By this asynchronous `TbxConnect` feature, applications have the possibility to do some useful work in parallel, e.g. to prompt the user for his login and password, or to initialize their own data structures.

Errors:

```
UNKNOWN_DATABASE
SERVER_NOT_ACTIVE
```

Example:

```
char * db = "sample@server";
Id dbid;
if(res = TbxConnect(db, & dbid)) error(res);
```

Example:

```
char * db = "sample@server";
Id dbid;
if ( res = tbx (CONTACTK, db, & dbid)) error(res);
    /* do some other things */
if ( res = tbx (ACCEPTK, dbid)) error(res);
```

2.7.2 Login

Input Parameter:

```
Id dbid;
char * username;
char * passwd;
```

Result Parameter: None

Call Syntax:

```
tbx (LOGIN, dbid, username, passwd);
TbxLogin (dbid, username, passwd);
```

Effect: The `TbxLogin` request checks the user authorization. Prior to `TbxLogin`, a successful `TbxConnect` request must have been issued.

If the user given by `username` is not registered as user in the database given by `dbid` or if the given `passwd` is not valid for this user, the error `LOGIN_FAILED` will be returned.

Errors:

```
LOGIN_FAILED
ILL_DB_ID
```

Example:

```
Id dbid, dbstate;
char pw [20];
char un [20];
getname(un);
getpasswd(pw);
if (TbxLogin(dbid, un, pw) == LOGIN_FAILED) {
  printf("No User or wrong password\n");
  TbxDisconnect(dbid, & dbstate);
  exit (1);
}
```

2.7.3 TbxDisconnect - Disconnect from a database

Input Parameter:

```
Id dbid;
```

Result Parameter:

```
State dbstate;
```

Call Syntax:

```
tbx (DISCONNECT, dbid, &dbstate);
TbxDisconnect (dbid, &dbstate);
```

Effect: Disconnects the calling application program from a database it had been connected to. If an invalid dbid is given, an error code will be returned. If successful, the corresponding dbid is no longer valid.

Error occurs if a transaction is active on the database, i.e. has made at least a `TbxDml` or `TbxRun` call against the database and has neither been committed nor aborted.

After the `TbxDisconnect` call, `dbstate` contains one of the values also returned by the `TbxGetDbState` call.

Errors:

```
ILL_DB_ID
TA_OPEN
```

Example:

```
Id dbid;
State dbstate;
...
TbxDisconnect(dbid, & dbstate);
```

2.7.4 TbxBt - Begin a Transaction

Input Parameter: None

Result Parameter:

```
Id taid;
```

Call Syntax:

```
tbx (BT, & taid);
TbxBt (& taid);
```

Effect: Starts a transaction. If successful, a valid transaction identifier is returned in result parameter `taid`. This transaction identifier has to be used to identify the transaction in further calls.

Note that the transaction is not database specific, i.e. it automatically will be extended to all databases which participate in the transaction, i.e. which receive `TbxDml` or `TbxRun` calls with that transaction identifier. Consequently, it is not even necessary for the application to be connected to one or more databases before issueing the `TbxBt` call.

An application program is restricted to have only one open transaction per database at a time, because two parallel transactions on the same database might deadlock each other if they are driven by one and the same application. This restriction is not checked statically (i.e. it is permitted to issue two subsequent `TbxBt` calls), but is checked dynamically, e.g. when issueing two `TbxRun` calls on the same database, but within two different transactions. See examples 2 and 3 below.

The maximum number of open transactions per application program is given by the constant `MAX_TA` contained in `tbx.h`.

Errors:

```
TOO_MANY_TAS
TA_OPEN
```

Example:

```
Id taid;
...
TbxBt, &taid);
TbxConnect(...);
TbxConnect(...);
...
```

Example:

```
/* Two independent transactions on two databases */
Id taid1, taid2, dbid1, dbid2;
...
TbxBt(&taid1);
TbxBt(&taid2);
TbxConnect(dbname1, &dbid1);
TbxConnect(dbname2, &dbid2);
 ...  /* LOGIN on both databases .. */
TbxRun(dbid1, taid1, statement, ...);
TbxRun(dbid2, taid2, statement, ...);
...
```

Example:

```
    /* Two transactions on the same database - ILLEGAL!! */
Id taid1, taid2, dbid;
...
TbxBt(&taid1);
TbxBt(&taid2);
TbxConnect(dbname, &dbid);
...  /* LOGIN on both databases .. */
TbxRun(dbid, taid1, statement, ...);
TbxRun(dbid, taid2, statement, ...);
     /* This last TbxRun call returns the error TA_OPEN */
...
```

2.7.5 TbxCt - Commit a Transaction

Input Parameter:

```
Id taid;
```

Result Parameter:

```
State tastate;
```

Call Syntax:

```
tbx (CT, taid, & tastate);
TbxCt (taid, & tastate);
```

Effect: Commits a transaction; all its updates are saved on disk. The commit call is automatically forwarded to all databases participating in this transaction. If the transaction has done modifications on more than a single database, a distributed 2-phase commit will automatically be performed.

The result parameter tastate will be set to one of the following values:

TA_COMMITTED , iff the transaction has been committed;

TA_ABORTED , iff the transaction has been aborted on all sites (e.g. due a system error on one of the participating nodes);

TA_UNDEF , iff the state of the transaction cannot be determined by tbx at the moment, e.g. due to communication problems.

Errors occur if taid is no valid transaction identifier.

Errors:

```
ILL_TA_ID
```

2.7.6 TbxAt - Abort a Transaction

Input Parameter:

```
Id taid;
```

Result Parameter:

```
State tastate;
```

Call Syntax:

```
tbx (AT, taid, & tastate);
TbxAt (taid, & tastate);
```

Effect: Aborts a transaction; all its updates are undone on disk. The abort call is automatically forwarded to all databases participating in this transaction.

The result parameter tastate will be set to the value:

TA_ABORTED , iff the transaction has been aborted on all sites.

TA_UNDEF , iff the state of the transaction cannot be determined by tbx at the moment. Although `TA_UNDEF` is returned (instead of `TA_ABORTED`) the application can be sure that the transaction specified will be aborted in any case, but perhaps after some delay.

Error occurs if taid is no valid transaction identifier.

Errors:

```
ILL_TA_ID
```

2.7.7 TbxRun - Run a Statement

Input Parameter:

```
Id dbid;
Id taid;
char * statement;
```

Result Parameter:

```
Query_descr qd;
Result res;
```

Call Syntax:

```
tbx (RUN, dbid, taid, statement, & qd, & res);
TbxRun (dbid, taid, statement, & qd, & res);
```

or

```
tbx (RUN, dbid, taid, statement, & qd, NULL);
TbxRun (dbid, taid, statement, & qd, NULL);
```

Effect: The statement given by the input parameter `statement` (as a character string) is executed. There is no restriction on the length of the `statement`.

The statement can be a SELECT, INSERT, UPDATE, DELETE, CALL, DDL, SPOOL, LOCK or LOAD statement. In an UPDATE or DELETE Statement there must no CURRENT clause be specified.

Note that for SELECT statements only the first tuple of the query result is evaluated. Since the query automatically is closed, no further tuples can be fetched, i.e. no further `TbxEval` call could be applied to `qd`.

UPDATE, INSERT and DELETE statements also can be evaluated by the sequence TbxDml2.7.8, TbxEval and TbxClose. TbxRun, however, is faster because it saves some process communication.

The two result parameters `qd` and `res` contain all the information that would be returned if the statement were performed by the sequence TbxDml, TbxEval and TbxClose (see 2.7.8). If the `res` parameter is set to NULL upon calling `TbxRun`, no result is delivered in `res` (e.g. a GRANT statement would not produce any meaningful result tuple).

Note: The heap array `field` as well as its fieldpointers to the delivered tuple remain valid only until the next `TbxRun` call.

The `TbxRun` call is well-suited for SELECT queries that retrieve exactly one tuple, e.g. a tuple which is fully specified by its primary key values or for aggregate functions MAX, MIN, SUM or AVG. In any case, the variable `eod` in `qd` and `res` is set to the value `NO_TUPLE`, `ONE_TUPLE`, `MORE_TUPLE` if the SELECT query evaluates no tuple, exactly 1 tuple or would evaluate more than 1 tuples, resp. (note that also in the last case only the first is delivered).

Note that DDL statements, SPOOL statements, LOCK statements and LOAD statements can be executed by the `TbxRun` call only. In case of these statements, the result parameters are sparingly used (see below). Various errors are returned as error codes, e.g. permission errors. Actually only `qtype` in both `qd` and `res` will be set appropriately.

In case of a SPOOL FROM statement which reads data from an external file into a database relation, the result parameter `res` contains information on the number of tuples inserted:

`res._var.count.ntuples` gives the number of tuples inserted whereas `res._var.count.tried` gives the number of tuples in the external file (see the description of `INSERT`, also). In case of a SPOOL INTO statement, `res._var.count.ntuples` and `res._var.count.tried` are set equal to the number of tuples written into the external file.

In case of LOCK statements only a minimal query descriptor and a minimal result output parameter are returned. In this case the error code returned signals if all locks have been granted (`NO_ERROR`) or if a timeout or a deadlock has occurred. For the syntax of the LOCK statement see TB/SQL reference manual. Note that since the LOCK statement is passed to the TB/SQL compiler, various compiler error codes can be returned.

The UNLOCK statement is restricted to read locks which are not being used in a currently active query. Otherwise the error code `OBJ_ACTIVE` will be returned. Note that releasing read locks before the end of a transaction results in a lower read consistency. For a more detailed discussion of locks see the Transbase System Guide.

Errors:

```
ILL_DB_ID
ILL_TA_ID
various DML syntax error codes
```

Example:

```
Id dbid, taid;
Query_descr qd;
...
TbxRun(dbid, taid,
    "select count(*) from suppliers",
    & qd, NULL);
printf("%ld suppliers",
    * (Integer*) qd.field[0].fieldpointer);
```

Example:

```
Id dbid, taid;
Query_descr qd;
Result res;
...
TbxRun(dbid, taid, "unlock suppliers", & qd, NULL);
```

Example:

```
Id dbid, taid;
Query_descr qd;
Result res;
...
TbxRun(dbid, taid,
    "delete from suppliers", & qd, & res);
printf ("%ld tuples deleted\n",
    res._var.count.ntuples);
```

Example:

```
Id dbid, taid;
Query_descr qd;
Result res;
...
TbxRun(dbid, taid,
    "spool into supp select * from suppliers", & qd, & res);
printf ("%ld tuples written to file supp\n",
    res._var.count.ntuples);
```

Example:

```
Id dbid, taid;
Query_descr qd;
...
TbxRun(dbid, taid, "drop table suppliers", & qd, NULL);
```

Example:

```
Id dbid, taid;
Query_descr qd;
Result res;
...
TbxRun(dbid, taid,
    "call javaproc(3, 'xyz'", & qd, & res);
```

2.7.8 TbxDml/TbxCursorOpen - Activate a Cursor or Scrolling Cursor

Input Parameter:

```
Id dbid;
Id taid;
char * statement;
unsigned long openmode;
```

Result Parameter:

```
Query_descr qd;
```

Call Syntax:

```
tbx (DML, dbid, taid, statement, &qd);
TbxDml (dbid, taid, statement, & qd);
TbxCursorOpen(dbid, taid, statement, &qd, openmode);
```

Effect: Parameter `statement` must point to a string which represents a SELECT, INSERT, UPDATE or DELETE Statement (see TB/SQL Reference Manual). It is recommended to process INSERT/UPDATE/DELETE statements by TbxRun to save communication overhead (one TbxRun call versus the sequence TbxDml, TbxEval,TbxClose).

The parameters "dbid" and "taid" must hold valid connection and transaction identifiers obtained by corresponding TbxConnect and TbxBt calls.

The statement is compiled and optimized and a description of the query is stored into the structure `qd`.

The following 3 variants are equivalent ways to open a non-scrolling cursor:

- tbx(DML,dbid,taid,statement,&qd)
- TbxDml(dbid,taid,statement,&qd)
- TbxCursorOpen(dbid, taid, statement, &qd, 0)

The result tuples of a non-scrolling cursor can be fetched (sequentially) with TbxEval calls, see next section.

A scrolling cursor is opened by:

- TbxCursorOpen(dbid, taid, statement, &qd, `0 | CURSOR_SCROLLABLE`)

A scrolling cursor enables forward and backward positioning on the result tuples. All result tuples are stored on kernel side for repeated access. This causes some overhead compared to a non-scrolling cursor.

The result tuples of a scrolling cursor can be fetched with TbxCursorFetch calls (see 2.7.10) or also with standard TbxEval calls (see 2.7.9) . TbxCursorFetch enables absolute and relative positioning on the result tuples.

After successful compilation, Transbase tries to lock the query objects. A SELECT Statement without FOR UPDATE clause requires read locks to be set, all other statements require update locks to be set (see TB/SQL Reference Manual).

If the requested locks would produce a deadlock the compilation of the query will immediately return the error code `LOCKS_NOT_GRANTABLE` (message: `query rejected: ..`). If the locks do not produce a deadlock, the `TbxDml` call possibly is delayed until all locks can be granted. The maximum period of waiting time can be controlled by the TIMEOUT setting. If a query times out, the error code `LOCK_TIMEOUT` (message: `query timed out: ..`) is returned. In either case the transaction remains active.

Upon successful return the result parameter `qd` will contain a description of the compiled query (for details see The Datatype Querydescr). `qd` has to be used as input parameters in subsequent `TbxEval` calls refering to this query. Note that the `fieldpointers` of the `field` array of `qd` are still undefined (they are set by the `TbxEval` call).

More than one query can be active at a time; however, all active queries of a single transaction must have compatible lock sets to define a clean update semantics. If a query conflicts with another query held active by the same transaction, an error code `QUERY_CONFLICT` (message: `query conflict: ..`) will be returned.

Errors:

```
ILL_DB_ID
ILL_TA_ID
QUERY_CONFLICT
Locks_NOT_GRANTABLE
LOCK_TIMEOUT
various DML syntax error codes
```

Example: See TbxEval - Evaluation of an Active Query

2.7.9 TbxEval - Evaluate a non-scrolling cursor

Input Parameter:

```
Query_descr qd;
```

Result Parameter:

```
Query_descr qd;
Result res;
```

Call Syntax:

```
tbx (EVAL, & qd, & res);
TbxEval (& qd, & res);
```

or

```
tbx (EVAL, & qd, NULL);
TbxEval (& qd, NULL);
```

Effect: Evaluates a previously compiled DML query or an activated stored query. The query is identified by `qd`. In case of a SELECT query (cursor), each `TbxEval` call evaluates one tuple of the result set of the query: if the result set is not yet exhausted then `eod` of `qd` is set to `ONE_TUPLE` and the `fieldpointers` of `qd` are set to the field values of the tuple (NULL pointers indicate SQL null values); otherwise `eod` is set to `NO_TUPLE` and the `fieldpointers` are undefined. In case of an UPDATE, INSERT, or DELETE query exactly one `TbxEval` call is necessary to evaluate (it is recommended, however, to evaluate these statements with TbxRun).

If a `res != NULL` is specified, `res` contains the result of the `TbxEval` call (i.e. a tuple or result information resp.). To let the program determine about the type of the result, the first component of `res`, namely `res.qtype` is an indicator which type of query was evaluated (`qtype` of `qd` contains the same information). This indicator is used to determine the appropriate union member of res, either `res._var.tuple` or `res._var.count`. For result tuples (accessed by `res._var.tuple`) a number of macros are available to access the fields of the tuple. For UPDATE, INSERT, DELETE queries, the structure `res._var.count` describes how many tuples have been updated, inserted or deleted.

Note: The array `field` of `qd` is dynamically allocated and freed on heap by Transbase, do not malloc or free on this pointer. After complete evaluation of the query, the query has to be CLOSEd explicitly.

Note: The component `eod` of `qd` is not a boolean flag; its values must be explicitly compared with the described symbolic constants (`NO_TUPLE` etc.).

Note: TbxEval also can evaluate a scrolling cursor although there is no use to create a scrolling cursor without direct positioning via TbxCursorFetch. (`NO_TUPLE` etc.).

Errors:

```
QU_NOT_OPEN
ILL_QU_ID
```

Example:

```
Id dbid, taid;
Query_descr qd;
...
TbxDml(dbid, taid, "SELECT * FROM systable", & qd);
while (1) {
   TbxEval(&qd, NULL);
   if (qd.eod == NO_TUPLE) break;
   printf("suppno: %ld ",*(Integer*)qd.field[0].fieldpointer);
   ...
}
TbxClose(&qd);
```

2.7.10 TbxCursorFetch - Evaluate a scrolling cursor

Input Parameter:

```
Query_descr qd;
int positionmode; -- CURSOR_POS_REL or CURSOR_POS_ABS
long position;
```

Result Parameter:

```
long actpos;
long maxpos;
```

Call Syntax:

```
TbxCursorFetch(&qd,positionmode,position,&actpos, &maxpos);
```

Effect: Fetches a tuple from a scrolling cursor. A scrolling cursor is created by TbxCursorOpen(..., `0 | CURSOR_SCROLLABLE`).

Parameters positionmode and position specify the actual position on the resultset.

Assume that the resultset contains n tuples.

The following describes the behaviour for positionmode==`CURSOR_POS_ABS`.

`CURSOR_POS_ABS` specifies an absolute positioning on the "position"-th tuple of the resultset. A value of "position" between 1 and n fetches the tuple, i.e. qd.field[i].fieldpointer are set, qd.eod is set to `ONE_TUPLE`, "actpos" is set to the value of position. If a "position" == 0 is specified, then no tuple is fetched and qd.eod is set to `NO_TUPLE`, actpos is set to 0. Semantically, the cursor is placed "before the first row". If "position" is `>` n, then also no tuple is fetched and qd.eod is set to `NO_TUPLE`, actpos is set to n+1. Semantically, the cursor is placed "behind the last row". In this case, the value of n becomes known to the kernel and "maxpos" is set to n. If "position" contains a negative number -k (for k `>=` 1), then the cursor is placed on the (n-k+1)-th tuple. For example, -1 positions onto the last tuple, -2 onto the last but one, etc. For k>n, the cursor is placed before the first row. For all negative values, also the value of maxpos becomes available.

`CURSOR_POS_REL` specifies positioning relative to the current position of the cursor (note that after opening the cursor, the actual position automatically is 0. i.e. before the first row). A position value of 1 advances the cursor by one tuple, a value of k `>=` 1 advances by k tuples, a value of 0 leaves the cursor on its actual position, a negative value moves the cursor the corresponding distance before the actual position. If the cursor is moved outside the range of 1 to n, no tuple is fetched and the actual position again is 0 (before the first row) or n+1 (behind the last row), resp., and qd.eod is set to `NO_TUPLE`.

Note: The component `eod` of `qd` is not a boolean flag; its values must be explicitly compared with the described symbolic constants (`NO_TUPLE`, `ONE_TUPLE` etc.).

Note: The positionmodes `CURSOR_POS_REL` and `CURSOR_POS_ABS` also can be mixed across several TbxCursorFetch calls.

Note: Positioning relative to the end (negative relative position) requires complete evaluation of the resultset inside the kernel (if done for the first time within this cursor).

Errors:

```
QU_NOT_OPEN
ILL_QU_ID
CRS_WRONG_TYPE
```

Example:

```
Id dbid, taid;
Query_descr qd;
long pos, actpos, maxpos=-1;
...
TbxCursorOpen(dbid, taid, "SELECT * FROM systable", &qd, 0|CURSOR_SCROLLABLE);
/* fetch last tuple of resultset */
TbxCursorFetch(&qd, CURSOR_POS_ABS,-1,&actpos,&maxpos);
if(qd.eod==ONE_TUPLE)
       printf("suppno: %ld ",*(Integer*)qd.field[0].fieldpointer);
/* fetch each 10-th result tuple reading from backward */
while (actpos > 10) {
   TbxCursorFetch(&qd, CURSOR_POS_REL,-10,&actpos,&maxpos);
   printf("suppno: %ld ",*(Integer*)qd.field[0].fieldpointer);
   ...
}
TbxClose(&qd);
```

2.7.11 TbxClose - Close a Query

Input Parameter:

```
Query_descr qd;
```

Result Parameter: None

Call Syntax:

```
tbx (CLOSE, & qd);
TbxClose (& qd);
```

Effect: Closes the query identified by `qd`. The query thus becomes inactive (the query descriptor qd remains unchanged). If the query was a dynamic query (i.e. compiled with a `TbxDml` call) it is not retained in any form in Transbase (a stored query, however, remains stored).

If the consistency level is set low (`CONS_2`) and the query identified by `qd` is a pure SELECT query (i.e. without FOR UPDATE), the `TbxClose` request automatically releases all locks of the query. If consistency level is high (default) no locks are released by the `TbxClose` request.

Errors:

```
QU_NOT_OPEN
ILL_QU_ID
```

Example:

```
Id dbid, taid;
Query_descr qd;
Result res;
...
TbxDml(dbid, taid, "delete from suppliers", & qd);
TbxEval(&qd, & res);
printf ("%ld tuples deleted\n", res._var.count.ntuples);
TbxClose(&qd);
```

2.7.12 TbxUpdPos - Perform an Update Positioned Statement

Input Parameter:

```
Query_descr qd;
char * statement;
```

Result Parameter: None

Call Syntax:

```
tbx (UPDPOS, & qd, statement);
TbxUpdPos (& qd, statement);
```

Effect: Parameter `qd` must be the query descriptor of an active SELECT FOR UPDATE query. The current tuple of that query is updated accordingly to the parameter `statement` which must represent an UPDATE statement where the CURRENT clause is specified.

The current tuple is defined to be the tuple retrieved by the last `TbxEval` call on the query specified by `qd`. No current tuple is defined if no `TbxEval` call has been performed on the specified query or if a `TbxDelPos` call or a `TbxDelPosStored` call has been performed on the current tuple or if the query has been completely evaluated.

All columns of the current tuple including primary key fields can be updated with the `TbxUpdPos` statement.

Errors are returned if the query specified by `qd` is not an active SELECT FOR UPDATE query (`QU_NOT_OPEN`) or if the statement is not an UPDATE statement with CURRENT clause (`NO_UPD_POS_STAT`) or if no current tuple is defined (`NOT_ON_ROW`) or if the UPDATE statement specifies another table than the corresponding SELECT statement (`WRONG_RELATION`).

Errors:

```
QU_NOT_OPEN
NOT_ON_ROW
NO_UPD_POS_STAT
NO_UPD_SELECT
WRONG_RELATION
various DML syntax errors
```

Example:

```
Id dbid, taid;
Query_descr qd;
char *selforupdate = "SELECT * FROM suppliers FOR UPDATE";
char *update = "UPDATE suppliers \
                SET address = '137 Park Avenue' \
                WHERE CURRENT";

...
TbxDml(dbid, taid, selforupdate, & qd);
...
TbxEval(&qd, NULL);
if (TbxUpdPos(&qd, update) == NO_ERROR)
    printf ("Updated\n");
```

2.7.13 TbxDelPos - Perform a Delete Positioned Statement

Input Parameter:

```
Query_descr qd;
```

Result Parameter: None

Call Syntax:

```
tbx (DELPOS, & qd);
TbxDelPos (& qd);
```

Effect: Parameter `qd` must be the query descriptor of an active SELECT FOR UPDATE query.

The current tuple of that query is deleted. The current tuple is defined to be the tuple retrieved by the last `TbxEval` call on the query specified by `qd`. No current tuple is defined if no `TbxEval` call has been performed on the specified query or if a `TbxDelPos` call or a `TbxDelPosStored` call has been performed on the current tuple or if the query has been completely evaluated.

After the `TbxDelPos` call no current tuple is defined until the next `TbxEval` call.

Errors are returned if the query specified by `qd` is not an active SELECT query (`QUERY_NOT_OPEN`) or if no current tuple is defined (`NOT_ON_ROW`).

Errors:

```
QU_NOT_OPEN
NOT_ON_ROW
NO_UPD_POS_STAT
```

Example:

```
Query_descr qd;
char *selforupdate = "SELECT * FROM suppliers FOR UPDATE";
...
TbxDml(dbid, taid, selforupdate, & qd);
TbxEval(&qd, NULL);
if (TbxDelPos(&qd) == NO_ERROR)
    printf ("Deleted\n")
```

2.7.14 TbxStore Statement - Store a Statement

Input Parameter:

```
Id dbid;
char *statement;
```

Result Parameter:

```
Id stmtid;
Query_descr qd;
```

Call Syntax:

```
tbx (STORE, dbid, statement, & stmtid, & qd)
TbxStore (dbid, statement, & stmtid, & qd)
```

Effect: The input parameter `statement` must point to a string which represents a SELECT, INSERT, UPDATE, DELETE or CALL Statement (see TB/SQL Reference Manual). The UPDATE Statement or DELETE Statement may also be specified with the CURRENT-clause. The statement may have formal parameters.

The statement is compiled, optimized and stored. A statement identifier is returned in result parameter `stmtid` which is used as input parameter in a `TbxOpenStored`, `TbxRunStored`, `TbxAddBatchParams`, or `TbxRunStoredBatch` call to identify the stored query. The returned stmtid is valid until the next `TbxDisconnect` call against the specified database. Arbitrary many `TbxOpenStored`, `TbxRunStored`, `TbxAddBatchParams` and `TbxRunStoredBatch` calls may be run against a STOREd query.

If the input parameter `qd` is not the NULL pointer, then a query description is returned into the structure pointed to by `qd`. Note that there is no valid `query_id` returned in `qd` because the query is not active. Likewise the `fieldpointers` of the `field` array of `qd` are undefined. Specifying a `qd` may be useful to inform about the properties of the stored query at store time already.

Note: The array `field` of `qd` is only valid until the next `TbxStore`, `TbxRunStored`, `TbxAddBatchParams`, `TbxClearBatchParams`, or `TbxRunStoredBatch` call.

Errors:

```
ILL_DB_ID
various DML syntax error codes
```

Example:

```
Query_descr qd;
Id stmtid;
...
TbxStore(dbid,
    "SELECT * FROM suppliers WHERE suppno = ?",
    & stmtid, & qd);
/* or :  "... WHERE suppno = #1(INTEGER)"  */
```

```
Query_descr qd;
Id stmtid;
...
TbxStore(dbid,
    "CALL javaproc(?,?)",
    & stmtid, & qd);
```

2.7.15 TbxRunStored - Run a Stored Statement

Input Parameter:

```
Id dbid;
Id taid;
Id stmtid;
Parameters par;
```

Result Parameter:

```
Query_descr qd;
Result res;
```

Call Syntax:

```
tbx (RUN_STORED, dbid, taid, stmtid, & par, & qd, & res);
TbxRunStored (dbid, taid, stmtid, & par, & qd, & res);
```

or

```
tbx (RUN_STORED, dbid, taid, stmtid, & par, & qd, NULL);
TbxRunStored (dbid, taid, stmtid, & par, & qd, NULL);
```

Effect: The stored query identified by `stmtid` is run.

The actual parameters given by `par` are bound to the formal parameters of the query. Error occurs if the number of parameter values does not match the number of formal parameters or if the types are not compatible. For differing but compatible types, type adaptation is performed.

A `TbxRunStored` call has exactly the same effect as the sequence (`TbxOpenStored` `TbxEval`; `TbxClose`), i.e. all three phases of statement execution are performed, namely query activation, exactly one evaluation, and the ing of the active query. The `TbxRunStored` call is not only an abbreviation but also results in better performance since it reduces communication overhead.

The two result parameters `qd` and `res` contain all the information that would be returned if the statement would be performed stepwise. They are analogous to the `TbxRun` statement. If the `res` parameter is set to NULL, no result is delivered in `res`.

For SELECT statements: note that only the first tuple of the query result (if any) is evaluated. Since the query automatically is closed, no further tuples can be fetched, i.e. no further `TbxEval` call can be applied to `qd`.

Note: The heap array `field` of `qd` including the `fieldpointers` remain valid only until the next `TbxStore` or `TbxRun` class call.

The `TbxRun` call is well-suited for SELECT queries that retrieve exactly one tuple, e.g. a tuple which is fully specified by its primary key values. In any case, the variable `eod` in `qd` and `res` is set to the value `NO_TUPLE`, `ONE_TUPLE`, `MORE_TUPLE` if the SELECT query evaluates no tuple, exactly 1 tuple or more than 1 tuple (only the first is delivered in the last case).

The identified stored query must not be an UPDATE positioned or DELETE positioned statement. These statements must be run with the special call `TbxUpdPosStored` or `TbxDelPosStored`.

Errors:

```
ILL_DB_ID
ILL_TA_ID
ILL_STMT_ID
WRONG_QUERY_TYPE
PARAMETER_MISMATCH
```

Example:

```
Query_descr qd;
Result res;
Id stmtid, dbid, taid;
Parameters par;
Param param;
Integer suppno;
TbxStore(dbid,
    "DELETE FROM suppliers WHERE suppno = ?",
    & stmtid, NULL);
...
par.param_no = 1;
par.param = & param;
param.type = TB__INTEGER;
param.value = (char*)&suppno;
...  read value onto variable suppno
TbxRunStored(dbid, taid, stmtid, & par, & qd, & res);
if(res._var.count.ntuples) printf ("tuple deleted\n"):
```

2.7.16 TbxAddBatchParams - Add batch parameters to a stored INSERT/UPDATE/DELETE statement

Input Parameter:

```
Id dbid;
Id taid;
Id stmtid;
Parameters par;
```

Result Parameter:

```
Query_descr qd;
long nres;
Count_result *res = NULL;
```

Call Syntax:

```
tbx(TB__ADD_BATCH_PARAMS,dbid,taid,stmtid,&par,&qd,&nres,&res);
TbxAddBatchParams(dbid,taid,stmtid,&par,&qd,&nres,&res);
```

or

```
tbx(TB__ADD_BATCH_PARAMS,dbid,taid,stmtid,&par,&qd,&nres,NULL);
TbxAddBatchParams(dbid,taid,stmtid,&par,&qd,&nres,NULL);
```

or

```
tbx(TB__ADD_BATCH_PARAMS,dbid,taid,stmtid,&par,&qd,NULL,NULL);
TbxAddBatchParams(dbid,taid,stmtid,&par,&qd,NULL,NULL);
```

Effect: Batch parameters are added to a stored INSERT/UPDATE/DELETE query identified by `stmtid`.

These batch parameters are initially stored in a local buffer. If the buffer reaches its size limit (MAXTUPLESIZE), usually after several calls to TbxAddBatchParams, then the batch is sent to the database server and executed. Result counts are retrieved for the executed part of the batch. The batch can be continued by issuing more TbxAddBatchParams calls. A batch is completed by calling TbxRunStoredBatch and thereby flushing the local parameter buffer and retrieving all results.

A sequence of n `TbxAddBatchParams` calls followed by a `TbxRunStoredBatch` have exactly the same effect as a sequence of n `TbxRunStored` statements. Running

stored queries in batch mode results in better performance since it minimizes communication overhead better than `TbxRunStored` calls.

The result parameters `qd`, `nres` and `res` contain all the information that would be returned if the statements would be performed stepwise, but only if the batch was sent to the server, i.e. `nres > 0` after `TbxAddBatchParams` calls or after a `TbxRunStoredBatch` call. Result counts can be accesses via `res[0].ntuples` and to `res[nres-1].ntuples` and `res[0].tried` to `res[nres-1].tried` respectively. If the `res` or `nres` parameters are set to NULL, no results are delivered.

Note: The array `field` of `qd` and the `res` array is dynamically allocated and freed on heap by Transbase, do not malloc or free on this pointer.

For SELECT statements: Select statements cannot be executed in batch mode.

Errors:

```
ILL_DB_ID
ILL_TA_ID
ILL_STMT_ID
WRONG_QUERY_TYPE
PARAMETER_MISMATCH
```

Example:

```
int i;
Id stmtid, dbid, taid;
Parameters par;
Param param[3];
Integer suppno;
char imgname[32];
char filename[32];
Query_descr qd;
Count_result *res = NULL;

TbxStore(dbid,
    "INSERT INTO graphic (imgname, imgid, data) VALUES (?,?,?)",
    & stmtid, NULL);
...
par.param_no = 3;
par.param = param;
param[0].type = TB__CHAR;
param[1].type = TB__INTEGER;
```

```
param[2].type = TB__BLOB;
param[0].value = imgname;
block[2].value = (char*) &bdesc;
bdesc.mode = FILENAME;
bdesc.loc.filename = filename;

for(i=0;i<5000;i++) {
    sprintf(imgname, "img%d", i);
    param[1].value = (char*) &i;
    sprintf(filename, "/tmp/file%d", i);
    TbxAddBatchParams(dbid, taid, stmtid, &par, &qd, &nres, &res);
    if(nres>0)
        [...] /* process results */
}

TbxRunStoredBatch(dbid, taid, stmtid, &qd, &nres, &res);
if(nres>0)
    [...] /* process results */
```

2.7.17 TbxClearBatchParams - Clear pending batch parameters from the applications batch buffer

Input Parameter:

```
Id dbid;
Id taid;
Id stmtid;
```

Result Parameter: None

Call Syntax:

```
tbx (TB__CLEAR_BATCH_PARAMS, dbid, taid, stmtid);
TbxClearBatchParams (dbid, taid, stmtid);
```

Effect: The local buffer holding a parameter set of a batch is freed.

Additionally the array `field` of `qd` and the `res` array freed on heap by Transbase.

Note: TbxAddBatchParams are completely undone by TbxClearBatchParams only if the local buffer has never been transferred to the server, i.e. `nres` was zero after all preceeding TbxAddBatchParams calls. Otherwise a batch can only be undone by calling TbxClearBatchParams and aborting the complete transaction.

Errors:

```
ILL_DB_ID
ILL_TA_ID
ILL_STMT_ID
```

Example:

```
Query_descr qd;
Count_result *res;
Id stmtid, dbid, taid;
Parameters par;
Param param;
Integer suppno;
State tastate;
int i;
```

```
int data_transferred=FALSE;

TbxStore(dbid,
    "INSERT INTO tmptable VALUES (?)", & stmtid, NULL);
...
par.param_no = 1;
par.param = & param;
param.type = TB__INTEGER;
bdesc.mode = FILENAME;
bdesc.loc.filename = filename;

for(i=0;i<5000;i++) {
    param.value = (char*) &i;
    TbxAddBatchParams(dbid, taid, stmtid, &par, &qd, &nres, &res);
    if(nres)
        data_transferred=TRUE;

TbxClearBatchParams(dbid, taid, stmtid, &par, &qd, &nres, &res);
if(data_transferred)
    TbxAt (taid, & tastate);
```

2.7.18 TbxRunStoredBatch - Run a stored INSERT/UPDATE/DELETE statement in batch mode

Input Parameter:

```
Id dbid;
Id taid;
Id stmtid;
```

Result Parameter:

```
Query_descr qd;
long nres;
Count_result *res;
```

Call Syntax:

```
tbx (TB__RUN_STORED_BATCH,dbid,taid,stmtid,&qd,&nres,&res);
TbxRunStoredBatch (dbid,taid,stmtid,&qd,&nres,&res);
```

or

```
tbx(TB__RUN_STORED_BATCH,dbid,taid,stmtid,&qd,&nres,NULL);
TbxRunStoredBatch(dbid,taid,stmtid,&qd,&nres,NULL);
```

or

```
tbx(TB__RUN_STORED_BATCH,dbid,taid,stmtid,&qd,NULL,NULL);
TbxRunStoredBatch(dbid,taid,stmtid,&qd,NULL,NULL);
```

Effect: Completes a batch of INSERT/UPDATE/DELETE statements by sending all pending batch parameters to the database server and retrieving all result counts. All resource associated top the batch are freed.

The result parameters `qd`, `nres` and `res` contain all the information that is returned by TbxAddBatchParams if the batch was sent to the server, i.e. `nres > 0` after the AddBatchParams call. Result counts can be accesses via `res[0].ntuples` and to `res[nres-1].ntuples` and `res[0].tried` to `res[nres-1].tried` respectively. If the `nres` or `res` parameters are set to NULL, no results are delivered.

Errors:

```
ILL_DB_ID
ILL_TA_ID
ILL_STMT_ID
WRONG_QUERY_TYPE
PARAMETER_MISMATCH
```

Example: Refer to 'TbxAddBatchParams'.

2.7.19 TbxOpenStored - Activate a Stored Statement

Input Parameter:

```
Id dbid;
Id taid;
Id stmtid;
Parameters par;
```

Result Parameter:

```
Query_descr qd;
```

Call Syntax:

```
tbx (OPEN_STORED, dbid, taid, stmtid, & par, & qd);
TbxOpenStored (dbid, taid, stmtid, & par, & qd);
```

Effect: The stored query identified by `stmtid` is activated. After the call the query is active. A query description with a valid `query_id` is returned in `qd`.

The formal parameters of the query are replaced by the actual parameters supplied by par. The i-th formal parameter is substituted by the actual parameter with index `[i-1]` (i.e. where `par->param[i-1].value` points to). The i-th formal parameter is `#i(...)` if this syntax is used in the query or the i-th ? sign if this syntax is used. Error occurs if the number of parameter values does not match the number of formal parameters or if the types are not compatible. For differing but compatible types, type adaptation is performed.

After the `TbxOpenStored` call the query is in the same state as if an equivalent query without formal parameters had been compiled with the `TbxDml` call, i.e. the query is active, and for subsequent `TbxEval` and `TbxClose` calls, the query is identified by `qd`. However, an `TbxOpenStored` call is faster than a `TbxDml` call.

The identified stored query must not be an UPDATE positioned or DELETE positioned statement. These statements must be run with the special call `TbxUpdPosStored` or `TbxDelPosStored`.

Errors:

```
ILL_DB_ID
ILL_TA_ID
ILL_STMT_ID
WRONG_QUERY_TYPE
PARAMETER_MISMATCH
```

Example:

```
Query_descr qd;
Id stmtid, dbid ,taid;
Parameters par;
Param param;
char charbuf[100];
...
TbxStore(dbid,
    "SELECT * FROM inventory WHERE description = ?",
    & stmtid, NULL);
par.param_no = 1;
par.param = & param;
param.type = TB__CHAR;
param.value = charbuf;
/* ...  read character string onto charbuf */
TbxOpenStored(dbid, taid, stmtid, & par, & qd);
TbxEval(&qd, NULL);
```

2.7.20 TbxCursorOpenStored - Open Scrolling Cursor on a Stored Statement

Input Parameter:

```
Id dbid;
Id taid;
Id stmtid;
Parameters par;
long openmode;
```

Result Parameter:

```
Query_descr qd;
```

Call Syntax:

```
TbxCursorOpenStored(dbid, taid, stmtid, & par, & qd, openmode);
```

Effect: The stored query identified by `stmtid` is activated. After the call the query is active. A query description with a valid `query_id` is returned in `qd`.

The formal parameters of the query are replaced by the actual parameters supplied by par. The i-th formal parameter is substituted by the actual parameter with index `[i-1]` (i.e. where `par->param[i-1].value` points to). The i-th formal parameter is `#i(...)` if this syntax is used in the query or the i-th ? sign if this syntax is used. Error occurs if the number of parameter values does not match the number of formal parameters or if the types are not compatible. For differring but compatible types, type adaptation is performed.

After the `TbxCursorOpenStored` call the query is in the same state as if an equivalent query had been opened with the `TbxCursorOpen` call2.7.8.

Especially, a scrolling cursor is opened if parameter "openmode" has the value `0 | CURSOR_SCROLLABLE`.

The identified stored query must not be an UPDATE positioned or DELETE positioned statement. These statements must be run with the special call `TbxUpdPosStored` or `TbxDelPosStored`.

Errors:

```
ILL_DB_ID
ILL_TA_ID
ILL_STMT_ID
WRONG_QUERY_TYPE
PARAMETER_MISMATCH
```

Example:

```
Query_descr qd;
long actpos, maxpos;
Id stmtid, dbid ,taid;
Parameters par;
Param param;
char charbuf[100];
...
TbxStore(dbid,
    "SELECT * FROM inventory WHERE description LIKE ?",
    & stmtid, NULL);
par.param_no = 1;
par.param = & param;
param.type = TB__CHAR;
param.value = charbuf;
...  read character string onto charbuf
TbxCursorOpenStored(dbid, taid, stmtid, & par, & qd, 0 | CURSOR_SCROLLABLE);
/* position to last result tuple */
TbxCursorFetch(&qd, CURSOR_POS_ABS,-1,&actpos,&maxpos);
TbxClose(&qd);
```

2.7.21 TbxUpdPosStored - Run a Stored Update Positioned Statement

Input Parameter:

```
Query_descr qd;
Id stmtid;
Parameters par;
```

Result Parameter: None

Call Syntax:

```
tbx (UPDPOS_STORED, & qd, stmtid, & par);
TbxUpdPosStored (& qd, stmtid, & par);
```

Effect: Parameter `stmtid` must identify a stored UPDATE Statement where the CURRENT-Clause is specified. Parameter `qd` must be the query descriptor of an active SELECT FOR UPDATE query. The table specified in the UPDATE statement must be the same as that specified in the corresponding SELECT statement.

The current tuple of that query is updated accordingly to the UPDATE Statement identified by `stmtid`. The current tuple is defined to be the tuple retrieved by the last `TbxEval` call on the query specified by `qd`. No current tuple is defined if no `TbxEval` call has been performed on the query specified or if a `TbxDelPos` or `TbxDelPosStored` call has been performed on the current tuple or if the query has been completely evaluated.

All columns of the current tuple including primary key attributes can be updated.

Note that the SELECT statement specified by `qd` can be itself a stored activated query as well as a query compiled and activated by a `TbxDml` call.

Errors:

```
QU_NOT_OPEN
NOT_ON_ROW
NO_UPD_POS_STAT
NO_UPD_SELECT
WRONG_RELATION
```

Example:

```
Id dbid, taid;
Query_descr qd;
Id stmtid;
Parameters par;
Param param;
Double d;
...
TbxStore(dbid,
    "UPDATE quotations SET price = ? WHERE CURRENT",
    & stmtid, NULL);
par.param_no = 1;
par.param = & param;
param.type = TB__DOUBLE;
param.value = (char*) &d;
TbxDml(dbid, taid,
    "SELECT * FROM quotations",
    & qd);
while (1) {
    TbxEval(&qd, NULL);
    if( <condition> ) {
          .. read value onto variable d
          TbxUpdPosStored(&qd, stmtid, &par);
    }
}
```

2.7.22 TbxDelPosStored - Run a Stored Delete Positioned Statement

Input Parameter:

```
Query_descr qd;
Id stmtid;
```

Result Parameter: None

Call Syntax:

```
tbx (DELPOS_STORED, & qd, stmtid)
TbxDelPosStored (& qd, stmtid)
```

Effect: Parameter `stmtid` must identify a stored DELETE Statement where the CURRENT-Clause is specified. Parameter `qd` must be the query descriptor of an active SELECT FOR UPDATE FOR UPDATE query. The table specified in the UPDATE statement must be the same as that specified in the corresponding SELECT statement.

The current tuple of that query is deleted. The current tuple is defined to be the tuple retrieved by the last `TbxEval` call on the query specified by `qd`. No current tuple is defined if no `TbxEval` call has been performed on the query specified or if a `TbxDelPos` or `TbxDelPosStored` call has been performed on the current tuple or if the query has been completely evaluated.

After the `TbxDelPosStored` call no current tuple is defined until the next `TbxEval` call.

Note that the SELECT statement specified by `qd` can be itself a stored activated query as well as a query compiled and activated by a `TbxDml` call.

Errors are returned if the query specified by `qd` is not an active SELECT query (`QU_NOT_OPEN`) or if no current tuple is defined (`NOT_ON_ROW`).

Errors:

```
NOT_ON_ROW
NO_UPD_POS_STAT
NO_UPD_SELECT
```

Example:

```
Id dbid, taid;
Query_descr qd;
Id stmtid;
TbxStore(dbid,
    "DELETE FROM quotations WHERE CURRENT",
    & stmtid, NULL);
TbxDml(dbid, taid,
    "SELECT * FROM quotations",
    & qd);
while (1) {
    TbxEval(&qd, NULL);
    if( <condition> ) {
        TbxDelPosStored(&qd, stmtid);
    }
}
```

2.7.23 TbxDropStored - Drop a Stored Statement

Input Parameter:

```
Id dbid;
Id stmtid;
```

Result Parameter: None

Call Syntax:

```
tbx (DROP_STORED, dbid, stmtid);
TbxDropStored (dbid, stmtid);
```

Effect: The stored query identified by `stmtid` is dropped. Space occupied by the stored query is released.

Note: It is not necessary for an application to drop a stored query except that the Transbase kernel would run out of space (stored queries occupy main memory). At the `TbxDisconnect` of an application, all stored queries are automatically dropped.

Errors:

```
ILL_DB_ID
ILL_STMT_ID
```

Example:

```
Id stmtid, dbid;
TbxStore(dbid,
    "SELECT * FROM inventory WHERE description = #1(CHAR(*))",
    & stmtid, NULL);
            /* use it s often as you want */
TbxDropStored(dbid, stmtid);
```

2.7.24 TbxDropAllStored - Drop all Stored Statement

Input Parameter:

```
Id dbid;
```

Result Parameter: None

Call Syntax:

```
tbx (DROP_ALLSTORED, dbid);
TbxDropAllStored (dbid);
```

Effect: All queries on the specified database which have been stored by the application are dropped. Space occupied by the stored queries are released.

Note: It is not necessary for an application to drop a stored query except that the Transbase kernel would run out of space (stored queries occupy main memory). At the `TbxDisconnect` of an application, all stored queries are automatically dropped.

Errors:

```
ILL_DB_ID
```

Example:

```
Id stmtid, dbid;
TbxStore(dbid,
    "SELECT * FROM inventory WHERE description = #1(CHAR(*))",
    & stmtid, NULL);
            /* and some other queries ; use it as often as you want */
TbxDropAllStored(dbid);
```

2.7.25 TbxSetSortOrder - Setting a User Sortorder

Input Parameter:

```
Id dbid;
unsigned char sort_array[256];
```

Result Parameter: None

Call Syntax:

```
tbx (SET_SORTORDER, dbid, sort_array);
TbxSetSortOrder (dbid, sort_array);
```

Effect: The call defines a character sortorder for result tuples of queries. The sortorder is defined via parameter `sort_array`, e.g. if `sort_array['A']` is less than `sort_array['a']` then 'A' is considered smaller than 'a' in the result set ordering. The sortorder is valid for result fields of type CHAR and BINCHAR. Note that the sortorder explicitly set by this call only influences the interpretation of the explicit ORDER BY Clause in the SQL SELECT query but not the result set nor the internal sortorder of stored tuples.

When an application starts, the initial sortorder corresponds to the machine code. An explicit sortorder is valid until the next `TbxSetSortOrder` call.

A `TbxSetSortOrder` call can be issued inside or outside a transaction and is not set back when a transaction aborts.

Errors:

```
- ILL_DB_ID
```

Example:

```
Id dbid;
unsigned char sort_array[256];
         /* store values into sort_array .. */
TbxSetSortOrder(dbid, sort_array);
```

2.7.26 TbxGetSortOrder - Retrieving the Actual Sortorder

Input Parameter:

```
Id dbid;
```

Result Parameter:

```
unsigned char sort_array[256];
```

Call Syntax:

```
tbx (GET_SORTORDER, dbid, sort_array);
TbxGetSortOrder (dbid, sort_array);
```

Effect: The call retrieves the actually valid sortorder into the output parameter sort_array.

Errors:

```
ILL_DB_ID
```

Example:

```
Id dbid;
unsigned char sort_array[256];
TbxGetSortOrder(dbid, sort_array);
        /* now the actual sortorder is in "sort_array" */
```

2.7.27 TbxTbmode - Run a Tbmode Statement

Input Parameter:

```
Id dbid;
char * statement;
```

Result Parameter: None

Call Syntax:

```
tbx (TBMODE, dbid, statement);
TbxTbmode (dbid, statement);
```

Effect: The statement given by the input parameter `statement` (as a character string) is executed. The statement must be one of the Tbmode statements (see TB/SQL Reference Manual). The call may be issued inside or outside a transaction.

Note: Tbmode statements itself begin with the keyword `TbxTbmode`, so the TBX call looks somewhat redundant.

Errors: various syntax error codes

Example:

```
Id dbid;
TbxTbmode(dbid, "TBMODE CATALOG SIZE 100");
TbxTbmode(dbid, "TBMODE RESULT BUFFER TUPLES 1");
```

2.7.28 TbxSetDatDir - Define a Directory for Spool Files

Input Parameter:

```
Id dbid;
char * pathname;
```

Call Syntax:

```
tbx (SET_DAT_DIR, dbid, pathname);
TbxSetDatDir (dbid, pathname);
```

Effect: The directory whose name is given by the input parameter `pathname`, is used to locate user files for SPOOL INTO and SPOOL FROM statements run against the database identified by `dbid`. By having `dbid` as input parameter, it is possible to have different directories for each database connected to.

If no `TbxSetDatDir` call is issued, the current directory of the application program is used as default.

No error will be returned if pathname is an invalid directory name. Instead, this error will occur when running a SPOOL statement against the particular database.

The directory pathname must be an absolute path, i.e. on UNIX machines, it must begin with a slash, e.g. /tmp/spool. If this directory pathname has been specified in a `TbxSetDatDir` command and the filename spfile occurs in a SPOOL statement, actually the file /tmp/spool/spfile would be used.

On WINDOWS the pathname can be specified with the DOS syntax, e.g. `C:\tmp\spool`. On VMS, the filename syntax would look like `[tmp.spool]` and is automatically prefixed with the current drive. On VMS the directory names are restricted in that they cannot contain a /.

On both machines one can also use UNIX directory syntax provided the name ends with a slash, i.e. one could specify /tmp/spool/ on both VMS and WINDOWS.

Errors:

```
ILL_DB_ID
```

2.7.29 TbxSetTimeOut Statement

Input Parameter:

```
Int4 timeout;
```

Call Syntax:

```
tbx (SET_TIME_OUT, timeout);
TbxSetTimeout (timeout);
```

Effect: A new default value for timeouts is set. The value specified by the `timeout` parameter, is given in seconds.

A value `timeout <= 0` means that no timeout will occur. This may result in indefinite delay due to tbx calls.

If no `TbxSetTimeOut` call is given a default value of 60 seconds applies.

Note that the timeout setting is valid for all databases from now on whether they have been connected already or not.

2.7.30 TbxSetConsistency Statement

Input Parameter:

```
int level;
```

Result Parameter: None

Call Syntax:

```
tbx (SET_CONSISTENCY, level);
TbxSetConsistency (level);
```

Effect: The consistency level for the application program is set to the specified level. Level may have one of the following 3 values:

`CONS_3`	(highest consistency)
`CONS_2`	(medium consistency)
`CONS_1`	(lowest consistency)

`CONS_3` holds read locks until the end of a transaction.

At level `CONS_2` read locks of a pure SELECT statement are released automatically by the corresponding `TbxClose` statement.

At level `CONS_1` read queries run without any locks.

Note: `TbxSetConsistency` must not be called within an active transaction.

Errors:

```
OUT_TA_ONLY
ILL_CONS_LEVEL
```

2.7.31 TbxGetTaState - Get Transaction State

Input Parameter:

```
Id taid;
```

Result Parameter:

```
State tastat;
```

Call Syntax:

```
tbx (GET_TA_STATE, taid, & tastat);
TbxGetTaState (taid, & tastat);
```

Effect: The current state of the transaction given by identifier `taid` is returned in tastat. Note that `taid` need not be an identifier returned by a previous `TbxBt` call, but may be any value in the intervall `[0 .. MAX_TA]`.

Upon return, tastat will contain one of the following values:

TA_NOT_ACTIVE : The given value has not been used yet (i.e. no `TbxBt` call has returned this value) or the identifier is beyond the intervall `[0 .. MAX_TA]`.

TA_ACTIVE : The given value identifies an active transaction, i.e. it has been returned in a `TbxBt` call, but the transaction has not yet been ended (i.e. committed or aborted).

TA_COMMITTED : The given value has been used as a transaction identifier within this application. The last transaction identified by this value has been committed successfully, i.e. their updates have been saved to disk and are visible to other transactions. No active transaction is being identified by this identifier at the moment.

TA_ABORTED : The given value has been used as a transaction identifier within this application. The last transaction identified by this value has been aborted successfully, i.e. their updates have been rolled back to the state as of transaction start. No active transaction is being identified by this identifier at the moment.

TA_UNDEF : The given value has been used as a transaction identifier within this application. The state of this transaction cannot be determined by tbx, yet, probably due to a communication failure. No active transaction is being identified by this identifier at the moment.

Errors:

ILL_TA_ID

2.7.32 TbxGetDbState - Get Database State

Input Parameter:

```
Id dbid;
```

Result Parameter:

```
State dbstat;
```

Call Syntax:

```
tbx (GET_DB_STATE, dbid, & dbstat);
TbxGetDbState (dbid, & dbstat);
```

Effect: The current state of the connection given by identifier `dbid` is returned in dbstat. Note that `dbid` need not be an identifier returned by a previous `TbxConnect` call, but may be any value in the intervall `[0...MAX_DB]`.

Upon return, dbstat will contain one of the following values:

DB_DCONN : The given value has not been used yet (i.e. no `TbxConnect` call has returned this value) or a corresponding `TbxDisconnect` request has been issued for this connection or the identifier is beyond the intervall `[0...MAX_DB]`.

DB_CONN : The given value identifies an active connection, i.e. it has been returned in a `TbxConnect` call, but the connection has not yet been ended (by a `TbxDisconnect` call).

DB_LOGGED : The given value has been used as a connection identifier within this application. A successful `TbxLogin` request has been issued for this database connection.

DB_KILLED : The given value has been used as a connection identifier within this application. The corresponding Transbase process has been killed and has not yet restarted. The connection can no longer be used.

Errors:

```
ILL_DB_ID
```

2.7.33 TbxSendEvent - Send an Application Event to the Database

Input Parameter:

```
Id dbid;
char * eventinfo;
char * descr_1;
char * descr_2;
```

Result Parameter: None

Call Syntax:

```
tbx (SEND_EVENT, dbid, eventinfo, descr_1, descr_2);
TbxSendEvent (dbid, eventinfo, descr_1, descr_2);
```

Effect: Sends an APPLIC event to the specified database. If the account-logging for APPLIC events is activated, an entry with the supplied informations for `Eventinfo`, `Descr_1` and `Descr_2` is added to the accounting file of the database.

If the NULL pointer is given for `eventinfo`, `descr_1` or `descr_2`, a corresponding NULL value representation is generated. The supplied informations possibly are truncated to the maximum tuplesize of the actual database.

Errors:

```
ILL_DB_ID
```

Example:

```
Id dbid;
TbxSendEvent(dbid, argv[0], "Starting...", NULL);
```

2.7.34 TbxGetPlan - Get Evaluation Plan from Database

Input Parameter:

```
Id dbid; char * filename; char * mode;
```

Result Parameter: None

Call Syntax:

```
tbx (TB__GETPLAN, dbid, filename, mode);
TbxGetPlan (dbid, filename, mode);
```

Effect: Gets an evaluation plan from a database. The plan retrieved referres to the very last closed query. The file contains two plans, one after compilation of the query and one after closing. For `mode` you can specify `"a"` for appending data to `filename` or `"w"` otherwise.

Errors:

```
ILL_DB_ID
```

Example:

```
Id dbid; TbxGetPlan(dbid, "plan.txt", "w");
```

constant	desired action
`TbxConnect`	connect to a database
`TbxDisconnect`	disconnect from a database
`TbxLogin`	authorize as user
`TbxBt`	begin transaction
`TbxCt`	commit transaction
`TbxAt`	abort transaction
`TbxDml`	compile a DML-statement
`TbxEval`	evaluate a DML-statement
`TbxClose`	close a DML-statement
`TbxRun`	run , i.e. compile and evaluate a statement
`TbxUpdPos`	update the current tuple within evaluation phase
`TbxDelPos`	delete the current tuple within evaluation phase
`TbxStore`	precompile and store a DML statement
`TbxOpenStored`	activate a stored DML statement
`TbxRunStored`	run a stored DML statement
`TbxAddBatchParams`	add batch parameters to a stored INSERT/UPDATE/DELETE statement
`TbxClearBatchParams`	clear pending batch parameters from the applications batch buffer
`TbxRunStoredBatch`	run a stored INSERT/UPDATE/DELETE statement in batch mode
`TbxUpdPosStored`	run a stored UPDATE positioned
`TbxDelPosStored`	run a stored DELETE positioned
`TbxDropStored`	drop a stored query
`TbxDropAllStored`	drop all stored query
`TbxSetSortOrder`	set a user sortorder
`TbxGetSortOrder`	retrieve the actually valid sortorder
`TbxTbmode`	run a Tbmode statement
`TbxSetDatDir`	define a directory for external files
`TbxSetTimeOut`	define a value for timeout length
`TbxSetConsistency`	define the consistency level
`TbxGetTaState`	get the actual transaction state
`TbxGetDbState`	get the actual database state
`TbxGetBlob`	get a BLOB object
`TbxMakeBlob`	create a BLOB object

Table 2.3: Overview of Interface Functions

Chapter 3

Tuples and Fields

The result data of a SELECT query can be accessed by an application program in two different fashions. The field values can be read via the `fieldpointer` of the `Query_descr` or the tuple can be retrieved as a whole into a structure of type Result. In the latter case a couple of macros can be used to retrieve the field values, the number of fields, or the length of a tuple. All these macros are not needed if the tuple is retrieved via the `fieldpointer` of a `Query_descr`. The macros are defined in `tbx.h`.

The structure of a tuple with n fields is given by two parts:

- An array of n+1 pointers where each pointer points to the offset of the i-th field, the pointer n+1 points behind the last field.
- n field values, where each field starts on an alignment boundary, so that there are possibly up to three padding bytes included at the end of a field.

A tuple of three fields on a 4-byte-alignment machine would look like:

Offset	Contents	Comments
0	`8`	Offset of first field
2	`12`	Offset of second field
4	`16`	Offset of third field
6	`20`	Offset of tuple end
8	`ab\0`	First field (string)
11	`\0`	Padding to 12
12	`2341`	Second field (integer)
16	`X\0`	Third field (string)
18	`\0\0`	Padding to 20
20		end of tuple

3.1 Access to a Field

Syntax:

```
char *ptoattornull(tp,i)        /* Macro */
Tuple *tp;
unsigned i;
```

Effect: Returns a pointer to the i-th field (`i>=0`) of a tuple pointed to by a tuple pointer `tp`. Note that field numbering starts at 0. Returns the NULL pointer if the field contains the null value. The returned pointer is of type (char *); if the field type is not String, an appropriate cast operator must be applied to the pointer before referencing the value of the field.

Example: Assume we have a 3-ary tuple delivered in the structure `res` of type `Result` with three fields, the first of type Integer, the second of type Double and the third of type String. We assume further that no null values are delivered. If this cannot be assumed, each field has to be checked for the null value, before referencing it.

```
Query_descr qd;
Result res;
Integer a;
Double b;
String c;
...
TbxEval(&qd, & res);
a = *(Integer *)ptoattornull(res._var.tuple, 0);
b = *(Double *)ptoattornull(res._var.tuple, 1);
strcpy (c, ptoattornull (res._var.tuple, 2));
```

Note: Analogous type castings on pointers to field values must be performed when using the `fieldpointer` of a `Query_descr`. See The Datatype Querydescr.

3.2 Number of Fields

Syntax:

```
short tpattno(tp)               /* Macro */
Tuple *tp;
```

Effect: Returns number of fields of the tuple pointed to by `tp`. The number of fields of a result tuple can also be found in the component `qattr_no` contained in the description of the corresponding query See The Datatype Querydescr.

3.3 Length of Tuple

Syntax:

```
short tplgth(tp)                /* Macro */
Tuple *tp;
```

Effect: Returns length of tuple tp in bytes. This length is the true size of the tuple as stored on disk or in memory. Especially, the size includes all padding bytes which are eventually necessary to obey alignment restrictions.

3.4 Length of Field

Syntax:

```
short attlgth(tp,i)            /* Macro */
Tuple *tp;
unsigned i;
```

Effect: Returns length of the i-th field (`i>=0`) of a tuple pointed to by the tuple pointer `tp`. Note that field numbering start with 0.

Note: Due to alignment issues on some machines the length of a field might be up to 3 bytes greater than the length of the actual data stored in this field.

Example: The size of a four-byte string suffixed by a `'\0'` byte is 5. If this string is contained as a field in a tuple on a 4-byte-alignment machine, it will be padded with three `'\0'` bytes and the value returned by attlgth will be 8.

3.5 End of Result

Syntax:

```
int is_empt_tuple(tp)            /* Macro */
Tuple *tp;
```

Effect: Returns TRUE (1) if the tuple pointed to by `tp` is the endmarker of the result set of a SELECT query, otherwise FALSE (0).

Note: Checking the result end of a SELECT query can also be performed via the `eod` variable in `Query_descr` (but note that eod is not a boolean flag but has 3 distinct values, See The Datatype Querydescr.

Chapter 4

BLOBs at the TBX Programming Interface

4.1 Type and Type Encoding of BLOBs

The TBX Programming Interface uses a data structure called `Blobdesc` to retrieve and insert BLOB objects. Via a `Blobdesc` the user describes where the BLOB is to be stored in case of retrieval and where the BLOB can be found for an INSERT statement. A `Blobdesc` has the following structure:

```
typedef struct {
    int mode;                     /* FILENAME / MMADR */
    Int4 size;
    union {
        char *filename;
        struct {
            short usrmalloc;      /* 0/1, set by user program */
            unsigned mallocsize;  /* for system */
            char *mmadr;          /* main mem. address */
        } mmem;
    } loc;
} Blobdesc;
```

The structure Query_descr. from the compilation of the statement contains the field types of the result tuples in the components field[i].fieldtype. The symbolic constant `TB__BLOB` is used to describe a field with BLOB objects. For example, the following code fragment tests if the first field is of type `TB__BLOB`:

```
if(qd->field[0].fieldtype == TB__BLOB) ...
```

4.2 Fetching BLOB Objects with TBX

After an `TbxEval` call for a query with BLOBs, each BLOB result field contains a structure of the C type "Blob" which is defined in the include file tbx.h.

```
typedef struct {
    unsigned long size;     /* size of BLOB object */
    Blobadr       blobadr;
} Blob;     /* this is the type of a BLOB as it appears in a tuple
```

The component "size" of the BLOB field can be read and interpreted by the application. The component "blobaddr" is of no other use than to deliver it as parameter to a successive call to fetch the BLOB object itself. Note that the `TbxEval` call does not transfer the BLOB object itself to the application. A separate TBX function TbxGetBlob supplied with a pointer to the delivered result field of type Blob must be used to get the BLOB object. This also means that if no `TbxGetBlob` call is made (immediately) after an `TbxEval` call, the BLOB object(s) of the current tuple are not fetched and thus ignored.

A `TbxGetBlob` call for the most recently fetched tuple can only be done before the next `TbxEval` call or the next TbxRun call or the next TbxClose call. Thus, if no `TbxGetBlob` calls are made for the BLOBs of the last retrieved tuple, its BLOBs cannot be fetched any more.

For a SELECT query which delivers exactly 1 tuple, the TbxRun call normally is the most efficient way to run the query. A `TbxGetBlob` call, however, is not allowed to the result of a TbxRun call. Thus, in the case of BLOB objects, a SELECT query must always be run via the `TbxEval` mechanism.

4.2.1 TbxGetBlob

Input Parameter:

```
Id dbid;
Id taid;
Blob *blobfield;
Blobdesc *blobdesc;
```

Result Parameter: None

Call Syntax:

```
tbx (GETBLOB, dbid, taid, blobfield, blobdesc);
TbxGetBlob(dbid, taid, blobfield, blobdesc);
```

Effect: The `TbxGetBlob` call fetches the BLOB identified by parameter `blobfield`. The contents of `blobdesc` control the storage of the BLOB.

If `blobdesc->mode == FILENAME` then a file with the name supplied in `blobdesc->loc.filename` is created and the BLOB is stored in the created file. An error occurs if a file with the supplied name cannot be created.

If `blobdesc->mode == MMADR` then the BLOB is stored in main memory. For this purpose TBX inspects the component `blobdesc->loc.mmem.usrmalloc` as described below.

A value of 1 in `blobdesc->loc.mmem.usrmalloc` indicates that the user has reserved enough space to store the BLOB, TBX takes `blobdesc->loc.mmem.mmadr` as the target address to store the BLOB.

A value of 0 in `blobdesc->loc.mmem.usrmalloc` indicates that the user leaves it up to TBX to reserve storage. Here the component `blobdesc->loc.mmem.mmadr` must be set to NULL by the user program before the variable blobdesc is used for the first time in the `TbxGetBlob` call. TBX then allocates appropriate storage for the BLOBs, remembers in `blobdesc->loc.mmem.mallocsize` the actually allocated storage size and in `blobdesc->loc.mmem.mmadr` the actual storage address. Whenever a BLOB is not greater than the last stored one, the storage is immediately reused, otherwise it is released and reallocated with the appropriate size. Note, however, that the most recently allocated storage remains allocated. If the `blobdesc` is not used for a long time or never again, the user program may free the storage and reinitialize `blobdesc` as above.

In all cases described above TBX sets the component `blobdesc->size` to the size of the retrieved and stored BLOB object.

Note To Examples: In all of the examples below it is assumed that the program has connected and a transaction open, i.e. dbid and taid are set.

Note: On Windows platforms the memory space must be allocated by the application. Memory cannot be allocated by tbx on these platforms.

Example:

```
/***************************
a 1-tuple SELECT query with the TbxRun call;
the BLOB will be stored in file BLOB001
***************************/

Id dbid, taid;
Query_descr qd;
char *statement =
```

```
   "SELECT image FROM graphic WHERE graphic_name = 'g002'";
Blobdesc blobdesc;
Blob *bfield;
TbxDml(dbid, taid, statement, & qd);
TbxEval(&qd, NULL);
if ( qd.eod != NO_TUPLE )
{
   bfield = (Blob*)qd->field[0].fieldpointer;
      /* bfield->size is the size of the BLOB */
   blobdesc.mode = FILENAME;
   blobdesc.loc.filename = "BLOB001";
   TbxGetBlob(dbid, taid, bfield, &blobdesc);
}
```

Example:

```
/***************************
a SELECT query with DML and TbxEval calls;
the BLOBs are transferred into main memory
storage is reserved by the user program
***************************/
Id dbid, taid;
Query_descr qd;
char *statement =
    "SELECT image FROM graphic WHERE image IS NOT NULL";
Blobdesc blobdesc;
Blob *bfield;
TbxDml(dbid, taid, statement, & qd);
TbxEval(&qd, NULL);
if ( qd.eod != NO_TUPLE )
{
    bfield = (Blob*)qd->field[0].fieldpointer;
    /* bfield->size is the size of the BLOB */
    blobdesc.mode = MMADR;
    blobdesc.loc.mmem.usrmalloc = 1;
    blobdesc.loc.mmem.mmadr = (char*)malloc(bfield->size);
    TbxGetBlob(dbid, taid, bfield, &blobdesc);
    ...
}
```

4.3 Inserting and Updating BLOB Objects with TBX

Insertion of BLOB objects or update of tuples with new BLOB objects is also done in 2 steps. First a BLOB object is prepared in the database by the `TbxMakeBlob` call. The user supplies a `blobdesc` to the `TbxMakeBlob` call. The semantics of `blobdesc` is analogous to that of BLOB selection: the user indicates where the BLOB to be created is stored (file or main memory). Note that in the latter case, the user must also set the component `blobdesc.size` - this is the only case where `blobdesc.size` is set by the user. The user also supplies a so called `blobname` to the `TbxMakeBlob` call. This name is of temporary validity and only serves to identify the created BLOB object in a successive INSERT or UPDATE statement. After the INSERT or UPDATE statement has been performed, the blobname becomes invalid.

Note that the `TbxMakeBlob` call only transfers the BLOB object to the host kernel process and declares a blobname. Without an INSERT or UPDATE statement referring to the blobname, the corresponding BLOB would not be inserted into the database (and neither would occupy space in the database).

Theoretically, the same blobname can be used on more than one field position in an INSERT or UPDATE statement (if the tuple to be inserted has more than one BLOB field); but note that each BLOB reference in the statement causes the storage of a separate copy of the BLOB object. Likewise it is not possible, to store multiple references to one and the same BLOB object in different tuples. Of course, multiple references can easily be modelled by an appropriate database schema.

When a blobname is used in the VALUES clause of an INSERT statement or in the SET clause of an UPDATE statement, it must be preceded by the keyword BLOB. The blobname may be single-quoted, double-quoted or non-quoted.

4.3.1 TbxMakeBlob

Input Parameter:

```
Id dbid;
Id taid;
char * blobname;
Blobdesc *blobdesc;
```

Result Parameter: None

Call Syntax:

```
tbx (MAKEBLOB, dbid, taid, blobname, &blobdesc);
TbxMakeBlob(dbid, taid, blobname, &blobdesc);
```

Effect: The `TbxMakeBlob` call creates a BLOB object with the name supplied in parameter `blobname` in the database. The contents of `blobdesc` specifies where the BLOB is supplied.

If `blobdesc->mode == FILENAME` then the BLOB is fetched from a file with name `blobdesc->loc.filename`. This file must exist.

If `blobdesc->mode == MMADR` then the BLOB is fetched from the main memory address `blobdesc->loc.mmadr`. In this case, the user must specify the size of the BLOB in the component `blobdesc->size`.

The blobname is valid until the next INSERT or UPDATE statement has been performed.

Example:

```
/* an INSERT and UPDATE query where the BLOB is fetched from a file "BLOB001"
*/
Id dbid, taid;
Query_descr qd;
Blobdesc blobdesc;
char *blobname = "tmpbname";
char *upd, *ins;
ins = "INSERT INTO graphic VALUES('imag001', 134, BLOB tmpbname)";
ins = "INSERT INTO graphic VALUES('imag001', 134, BLOB 'tmpbname')";

blobdesc.mode = FILENAME;
blobdesc.loc.filename = "BLOB001";
TbxMakeBlob(dbid, taid, blobname, &blobdesc);
TbxRun(dbid, taid, ins, & qd, NULL);

/* or an UPDATE statement */

upd = "UPDATE graphic SET image = BLOB tmpbname \
       WHERE graphic_name = 'imag001'";
TbxRun(dbid, taid, upd, & qd, NULL);
```

Chapter 5

Signal Handling

Transbase not only offers a rigorous transaction concept with the ability to recover from any database actions back to a consistent state. Transbase also offers a method to asynchronously abort a running transaction. This is a very convenient feature for interactive applications. Imagine e.g. a situation where a mis-typed query has been issued whose runtime is extremely long. Instead of trying to abort the whole process by OS commands one can abort this transaction asynchronously, using the Transbase features described in this section.

The application interface tbx itself does not take care of any signals. This means that the application has full responsibility for signal handling, whether to ignore signals or to define user-specific signal handling or to leave the system defaults unchanged.

tbx provides a function for signal handling which may be called asynchronously at any time. The name of this additional function is `TbxInterrupt`. Note that `TbxInterrupt` is the only tbx function which may be called by a user-defined signal handling procedure. Asynchronously calling other tbx functions except `TbxInterrupt` may confuse tbx and its communication protocols totally.

For simplicity, we assume the name of the user-defined signal handling procedure to be `ap_sighandler` in the sequel.

Calling `TbxInterrupt` aborts all the application's active transactions (on all databases) asynchronously. In any case, the application remains connected to all databases it had been connected to. By the result parameter of `TbxInterrupt`, the following two cases must be distinguished by `ap_sighandler`:

No tbx call has been interrupted when receiving the signal:

All active transactions are aborted by tbx immediately. Upon return of `TbxInterrupt`, all active transactions will have been aborted. In this case, `ap_sighandler` is free to decide its subsequent actions.

A tbx call has been interrupted when receiving the signal:

All active transactions are marked for abortion by tbx but remain active for the moment. A signal is sent to all Transbase processes to which the application has been connected. Upon receipt of this signal, all those processes abort their current transaction and will return an error code, indicating that they have been aborted by a signal. Note that signals may be sent to a Transbase process from outside as well (e.g. by a kill command).

The error code `ABORT_SIGNAL` is delivered to the application program not upon return from `TbxInterrupt`, but from the active tbx call (delayed error code delivery).

In this case, `ap_sighandler` must return immediately into the active tbx call. The application program finally receives the (delayed) error code upon return of the active tbx call. After error code delivery, the application program is free to decide its subsequent actions, e.g. to do a "longjmp" (see also the example in the next section).

Since tbx does not handle any signals directly, there are no restrictions on application programs to use a predefined signal for asynchronous transaction abort nor is a signal reserved for tbx. The application program might even decide, to handle no signals at all, i.e. to exit the program on receipt of any signal. In that case, the Transbase processes will automatically abort their current transactions and exit, too. Thus, signal handling is necessary for an application only if the application wants to continue its operation upon receipt of a signal.

5.1 Asynchronous Abort: TbxInterrupt

Input Parameter: None

Result Parameter:

```
State state;
```

Call Syntax:

```
tbx (SIG_HANDLE, & state);
TbxInterrupt(&state);
```

Effect: May be called from within an application-defined signal handling procedure. For this call, to abort a TA which runs on a database on a machine m, the tbserver daemon must be active on m, otherwise an error is returned by this function.

Upon return of `TbxInterrupt`, state contains either `TA_ACTIVE` indicating that a tbx call has been interrupted or `TA_ABORTED` indicating that no tbx call has been interrupted and that all transactions have been aborted immediately.

In the first case (a tbx call has been interrupted) the signal handling procedure must return into the tbx call in order to avoid confusion of tbx. As a consequence of the `TbxInterrupt` call, this currently active tbx call will be aborted prematurely and will return the ABORTSIGNAL error code.

Errors:

    NO_ERROR

Example:

```
ap_sighandler()      /* interrupt routine */
{
    Id state;
    int error;
    signal (SIGINT, ap_sighandler);
    error = TbxInterrupt(& state);
        /* check error and notify if there is one */
    if (state == TA_ABORTED)
            /* do what ever you like, e.g. */
        longjmp (&jmpbuf, 0);
    else
        return;      /* otherwise confusion !! */
}

...

main()
{
    ...
    signal (SIGINT, ap_sighandler);
    setjmp (& jmpbuf);
        ...
    if ( TbxDml(...) == ABORTSIGNAL )
        longjmp (&jmpbuf, 0);
    else
        ...
}
```

Chapter 6

Error Codes and Messages

Each call of tbx returns an error code of type int where an error code of 0 means that no error has occurred, otherwise a detailed error code will be given.

Error Code	Meaning
0	No error
!= 0	error

If an error has been encountered, an additional comprehensive textual description (of type String) is available to the application program which explains the kind of error. This textual error description can be accessed via the external variable `tb_errtxt` held in the tbx module. The include file `tbx.h` contains the corresponding external declaration:

```
extern char * tb_errtxt;
```

The error handling is shown in full detail by the following example:

Example:

```
#include <stdio.h>
#include "tbx.h"

int   e;
Id    taid;
State tastat;

main()
{
    ...
    if (e = TbxCt(taid, &tastat)) errprint(e);
```

```
    ...
}

errprint (code)
int code;
{
    printf("ERROR: Code %d: %s\n", code, tb_errtxt);
}
```

All error information (error code as well as the corresponding textual error message available via `tb_errtxt`) is contained in the file `tberror.h` in the `TRANSBASE` directory. For each error, a symbolic constant (of type integer) as well as the textual error message is declared. It is also possible to include `tberror.h` in the application programs and check error codes via the symbolic constants and thus handle certain errors.

6.1 Hard and Soft Errors

Each error either is classified as a hard error or a soft error. In contrast to soft errors, a hard error indicates that a transaction has run on a severe failure and therefore cannot proceed. A hard error has the effect that the current transaction is aborted automatically by tbx.

Examples for hard errors are:

- lack of sufficient memory or disk resources to process a query
- integrity violations

If a tbx call returns a soft error, that call simply has no effect, i.e. the application program can proceed as if the call had not been made. Especially, the current transaction will not be aborted automatically.

Typical examples for soft errors are:

- syntax errors in a TB/SQL-query
- evaluation call of an inactive query
- type exceptions (overflow) in select queries

In summary, a soft error makes the current tbx call undone, a hard error makes the current transaction undone.

The tbx call `TbxGetTaState` is suitable to check if a transaction has been aborted by an error occurrence.

Appendix A

Transbase Interface Functions

```
Error _far _pascal TbxInterrupt (State _far *);
Error _far _pascal TbxConnect (char _far *, Id _far *);
Error _far _pascal TbxDisconnect (Id, State _far *);
Error _far _pascal TbxLogin (Id, char _far *, char _far *);
Error _far _pascal TbxBt (Id _far *);
Error _far _pascal TbxCt (Id, State _far *);
Error _far _pascal TbxAt (Id, State _far *);
Error _far _pascal TbxDml (Id, Id, char _far *, Query_descr _far *);
Error _far _pascal TbxRun (Id, Id, char _far *, Query_descr _far *, Result _far *);
Error _far _pascal TbxClose (Query_descr _far *);
Error _far _pascal TbxEval (Query_descr _far *, Result _far *);
Error _far _pascal TbxDelPos (Query_descr _far *,);
Error _far _pascal TbxUpdPos (Query_descr _far *, char _far *);
Error _far _pascal TbxGetDbState(Id, State _far *);
Error _far _pascal TbxGetTaState(Id, State _far *);
Error _far _pascal TbxQuState (Id, State _far *);
Error _far _pascal TbxSetTimeout(Timeout);
Error _far _pascal TbxSetDatDir(Id, char _far *);
Error _far _pascal TbxSetConsistency(int);
Error _far _pascal TbxVersion (Id, char _far *);
Error _far _pascal TbxContact (char _far *, Id _far *);
Error _far _pascal TbxAccept (Id);
Error _far _pascal TbxOpenStored(Id, Id, Id, Parameters _far *, Query_descr _far *);
Error _far _pascal TbxRunStored(Id, Id, Id, Parameters _far *, Query_descr _far *,    Re
Error _far _pascal TbxUpdPosStored(Query_descr _far *, Id, Parameters _far *);
Error _far _pascal TbxDelPosStored(Query_descr _far *, Id);
Error _far _pascal TbxStore (Id, char _far *, Id, char _far *, Query_descr _far *);
Error _far _pascal TbxDropStored(Id, Id);
```

```
Error _far _pascal TbxDropAllStored(Id);
Error _far _pascal TbxGetBlob (Id, Id, Blob _far *, Blobdesc _far *);
Error _far _pascal TbxMakeBlob(Id, Id, char _far *, Blobdesc _far *);
Error _far _pascal TbxSetSortOrder(Id, unsigned char _far *);
Error _far _pascal TbxGetSortOrder(Id, unsigned char _far *);
Error _far _pascal TbxTbmode (Id, char _far *);
Error _far _pascal TbxDumpReqStart(Id, d,)
Error _far _pascal TbxSendEvent(Id, char _far *, char _far *, char _far *);
Error _far _pascal TbxAbortCallBack(PTbxCallBack);
```

Appendix B

TBX Interface under MS/Windows

The following example gives a short overview about the realization of Transbase application under Windows. To keep the example simple, dynamic configurations, error tests etc. has been omitted.

We assume an underlying database 'Test' with a table 'Files', which contains a character field 'File'. All data (file names) have been inserted into the table.

The application has a menu with 3 entries:

Start: A query is started which inserts all tuples into a ListBox.

Break: Interrupts the processing of the query.

Exit: Ends the programs.

TBTEST.DEF

```
NAME      TBTEST
EXETYPE   WINDOWS
CODE      PRELOAD MOVEABLE DISCARDABLE
DATA      PRELOAD MOVEABLE MULTIPLE
HEAPSIZE  1024
EXPORTS
```

TBTEST.RC

```
#include <windows.h>
```

```
#include "tbtest.h"

IDR_MENU MENU DISCARDABLE
BEGIN
    MENUITEM "&Start!",  IDM_START
    MENUITEM "&Break!",  IDM_BREAK
    MENUITEM "&Exit!",   IDM_EXIT
END
```

TBTEST.H

```
#define MY_DATABASE     "Test"          // database name
#define MY_LOGIN        "tbadmin"       // user login
#define MY_PASSWORD     ""              // password
#define MY_QUERY "SELECT * FROM Files"  // query

#define IDR_MENU     100
#define IDM_START    101
#define IDM_BREAK    102
#define IDM_EXIT     103

#define IDC_RESULT   200

LONG PASCAL _export CALLBACK MyWndProc(HWND,UINT,WPARAM,LPARAM);
```

TBTEST.C

```
#include <windows.h>
#include <shellapi.h>
#include <tbx.h>          // Transbase include file
#include "tbtest.h"

HINSTANCE   hInst;        // instance handle
HWND        hWnd;         // window handle
HWND        hResult;      // listbox control handle

Error       eRc;          // error return code
Id          idDB,idTA;    // database/transaction id
State       stDB,stTA;    // database/transaction status
Query_descr qdQuery;      // query descriptor
Result      resResult;    // result structure
```

```
char    szBuf[256];      // string buffer
DWORD   dwIdx;           // count/index variable
BOOL    bAborted;        // abort flag

static LONG PASCAL MyError(Error eRc) // error handling routine
{
  if(!bAborted)          // if not within abort routine
  {
                         // display error return code
                         // and Transbase error message
    wsprintf(szBuf,"Error: %ld",(long)eRc);
    MessageBox(NULL,tb_errtxt,szBuf,MB_ICONSTOP);
  }
  return eRc;
}

static BOOL PASCAL Init(VOID)
{
  WNDCLASS wc;
  memset(&wc,0x00,sizeof(WNDCLASS));
  wc.lpszClassName = MY_DATABASE;
  wc.hInstance  = hInst;
  wc.lpfnWndProc  = MyWndProc;
  wc.hbrBackground = COLOR_WINDOW+1;
  wc.hCursor   = LoadCursor(NULL,IDC_ARROW);
  wc.lpszMenuName = MAKEINTRESOURCE(IDR_MENU);
  if(!RegisterClass(&wc)) // register window class
    return 0;

  hWnd=CreateWindow(MY_DATABASE,MY_DATABASE,
   WS_OVERLAPPEDWINDOW|WS_VISIBLE,
   CW_USEDEFAULT,CW_USEDEFAULT,200,400,
   NULL,NULL,hInst,NULL); // create window
  return hWnd && hResult;
}


int PASCAL WinMain
(HANDLE hInstNew,HANDLE hInstPrev,LPSTR lpszCmdLine,int iCmdShow)
{
  MSG msg;
  if(hInstPrev) return 0;
  hInst=hInstNew;    // save instance handle
```

```
  if(!Init() || eRc) // init application
    return 0;

  while(GetMessage(&msg,0,0,0)) // run message loop
     {
       DispatchMessage(&msg);
       TranslateMessage(&msg);
     }
  return msg.wParam;
}


LONG PASCAL _export CALLBACK MyWndProc
(HWND hw, UINT iMsg, WPARAM uP1, LPARAM lP2)
{
  switch(iMsg)
    {
    case WM_CREATE:  // creation of window
      {
      RECT r;
      GetClientRect(hw,&r);
      hResult=CreateWindow("LISTBOX","Result",
        WS_CHILD|WS_VISIBLE|WS_THICKFRAME|WS_CAPTION|WS_VSCROLL|LBS_NOTIFY,
        0,0,r.right,r.bottom,
        hw,IDC_RESULT,hInst,NULL);        // create listbox

      if((eRc=TbxConnect(MY_DATABASE,&idDB)) && MyError(eRc))
              // connect to database
        break;
      if((eRc=TbxLogin(idDB,MY_LOGIN,MY_PASSWORD)) && MyError(eRc))
              // login into to db
        break;
      }
      break;

    case WM_COMMAND: // menu handling
      switch(uP1)
      {
        case IDM_START: // menuitem: start
         {
          SetWindowText(hResult,NULL); // reset listbox
          SendMessage(hResult,LB_RESETCONTENT,0,0L);
          if((eRc=TbxBt(&idTA)) && MyError(eRc))
```

```
        // begin transaction
    break;

        // compile query
    if((eRc=TbxDml(idDB,idTA,MY_QUERY,&qdQuery)) && MyError(eRc))
    break;

    if((eRc=TbxEval(&qdQuery,&resResult)) && MyError(eRc))
        // eval query, i.e.
        //  fetch first tuple
    dwIdx=0;
    while(!eRc && qdQuery.eod!=NO_TUPLE) // tuple(s) exist
    {
      if(qdQuery.field[0].fieldpointer) // field is not NULL
      {
                 // display counter
         SetWindowText(hResult,itoa((int)dwIdx++,szBuf,10));
         SendMessage(hResult,LB_ADDSTRING,0,
             (LPARAM)(LPCSTR)qdQuery.field[0].fieldpointer);
         UpdateWindow(hResult);
      }

      if(eRc=TbxEval(&qdQuery,&resResult)) // fetch next tuple
         MyError(eRc);
    }
    TbxClose(&qdQuery); // close query
    TbxCt(idTA,&stTA);  // commit transaction
  }
  break;

  case IDM_BREAK:          // menuitem: break
      TbxInterrupt(&stDB); //   send interrupt
      bAborted=TRUE;
      MessageBeep(MB_ICONEXCLAMATION);
      break;

 case IDM_EXIT:            // menuitem: exit
     TbxInterrupt(&stDB);  //   send interrupt
     bAborted=TRUE;
     SendMessage(hw,WM_DESTROY,0,0L); // force exiting
     break;

 case IDC_RESULT:          // listbox selection
     if(HIWORD(lP2)==LBN_DBLCLK) // by double click
```

```
            if(LB_ERR!=(dwIdx=SendMessage(hResult,LB_GETCURSEL,0,0L)))
            if(LB_ERR!=SendMessage(hResult,LB_GETTEXT,
                   (WPARAM)dwIdx,(LPARAM)(LPSTR)szBuf)) // get file name
                ShellExecute(hw,"open",szBuf,NULL,NULL,SW_SHOW); // execute file
            break;
    }
    break;
    case WM_DESTROY:              // destroying window
      TbxDisconnect(idDB,&stDB);  // disconnect from db
      PostQuitMessage(0);         // exit message loop
      break;

  default:
     return DefWindowProc(hw,iMsg,uP1,lP2);
  }
  return 0L;
}
```

TBTEST.MAK

```
# simple Makefile for NMAKE (Microsoft)

TRANSBASE = D:\TB\TB4111
TB_INC = $(TRANSBASE)\TBX.H
TB_LIB = $(TRANSBASE)\TBX.LIB

CFLAGS = /G2 /W3 /AM /GA
LFLAGS = /NOD /STACK:50000
LIBS = oldnames libw mlibcew shell.lib

ALL:  TBTEST.EXE

TBTEST.RES: TBTEST.RC TBTEST.H
    rc /r TBTEST.RC

TBTEST.OBJ: TBTEST.C TBTEST.H $(TB_INC)
    cl $(CFLAGS) /c TBTEST.C

TBTEST.EXE: TBTEST.OBJ TBTEST.DEF $(TB_LIB) TBTEST.RES
    link $(LFLAGS) TBTEST,,,$(LIBS) $(TB_LIB),TBTEST.DEF
    rc TBTEST.RES
```