

Transbase®
Embedded SQL

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	Introduction	3
2	Basic Concepts	5
2.1	ESQL Statements	5
2.2	Databases and User Authorization	6
2.2.1	Connecting to a Database	6
2.2.2	Login to a Database	6
2.2.3	States of a Database Connection	6
2.3	Transactions	7
2.3.1	Single Transaction Programs	7
2.3.2	Multiple Transaction Programs	8
2.3.3	States of a Transaction	9
2.4	Cursor	11
2.4.1	Cursor States and Operations	11
2.4.2	Updatable Cursors	11
2.4.3	Scrolling Cursor	12
2.4.4	Scope of Cursor Names	12
2.5	Host Variables	12
2.5.1	Declare Section for Host Variables, Block Structure	12
2.5.2	Input, Output and Statement Variables	13
2.5.3	Semantics of Host Variables	13
2.5.4	Syntax of Host Variables	13
2.5.5	Indicator Variables	14

2.6	SQL Types and Host Variable Types	15
2.6.1	Type Correspondences	15
2.6.2	Assignment of Field Values to Host Variables	15
2.6.2.1	Assignment of Arithmetic Field Values	15
2.6.2.2	Assignment of character String Field Values	16
2.6.2.3	Assignment of Logical Field Values	17
2.6.2.4	Assignment of Time Field Values	17
2.6.2.5	Supplying Values by Input Host Variables	17
2.6.2.6	Supplying Arithmetic Values	17
2.6.2.7	Supplying character Strings	17
2.6.2.8	Supplying Logical Values	18
2.6.2.9	Supplying Time Values	18
2.7	Stored Queries	18
3	The SQL Communication Area and Error Treatment	20
3.1	The SQLCA Structure	20
3.1.1	SQLCODE	21
3.1.2	SQLERRD	21
3.1.3	SQLWARN flags	22
3.1.4	taid and tastate	22
3.1.5	dbid and dbstate	23
3.1.6	Error Treatment	23
3.1.7	Hard and Soft SQLERRORS	24
3.2	The WHENEVER Directive	25
4	Syntax of ESQL Statements	27
4.1	ESQL Statement, EXEC SQL, Comment	27
4.2	Host-Var, Host-Var-Ind	28
4.3	Summary of ESQL-Statements	28
4.4	Embedded-TBSQL-Statement	29
4.5	Include-SQLCA-Statement	31
4.6	Declare-Section-Statement	32

4.7	Declare-Cursor-Statement	33
4.8	Whenever-Statement	35
4.9	Connect-Statement	36
4.10	Disconnect-Statement	38
4.11	Login-Statement	39
4.12	Begin-Work-Statement	40
4.13	Commit-Work-Statement	40
4.14	Rollback-Statement	41
4.15	Open-Statement	41
4.16	Fetch-Statement	42
4.17	Delete-Positioned-Statement	43
4.18	Update-Positioned-Statement	44
4.19	Close-Statement	45
4.20	Select-Into-Statement	46
4.21	Use-Transaction-Statement	47
4.22	Use-Database-Statement	48
4.23	Set-Sortorder-Statement	49
4.24	Get-Sortorder-Statement	50
4.25	Set-Data-Dir-Statement	50
4.26	Set-Timeout-Statement	51
4.27	Set-Consistency-Statement	52
4.28	Send-Interrupt-Statement	53
5	Retrieving and Inserting BLOB Objects	54
5.1	Using Blobdesc as Host Variable	54
5.2	Fetching BLOB Objects in TB/ESQL	55
5.3	Inserting and Updating BLOB Objects in TB/ESQL	57
5.4	BLOBs in Dynamic TB/ESQL	58

6	Dynamic ESQL Statements	59
6.1	Dynamic Cursors and Statement Variables	59
6.1.1	Runtime Structure of a Cursor	60
6.1.2	The Datatype Query_Descr	61
6.1.3	Reading Field Values of a Dynamic Cursor	63
6.2	Dynamic Executable ESQL Statements	65
6.2.1	Restrictions and their Remedies	66
6.2.2	Control Information for Dynamic Statements	67
6.2.2.1	The sqlerrd[0] Flag	67
6.2.2.2	The sqlerrd[1] Flag	67
7	The ESQL Text Expander	68
8	Signal Handling : Send-Interrupt-Statement	70
9	Modularization of ESQL programs	73
9.1	ESQL Modules	73
9.2	Mixture of ESQL and TBX Calls	73
10	Compilation of an ESQL Program	75
10.1	Command Syntax	75
10.1.1	Sample Makefile	76
10.2	Command Syntax on Windows Platforms	77
10.2.1	Sample Makefile	78
11	Debugging an ESQL Source	79
12	Appendix A: Sample Programs	80
13	Appendix B: Database Schema SAMPLE	84

Chapter 1

Introduction

This manual describes the embedding of TB/SQL in a C language environment (TB/ESQL). TB/SQL is a proper superset of the ISO/ANSI SQL database language.

TB/ESQL meets the requirements of the X-OPEN SQL Portability Guide. Beyond that standard, TB/ESQL offers many additional features such as dynamic queries. The full functionality of Transbase as described in the TB/SQL Reference Manual and the Programming Interface TBX Manual is available at the TB/ESQL interface. This also includes distributed transactions with atomic 2-phase-commit.

This TB/ESQL Manual is strongly related to the TB/SQL Reference Manual. The latter describes the SQL language as it is used for example in an interactive environment such as TBI. TB/SQL statements appear as subunits in TB/ESQL statements. Therefore the definitions of TB/ESQL statements within this manual often refer to the TB/SQL Reference Manual.

There is also a relationship to the lower C programming language interface of Transbase, namely the TBX-Interface. TBX as well as ESQL serve to write application programs accessing Transbase databases. Both TBX and ESQL offer a C environment and special constructs to interface with Transbase databases. In TBX, this interface is provided by a special C function (tbx) in a library. Thus TBX programs are pure C programs. In ESQL, this interface is provided by so called ESQL statements which syntactically are not compatible with C.

For the application programmer, TBX and ESQL are independent from each other although from a functional point of view they are not essentially different. Each ESQL program can also be written as a program which uses the TBX interface. In fact, an ESQL program is mapped by the ESQL precompiler onto an equivalent TBX program.

As far as ease of use is concerned, ESQL is the higher level interface. ESQL programs are much easier to write and automatically incorporate a lot of error checking mechanisms (type overflow checking in program variable assignment, database

exception handling, etc.). On the other hand, ESQL is slightly more static in query declarations. There are only few applications, namely highly dynamic ones, that would be hard or unfeasible to write as ESQL programs. One example is an environment where the result of one query triggers additional queries which run in parallel: while one could simply generate further queries in a TBX program, all cursor names must be statically declared in ESQL programs. Modularization of large application programs is also easier with TBX.

ESQL as well as TBX programs hide all communication details between the program and the Transbase kernel. It is thus transparent to the program whether Transbase is linked in or runs as a separate process on the same machine or even on a remote host. In any case, the application programmer sees the interface; as described by this manual.

Chapter 2

Basic Concepts

2.1 ESQL Statements

An ESQL program is a sequence of C statements and so called ESQL statements. ESQL statements for example serve to incorporate TB/SQL statements into the program (such as `SELECT-`, `INSERT-`, `UPDATE-`, `DELETE-`, `CREATE-`, `DROP-`, `LOCK-`, `UNLOCK` statements). Each TB/SQL statement can be incorporated and formulated as an ESQL statement by prefixing it with the words `EXEC SQL` and terminating it with a semicolon.

Additionally, there are ESQL statements that have no analogon in TB/SQL, but correspond to TBX function calls:

- statements to `CONNECT` and to `LOGIN` into a database;
- transaction control statements (`COMMIT`, `ROLLBACK`);
- statements to `OPEN` or `CLOSE` a TB/SQL retrieval query;
- statements to `DELETE` or `UPDATE` the current tuple of a retrieval query;
- statements to control errors, signals and exceptions.

A third class of ESQL statements have no correspondence either in TB/SQL or in TBX function calls, namely those statements that control the automatic transfer of database data into program variables:

- statements to `DECLARE` cursors for retrieval queries;
- the `FETCH INTO` statement to transfer the field values of result tuples into host variables;
- statements to embed program variable declarations thus making them usable as host variables.

2.2 Databases and User Authorization

2.2.1 Connecting to a Database

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 it by the Connect -Statement. As a result of a successful Connect-Statement the application program is delivered a database identifier and this database then is the **current database**. Each ESQL statement semantically referring to a database runs against the current database. In order to switch between several connected databases, the program may remember the database identifiers in host variables. To make a connected database the current one, a Use-Database-Statement with the appropriate database identifier is provided.

In **CONNECT** operations, databases are named 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 resides on the local host, it may simply be identified by the string '**ldb**'. Note that programs that specify databases by the complete name can be ported to other machines without changing them, whereas programs that specify the logical name only must be modified to make them portable.

A connection to a database is terminated by an explicit Disconnect-Statement.

It is legal to connect to a database which is already connected. This simply has no effect.

2.2.2 Login to a Database

After having connected to a database the application program has to **login** into the database. This is done by the Login-Statement where user identification (the user name) and user authorization (the password) are specified. Note that the authorization procedure has to be performed for **each** database connected to. After connection but before authorization the application is not logged in and thus has no privileges to use the database.

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

2.2.3 States of a Database Connection

With respect to a database identifier the following states of a connection are known to an ESQL program:

DB_DCONN:	no connection with this database identifier
DB_CONN:	connected, not logged in;
DB_LOGGED:	connected, logged in;
DB_KILLED:	broken connection (the database kernel process has been killed).

When a connection breaks, a potentially active transaction is automatically aborted. The application program can resume operation by issuing a new database connection and starting a new transaction.

2.3 Transactions

An application program can run one or more **transactions** against databases. Each ESQL database statement automatically runs inside a transaction. A transaction is the unit of consistency. Each transaction can either be committed or aborted by the application program (Commit-Work-Statement , Rollback-Work-Statement).

If an application program has active at most one transaction at a time it is called a single transaction program. Under certain restrictions, Transbase offers the possibility to have active more than a single transaction at a time. However, those programs (called multiple transaction programs) are slightly more complicated if written in Embedded SQL.

Transbase allows transactions being distributed over more than a single database; those transactions are called distributed transactions.

2.3.1 Single Transaction Programs

A transaction can be explicitly started by the application (Begin-Work-Statement) or is implicitly started whenever a ESQL database statement is run and there exists no current active transaction. Any 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 never shown to other transactions). A rollback undoes all effects of the 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 or in case of crashes by a built-in 2-phase-commit protocol.

Note that if an application program does not commit its transaction, but simply leaves or exits the program, the transaction will be automatically aborted by Transbase.

A transaction consists of a sequence of ESQL statements. Each statement addresses at most one of the connected databases. However, more than one database can be addressed within a transaction by different statements.

Recall that the application program has both to connect and login to the database addressed before any ESQL database statement against this database can be issued. Connecting to a database and start of a transaction, however, are independent actions for the application program. Thus it is possible to connect first and then (explicitly or implicitly) to start a transaction and it is possible to explicitly start a transaction first and then to connect to a database. In any case, if an active transaction is to be extended on another database, the program must simply connect to it if not already done and issue the desired ESQL statements against it.

Note carefully that a further explicit start of a transaction after connecting to the second database not only is unnecessary but would have a completely different effect namely the start of a second independent transaction.

2.3.2 Multiple Transaction Programs

As a unique feature of TB/ESQL, a program may have more than one transaction active in parallel. Each ESQL database and transaction control statement (COMMIT/ROLLBACK) exactly refers to one transaction. Whereas there are no restrictions in how Transactions may be distributed over databases, there is the following restriction for multiple transaction programs:

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 different databases.

If the program wants to have more than one transaction active at a time, each of these transactions must be started with the explicit Begin-Work-Statement. In addition, after starting a transaction the program must keep track of its **transaction identifier** which can be read into a host variable. After starting a transaction, this transaction is the current transaction. Each ESQL database and transaction control statement refers to the **current transaction**. Analogously to the

concept of current database, a Use-Transaction-Statement supplied with a valid transaction identifier makes the transaction the current one and thus enables the application program to switch between different transactions.

Note carefully that by this mechanism there is at most one transaction the current one even if several transactions are active. Initially no current transaction exists. Whenever a transaction has been finished, again no current transaction exists even if other transactions are still active. If in this situation a Use-Transaction-Statement with a valid transaction identifier is executed, then a current transaction exists. Recall that an implicit start of a new transaction occurs if an ESQL database statement is executed and no current transaction exists. Thus there is a slight danger that a new transaction is started inadvertently namely if one transaction is finished and a Use-Transaction-Statement to switch to another still active transaction is forgotten.

Note that transaction identifiers and explicit start of a transaction are of no concern if the application only runs one transaction at a time. Thus, for this simple but frequent case, an ESQL program is in no way more complicated than an equivalent program written in the standardized X-OPEN fashion.

2.3.3 States of a Transaction

With respect to a transaction identifier the following states of a transaction are known to an ESQL program:

TA_NOT_ACTIVE:	no transaction with that identifier exists or has existed during the programs life.
TA_ACTIVE:	a transaction with that identifier exists and has not yet ended (i.e. not yet committed or aborted).
TA_COMMITTED:	this transaction has been committed.
TA_ABORTED:	this transaction has been aborted. Note that any ESQL statement can lead to this state in case of a severe error, e.g. if communication problems arise.
TA_UNDEF:	a commit procedure has been tried for that transaction but due to communication failures it cannot be determined if the commit has been performed by the database kernel. In these very rare cases, the ESQL program cannot determine by any direct information services if the transaction has succeeded.

The transitions can be described by the following picture. Note that TA start may be an explicit start (`BEGIN WORK`) or an implicit start.

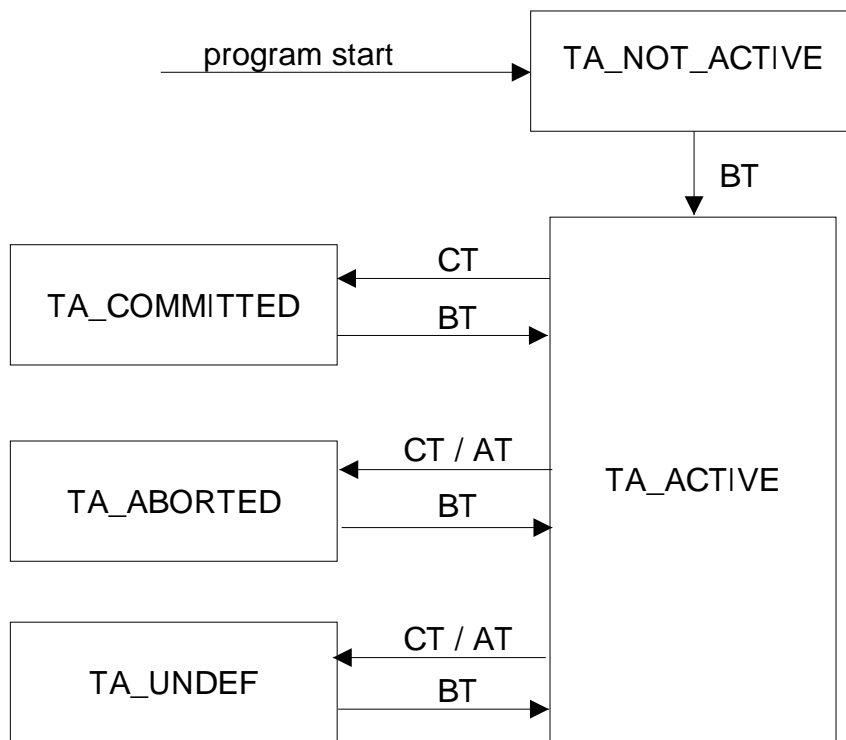


Figure 2.1: Transaction States

2.4 Cursor

Sequential programs must be provided a way to retrieve a query result (a set of tuples) in a tuple-at-a-time fashion. The **cursor** is the concept for a host program to declare retrieval queries (formulated by a TB/SQL Select Statement) and to fetch the field values of the result tuples into host variables one tuple at a time. A cursor is declared by a Declare-Cursor-Statement which attaches a program-wide unique name (the cursor name) to a Select-Statement. All ESQL statements referring to a cursor must identify the cursor by its name.

2.4.1 Cursor States and Operations

A cursor has a state which is either **open** or **closed**. Initially a cursor is in state closed. An Open-Cursor-Statement opens a closed cursor, a Close-Cursor-Statement closes an open cursor. It is illegal (and produces a TB/ESQL error) to open an open cursor or to close a closed cursor.

An open cursor has a position which is "before a tuple", "on a tuple" or "behind the last tuple". If the position is on a tuple, this tuple is called the **current tuple**. After opening the cursor, its initial position is before the first tuple of the result set of the Select-Statement. A Fetch-Statement advances the position of the cursor to the next result tuple if it exists and delivers its fields into host variables, otherwise the position is set behind the last tuple and the host variables remain unchanged. Further Fetch-Statements change neither the cursor position nor any host variables, i.e. they have no effect.

A cursor can be closed independently from its position. It is not necessary to retrieve all tuples of a result before closing the cursor.

When a transaction is finished, all cursors which are still open are automatically closed.

2.4.2 Updatable Cursors

If the cursor is on a tuple and the corresponding Select-Statement is updatable and has the **FOR UPDATE** clause specified (see TB/SQL Reference Manual) then a Delete-Positioned-Statement or an Update-Positioned-Statement can be issued. The Delete-Positioned-Statement deletes the tuple from the database table from which the current tuple is derived. The cursor position is set either "before the next result tuple" or (if the deleted tuple was the last result tuple) "after the last tuple".

An Update-Positioned-Statement updates the tuple in the database table from which the current tuple is derived and the cursor position remains unchanged.

2.4.3 Scrolling Cursor

By default, a cursor provides for sequential FETCHing the result tuples. If the Cursor has been declared with the SCROLL option then the FETCH operation allows absolute and relative positioning by specifying a result set position. The combination of SCROLL cursor with FOR UPDATE clause is not permitted.

Note that scrolling cursors produce an overhead because the kernel internally stores all result tuples fetched so far.

2.4.4 Scope of Cursor Names

The names of all declared cursors (and all declared host variables) in an ESQL compilation unit must be disjoint. The declaration of a cursor must textually precede its usage. A Declare-Cursor-Statement is allowed on all places where a C variable definition is allowed. A cursor declared in a function body is usable inside that function, and a cursor declared outside any function is usable in all functions of that program. For ESQL programs consisting of more than one modules see also chapter "Modularization of ESQL Programs".

2.5 Host Variables

2.5.1 Declare Section for Host Variables, Block Structure

Host variables are C program variables which are referenced in ESQL statements. Their types are restricted to basic C types such as char, int, etc., character arrays, character pointers and certain additional predefined types which are supported by the ESQL environment. All ESQL host variables are defined or declared in normal C syntax but their declaration must be indicated by the Declare-Section-Statement (this enables the ESQL Compiler to assemble the types of the host variables). For convenience this program part is called a **declare section**. Only host variable definitions are allowed in a declare section. All C storage classes are allowed (extern, static, automatic). In contrast to variable declaration in C, host variables *must not be initialized statically*.

Arbitrarily many declare sections are permitted in an ESQL program. Textually, a declare section may appear where a C variable or parameter declaration is legal. The names and scope of host variables are subject to the C scope rules, i.e. the **ESQL Compiler recognizes the block structure** of the ESQL program provided that the block structure does not depend on the #include files of the program. A host variable declared outside any function is usable in each function, a host variable declared in a function body or as formal parameter of a function is only usable in that function. For ESQL programs consisting of more than one module see also the chapter "Modularization of ESQL Programs".

2.5.2 Input, Output and Statement Variables

Each occurrence of a host variable in an ESQL statement either delivers input to the statement or serves to store output delivered by the statement. Output is delivered to host variables by the Fetch-Into-Statement and the (one tuple delivering) Select-Into-Statement. The INTO-clauses consist of a list of host variables into which the field values of the fetched tuple are stored. Other statements have host variables delivering input to the statement. In the sequel, occurrences of host variables are also simply called **input variables** or **output variables** (although the property of delivering or storing information is not attached to a host variable but to a special occurrence of it).

In particular, an embedded TB/SQL statement can have one or more input variables. **Input variables** in an embedded TB/SQL statement are allowed on each position where a **Simple_Primary** is allowed (see TB/SQL Reference Manual). This enables query parameterization on value level. A query of that kind is therefore called **parameterized query**.

A host variable may also replace a whole TB/SQL statement. This enables the application program to construct and execute a statement at runtime. An ESQL statement of this kind is called a **dynamic ESQL statement** and a host variable of this kind is called a **statement variable**. Note that a statement variable must be of type `char*`, `char[]`, or `String`.

2.5.3 Semantics of Host Variables

The semantics of input and statement variables is the following: whenever the execution of the program reaches an ESQL statement with input variables or a statement variable, then the variables are textually replaced by their current value before the statement is executed. Note that this mechanism only describes the semantics and not necessarily the actual processing.

An important exception to the above replacement rule must be kept in mind for the Declare-Cursor-Statement and Open-Cursor-Statement: whenever execution flow reaches an Open-Cursor-Statement, then all input variables or the statement variable in the corresponding Declare-Cursor-Statement are replaced and the cursor on the thus obtained query is opened. Another exception is the usage of host variables of type Blobdesc for transmission of BLOBs (binary large objects, see the chapter "Retrieving and Inserting BLOB Objects").

2.5.4 Syntax of Host Variables

In contrast to the declaration of host variables in the declare section which corresponds to normal C syntax the usage of a host variable in an ESQL statement

must be indicated by prefixing it with a colon (:). This serves to distinguish a host variable from an SQL field identifier. Prefixing with a colon also consistently is required for output host variables (although no field identifiers can appear on places where an output host variable is legal). A statement variable must be prefixed by a # when used inside a statement.

2.5.5 Indicator Variables

Indicator variables are special variables that serve to attach additional information to host variables. An indicator variable is a program variable of C-type short (signed two byte integer) and has to be declared within a Declare Section.

An indicator variable can be associated to the occurrences of input and output variables (but not to statement variables). Syntactically, the indicator variable is written immediately behind the host variable and also is prefixed with a colon. Indicator variables serve to describe SQL NULL values and to obtain information about truncated character strings as described in the sequel.

If an input variable has an attached indicator variable then the evaluation of the input variable proceeds as follows. The indicator variable is interpreted first. If its value is not equal to -1 then it is ignored (i.e. the value of the input variable is taken as the actual value). Otherwise the input variable evaluates to the constant NULL value of SQL (i.e. the input and indicator variable semantically are replaced by the SQL keyword NULL).

If an output variable has an attached indicator variable then the assignment of a field value proceeds as follows. If the field value is the NULL value then the indicator variable is set to -1 and the host variable remains unchanged; otherwise the indicator variable is set to a value different from -1 (see below) and the host variable is set to the field value (after type adaptations if necessary). Note that if a field value is NULL and no indicator variable is specified, then an error is reported.

If an output variable has an attached indicator variable and a field value to be assigned is not the NULL value then the indicator variable is always set to 0 except in the case of **string truncation**. A string truncation occurs if the field value is a character string (CHAR(p) or CHAR(*) or BINCHAR(p) or BINCHAR(*)) and the host variable is declared as character array with known size s (i.e. no char pointer, no extern array) and s is less or equal to the actual length of the field value. In this case the first s-1 characters and a zero byte is stored in the host variable and the indicator variable is set to the original length of the field value.

2.6 SQL Types and Host Variable Types

2.6.1 Type Correspondences

TB/SQL and the C language each support their own data types. SQL types such as TINYINT, SMALLINT, INTEGER, BIGINT, CHAR(p) directly correspond to basic C types (signed char, short, Int4, Int8, char[p+1], resp.). The SQL types NUMERIC, NUMBER, BINCHAR, BITSS, BITSS2, DATETIME and TIMESPAN do not have corresponding basic C types but are represented by C-structures. They can be manipulated by library functions.

Each input and output variable which is used in an ESQL statement must have a type which matches its usage. For many but not all SQL types, ESQL offers type notations which resemble those of SQL as well as pure C notations. For example, host variables which serve to store field values of types DATETIME or TIMESPAN must be declared with type Datetime[X:Y] or Timespan[X:Y], resp., whereby X and Y are ranges from YY to MS as used in the corresponding SQL type definitions. Note that these type names as all names in C are case sensitive. There is no basic C type available which is suited to exactly store time values. A counterexample is the SQL type INTEGER. A corresponding host variable can either be declared with type Int4 or with type Integer (note that the latter notation resembles the SQL type but is case sensitive in an ESQL program). Of course the type correspondences (between Int4 and Integer in our example) must be known by the ESQL programmer in order to correctly process the values assigned to his host variables. Table 2 shows the type correspondences between SQL and C types.

The types in the rows TINYINT through DOUBLE in Table 2 are called the arithmetic types. The types CHAR(p), CHAR(*), BINCHAR(p), BINCHAR(*), BITS(p), BITS(*) are called the character string types. The type BOOL is called the logical type. The types DATETIME and TIMESPAN are called the time types.

2.6.2 Assignment of Field Values to Host Variables

Assignment of field values to host variables occurs in the Fetch-Statement and in the Select-Into-Statement. The following rules apply depending on the type of fields and host variables.

2.6.2.1 Assignment of Arithmetic Field Values

For each arithmetic type (TINYINT through DOUBLE) the corresponding type of the host variable according to Table 2 guarantees that the field value is exactly representable (no truncation or rounding) and that no overflow occurs (loss of significant digits). However, for each arithmetic type the ESQL programmer may also choose each of the arithmetic host variable types and even also other basic

arithmetic C types which are not listed in the table: `char`, `short`, `int`, `long`, each of them qualified as signed or unsigned, in any valid C syntax variation (typenamees defined via

`#define` or `typedef` however are not permitted). Fractional parts of field values then may be truncated as needed but it is still assured that no significant digits or the sign is undetectedly lost (an error is reported by the ESQL runtime system in these cases).

2.6.2.2 Assignment of character String Field Values

For the assignment of field values of type `CHAR(p)` or `CHAR(*)` the following considerations are of importance: A host variable of type `String` is always appropriate but requires much space (`String` is a character array of size `MAXSTRINGSIZE` which is approximately the maximum tuple size). A host variable of type `char[p+1]` is appropriate for all field values with maximum length `p` (a zero byte is added as last character). For longer fields truncation occurs, i.e. only the first `p` characters plus a zero byte are transferred (see also the chapter "Indicator Variables"). Thus it is assured that the space of the host variable is not exceeded. Shorter fields are transferred with their original length plus a zero byte.

For a host variable of type `char*` it is assumed that at the time of field assignment the variable points to a memory region large enough to store the field value plus a zero byte. Of course this cannot be controlled at runtime by the ESQL environment. Analogous rules apply to host variables which are declared as `char[]` with storage class `extern`. Note that these are the only two cases where field value assignment to host variables may undetectedly cause damage of program data.

The assignment of a field variable of type `BINCHAR(*)` or `BINCHAR(p)` can be made as follows: if a host variable of type `Binchar(*)` is used (let its name be `b`) then the assignment always works without loss of data but the access to that type must be done via the following two macros:

`BINCHARLEN(&b)` delivers the length of the value measured in bytes.

`BINCHARARR(&b)` delivers the address of the first byte of the `BINCHAR` value; both macros must be called with the address of a `BINCHAR` variable.

`BINCHAR` values can also be assigned onto host variables of type `char*` or `char[p]`. In this case the value is terminated with a zero byte. Note that if there were zero bytes inside the `BINCHAR` value then the normal C string processing functions would recognize the first zero byte as the end of the string.

Field variables of type `BITS(*)` or `BITS(p)` are assigned to host variables of type `Bits(*)`. The access to that type must be done via the following two macros:

`BITSLEN(&b)` delivers the logical length of the value measured in bytes.

`BITSARR(&b)` delivers the address of the first byte of the `BITS` value; both macros must be called with the address of a `Bits` variable.

2.6.2.3 Assignment of Logical Field Values

For the assignment of logical field values (`BOOL`), host variables of all the arithmetic types mentioned in chapter "Assignment of Arithmetic Field Values" are appropriate. If the field value is `TRUE` the value 1 is assigned else the value 0.

2.6.2.4 Assignment of Time Field Values

For the assignment of time field values (`DATETIME` and `TIMESPAN`), host variables of type `Datetime[X:Y]` and `Timespan[X:Y]`, resp., are appropriate. For example, a host variable could be declared as `Datetime[YY:DD]`. In effect, the database values are casted to the exact type of the target host variable. Library functions for output (conversion to textual representation) as well as for complex computations are provided in the library `tbx.a`. All functions are described in the chapter "Routines for Datetime and Timespan".

2.6.2.5 Supplying Values by Input Host Variables

The following paragraphs describe how the values of input host variables are interpreted. The following rules apply depending on the type of host variables.

2.6.2.6 Supplying Arithmetic Values

To supply an arithmetic value to a query via a host variable all arithmetic types mentioned in chapter "Assignment of Arithmetic Field Values" are appropriate. Note that in particular also the type `char` is interpreted arithmetically.

2.6.2.7 Supplying character Strings

To supply character strings each of the four corresponding host variable types in Table 2 is appropriate. The value of the host variable evaluates to its character sequence up to the first zero byte. Note that although string literals in `TB/SQL` and in `ESQL` statements must be enclosed in single quotes (e.g. `'Hello'`) this does not hold for the character sequence supplied by the host variable. In fact if the host variable's string contained enclosing quotes these quotes would become part of the supplied string. On the other hand, if a string containing a quote is to be supplied it must not be written twice as in the `TB/SQL` string literal representation.

To supply values of type `BINCHAR`, also the host variable type `BINCHAR` can be used; the value must be built up using the macros described in "Assignment of character String Field Values".

2.6.2.8 Supplying Logical Values

To supply a logical value (TRUE, FALSE) only a host variable declared with type Bool is appropriate. Note that although Bool internally is mapped on char and can be used like char by the programmer in C statements, the type definition of Bool tells the ESQL precompiler that the value of the variable is not to be interpreted arithmetically. A value of 0 then evaluates to FALSE, each other value evaluates to TRUE.

2.6.2.9 Supplying Time Values

To supply time values (DATETIME and TIMESPAN), host variables of type Datetime[X:Y] and Timespan[X:Y], resp., are appropriate (the notation [X:Y] symbolically denotes the range). For example, the values of those host variables may have been assigned by their usage as output host variables. It is also possible to explicitly build up values in host variables - this is also described in the chapter "Routines for Datetime and Timespan".

2.7 Stored Queries

A certain class of ESQL statements consists of so called *storable* statements. The following query types are storable: Select-Into-Statement, Declare-Cursor-Statement, Insert-Statement, Update-Statement, Delete-Statement, Delete-Positioned-Statement, Update-Positioned-Statement. Storing a query means that a query is compiled once and can be executed many times without recompiling it. A stored query exists until the application ends and is then not retained in any form. Note that storing a query influences the performance only but not the functionality of the query. If a query is executed more than once then storing pays off else storing produces an undesirable overhead.

ESQL automatically takes advantage of query storing but the ESQL programmer has the means to override the default behaviour. The ESQL rule is that a query (of the above described class) is executed via the storing mechanism if and only if it contains input host variables (parameterized query). Normally, this default behaviour is quite reasonable, however if the programmer does not want the parameterized statement to be stored or if a parameterless statement is to be stored then the statement can be specified with a Store-Clause. The syntax of the Store-Clause is described in detail in the corresponding chapters.

TB/SQL type (case insensitive)	corresponding type notation for host variables (case sensitive)
TINYINT	Tinyint char
SMALLINT	Smallint short
INTEGER	Integer Int4
BIGINT	Bigint Int8
NUMBER	Number
NUMERIC NUMERIC(p) NUMERIC(p,s)	Numeric Numeric(p) Numeric(p,s)
FLOAT	Float float
DOUBLE	Double double
CHAR(p)	CHAR[p+1]
CHAR(*)	char* String extern char[]
BINCHAR(p) BINCHAR(*)	BINCHAR(*)
BITS(p)	Bits(*) BITS(*)
BOOL	Bool
DATETIME[X:Y]	Datetime[X:Y]
TIMESPAN[X:Y]	Timespan[X:Y]
BLOB	Blobdesc

Table 2.1: Type Correspondences

Chapter 3

The SQL Communication Area and Error Treatment

Each executable ESQL statement produces return information at runtime which describes whether the statement has completely succeeded or a kind of exception has occurred. For ease of discussion, the following names for different classes of results are introduced (note that these names are not available within the ESQL program):

SQLSUCCESS:	The statement executed successfully.
SQLWARNING:	The statement executed with restricted success. This means that the program safely can continue without interpreting the exception but it might be useful to interpret it.
SQLERROR:	The statement did not execute successfully. If the program continues without noticing the error it may in general produce further errors or unpredictable results.
NOT FOUND:	This event occurs if a statement executed successfully but one of the following conditions holds. A Fetch-Statement was executed and the position of the corresponding cursor is behind the last tuple. An Insert-, Update- or Delete-Statement executed without any tuple inserted, updated or deleted, resp. A Select-Into-Statement executed and the result set was empty.

3.1 The SQLCA Structure

SQLCA (SQL Communication Area) is a data region into which the return information for an executable ESQL statement is transferred. It is automatically included

into an ESQL program. It consists of a structure with name `sqlca` (lower case letters) of type `Sqlca` which has the following components:

```
typedef struct
{
  Int4  sqlcode;
  Int4  sqlerrd[6];      /* only indices 0 through 3 used */
  char  sqlwarn0;
  char  sqlwarn1;
  char  sqlwarn2;      /* not used */
  char  sqlwarn3;
  char  sqlwarn4;      /* not used */
  char  sqlwarn5;      /* not used */
  char  sqlwarn6;      /* not used */
  char  sqlwarn7;      /* not used */
  Int4  errline;        /* error program line */
  char  *errmodule;     /* name of current module */
  char  *tb_errtxt;     /* points to error message */
  Id    taid;           /* id of current TA */
  State tastate;      /* state of current TA */
  Id    dbid;           /* id of current database */
  State dbstate;       /* state of current database */
} Sqlca;
```

All values can be interpreted by an ESQL program. However, it is strongly discouraged to change any of the components of `sqlca` by explicit assignment operations.

3.1.1 SQLCODE

The component `sqlcode` is set as follows:

sqlcode	meaning
0	The statement executed successfully (SQLSUCCESS) or with warning
< 0	The statement failed (SQLError).
100	The event NOT FOUND occurred.

3.1.2 SQLERRD

The array `sqlerrd` is only used on entries 0 through 3. The other components are supported for compatibility to the XOPEN standard and are always 0.

`sqlerrd[2]` and `sqlerrd[3]` are set after a (successfully) executed Insert-, Delete-, Update-, Spool-Table- or Spool-File-Statement.

The component `sqlerrd[2]` contains the number of tuples which were inserted, deleted, updated or spooled, resp. The component `sqlerrd[3]` is equal to `sqlerrd[2]` in case of a Delete-, Update- and Spool-File-Statement. In case of an Insert- and Spool-Table-Statement, `sqlerrd[3]` is the number of tuples which were presented to the insertion process. Thus the (non-negative) difference between `sqlerrd[3]` and `sqlerrd[2]` is the number of duplicates which were simply ignored by the insertion process. Note that in a Transbase table duplicate tuples never appear. A tuple is considered a duplicate if exactly the same tuple (with exactly the same attribute values) is already contained in the particular table.

The entries `sqlerrd[0]` and `sqlerrd[1]` are described in the chapter "Dynamic ESQL Statements".

3.1.3 SQLWARN flags

The components `sqlwarn0`, `sqlwarn1` and `sqlwarn3` are used as warning flags, i.e. to indicate the `SQLWARNING` event (success with warning). If a statement returns with an event different from `SQLWARNING` all warning flags are set to the space character (' '). To test if all warning flags are blank it is sufficient to test if `sqlwarn0` is blank. If an `SQLWARNING` event occurs then `sqlwarn0` and one of `sqlwarn1` or `sqlwarn3` is set to the character 'W'.

<code>sqlwarn1 == 'W'</code>	at least one character string truncation in the assignment of field values to host variables occurred.
<code>sqlwarn3 == 'W'</code>	in a Fetch-Statement or a Select-Into-Statement the number of fields of the result tuple(s) is not equal to the number of host variables.

3.1.4 taid and taste

The component `taid` contains the transaction identifier of the current or last current transaction. Its type is predefined as `Id` and its initial value is -1. Its value is set as a side effect of ESQL statements as follows:

- The Begin-Work-Statement sets the value of `taid` to a transaction identifier delivered by the ESQL runtime system.
- The Use-Transaction-Statement sets the value of `taid` to the one specified in the Use-Transaction-Statement.
- For an open cursor, the Fetch-, Update-Positioned-, Delete-Positioned- and Close-Statement sets the value of `taid` to the identifier of the transaction for which the corresponding Open-Cursor-Statement has been performed.

Note that an explicit end (COMMIT/ROLLBACK) or an implicit end (ROLLBACK due to a severe error) does not change the `taid`.

The component `tastate` contains the status of the transaction identified by `taid`. Its type is predefined as `State`. The states are described in chapter "States of a Transaction".

Note that `tastate` is automatically adapted with respect to changes of the transaction's state. This also holds, whenever the `taid` is changed (see above). This means that both components are always consistent. Thus the program can run multiple transactions by remembering the transactions identifiers only; their states can be tested via the Use-Transaction-Statement.

3.1.5 dbid and dbstate

The component `dbid` contains the database identifier of the current or last current database. Its type is predefined as `Id`. Its initial value is -1. Its value is set as a side effect of ESQL statements as follows.

- The Connect-Statement sets the value of `dbid` to a database identifier delivered by the ESQL runtime system.
- The Use-Database-Statement sets the value of `dbid` to the one specified in the Use-Database-Statement.
- For an open cursor, the Fetch-, Update-Positioned-, Delete-Positioned- and Close-Statement sets the value of `dbid` to the identifier of the database against which the corresponding Open-Cursor-Statement has been performed.

Note that an explicit end of the connection (Disconnect-Statement) or an implicit end (broken connection due to a severe error) does not change the `dbid`.

The component `dbstate` contains the status of the connection identified by `dbid`. Its type is predefined as `State`. The states are described in chapter "States of a Database Connection".

Note that `dbstate` is automatically adapted with respect to changes of the connection's state. This also holds, whenever the `dbid` is changed (see above). This means that both components are always consistent. Thus the program can run multiple connections by remembering the database identifiers only; their states can be tested via the Use- Database-Statement.

3.1.6 Error Treatment

Whenever an SQLERROR occurs then three actions are performed automatically with the `sqlca` structure:

- The component `sqlcode` is set to an error specific code;
- The component `tb_errtxt` points to a textual error message.
- The component `errline` is set to the line number of the ESQL source program in which the error occurred.
- The component `errmodule` is set to the name of the module where the error occurred.

3.1.7 Hard and Soft SQLERRORs

Each SQLERROR code supplied in `sqlca.sqlcode` 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*.

Examples for hard errors are:

- Lack of sufficient memory or disk resources to process a query
- Integrity violations (e.g. inserting a non-duplicate with an already existing primary key).

If an ESQL statement finishes with a soft error, that call simply has no effect, i.e. the program in principle can proceed as if the call had not been made. Especially, the current transaction will not be aborted automatically.

Examples for soft errors are:

- Errors in a Select-Statement (e.g. unknown table).
- Fetch-Statement on a closed cursor.
- Type exceptions (overflow).

In summary, a soft error makes the current ESQL statement undone, a hard error makes the current transaction undone.

After an SQLERROR has occurred, it can easily be tested by inspecting the `sqlca` component `tastate` whether the transaction has been aborted or not (see chapter "taid and tastate").

3.2 The WHENEVER Directive

The components `sqlcode` and `sqlwarn0` of the `sqlca` structure provide sufficient information to catch the result events of ESQL statements. A more comfortable way is offered to the ESQL programmer by the **Whenever-Statement**. This is an ESQL compiler directive which can be used to attach independently to each of the result events `SQLWARNING`, `SQLERROR` and `NOT FOUND` one of three possible actions as follows:

Ignore the event (**CONTINUE**):

This action is the default action for all events.

Jump to a programmer defined label (**GOTO**):

There must be a corresponding label in the same C function and in the same or a surrounding block.

Call a user defined function (**CALL**):

There must be a corresponding function callable from the ESQL program, i.e. in one of the constituting ESQL compilation units. No parameter can be transferred to the function and no return value of the function can be interpreted.

Note that the Whenever-Statement is not a dynamic (executable) statement but a static directive which influences code generation. This means that an action attached to an event by a directive `d1` is valid for all executable ESQL statements which appear textually behind `d1` and before another directive `d2` which redefines the action for the same event.

Example:

The following C functions show an incorrect use of the **WHENEVER** statement:

```
func1()
{
    EXEC SQL OPEN c1;
    EXEC SQL WHENEVER NOT FOUND GOTO fe1;
    while (1) {
        EXEC SQL FETCH ...;
```

```
        ...
    }
fe1:
    EXEC SQL CLOSE c1;
}

func2()
{
    EXEC SQL OPEN c2;
    while (1) {
        EXEC SQL FETCH ...;
        ...
    }
    EXEC SQL CLOSE c2;
}
```

When translated by the C-Compiler, an error message would appear that the label `fe1` is not defined in function `func2`. Since no `WHENEVER` statement appears in `func2`, the `ESQL` preprocessor used the last `WHENEVER` statement, namely `"goto fe1"` also in `func2` where no such label is defined.

Chapter 4

Syntax of ESQL Statements

This part describes the syntax of each ESQL statement. The same syntax conventions as in the TB/SQL Reference Manual apply:

- Brackets [] are delimiters for an optional part.
- The vertical line | separates alternatives.
- Braces { } group several items together, e.g. to form complex alternatives. They are functionally equivalent to the standard braces () as used in arithmetic expressions.
- An ellipsis . . . indicates that the preceding item may be repeated arbitrarily often.

To distinguish terminal from non-terminal symbols, all non-terminal symbols start with an uppercase letter followed by lowercase letters, all terminal symbols are represented by themselves.

All keywords are written in uppercase letters.

4.1 ESQL Statement, EXEC SQL, Comment

An ESQL statement begins with the prefix EXEC SQL and ends with a semicolon. The semicolon may but need not be preceded by white space (blanks, tabs, new-lines).

An ESQL statement may span many lines but must begin on a separate line. White space (blanks, tabs) are allowed on the beginning of the line. After the terminating semicolon of an ESQL statement, arbitrary C statements and/or C comments are allowed within the same line.

C comments are also allowed inside an ESQL statement, of course they are not recognized inside SQL string literals.

SQL comments (introduced by `-`) are allowed inside an ESQL statement, not behind the terminating semicolon. They are implicitly terminated by end of line.

Keywords are case insensitive.

Whenever a blank or a newline is written in the syntax description, arbitrary white spaces are permitted.

Example:

```
EXEC SQL
  DECLARE C1 CURSOR          -- Cursor declaration
  SELECT tname, colno       -- Select clause
  FROM   systable           -- ...
  WHERE  colno > 7;        /* end of statement */
```

4.2 Host-Var, Host-Var-Ind

Serves to denote a host variable reference in an ESQL statement

Syntax:

```
Host-Var      ::=      Host-Var-Ref
Host-Var-Ind  ::=      Host-Var-Ref [ Indicator-Ref ]
Host-Var-Ref  ::=      :Identifier
Indicator-Ref ::=      :Identifier
```

Explanation: Host-Var is a host variable reference without an indicator variable. Host-Var-Ind is a host variable reference with an optional indicator variable. The identifiers refer to the names of C variables which must be declared in a declare section. The declaration must textually precede any use of the variables. The names of the variables must be valid in the current block (according to the scope rules of C).

The type of an indicator variable must be signed two-byte (short).

4.3 Summary of ESQL-Statements

```
ESQL-Statement ::=
  Declarative-Statement
  | Executable-Statement
```

```

Declarative-Statement ::=
    Include-Sqlca-Statement
    | Whenever-Statement
    | Declare-Section-Statement
    | Declare-Cursor-Statement

```

```

Executable-Statement ::=
    Connect -Statement
    | Disconnect-Statement
    | Login-Statement
    | Begin-Work-Statement
    | Commit-Work-Statement
    | Rollback-Work-Statement
    | Use-Transaction-Statement
    | Use-Database-Statement
    | Set-Sortorder-Statement
    | Get-Sortorder-Statement
    | Set-Data-Dir-Statement
    | Set-Timeout-Statement
    | Set-Consistency-Statement
    | Send-Interrupt-Statement
    | Open-Cursor-Statement
    | Fetch-Statement
    | Delete-Positioned-Statement
    | Update-Positioned-Statement
    | Close-Cursor-Statement
    | Select-Into-Statement
    | Embedded-TBSQL-Statement
    | Dynamic-ESQL-Statement

```

Explanation: A Declarative-Statement may appear on all places where a C variable declaration is legal.

An Executable-Statement may appear on all places where a C statement is legal.

All Statements are defined within the following paragraphs. For the definition of Dynamic ESQL Statements see the corresponding chapter "Dynamic ESQL Statements".

4.4 Embedded-TBSQL-Statement

Embedded-TBSQL-Statements are derived from TB/SQL Statements as defined in the TB/SQL Reference Manual. For the complete syntax and semantics see TB/SQL Reference Manual.

Syntax:

```
Embedded-TBSQL-Statement ::=
    EXEC SQL [ Store-Clause]

Storable-Statement ::=
    EXEC SQL Not-Storable-Statement

Store-Clause ::=
    STORED
    | NOT STORED

Not-Storable-Statement ::=
    Create-Table-Statement
    | Drop-Table-Statement
    | Create-View-Statement
    | Drop-View-Statement
    | Create-Index-Statement
    | Drop-Index-Statement
    | Create-Domain-Statement
    | Alter-Domain-Statement
    | Drop-Domain-Statement
    | Alter-Table-Statement
    | Grant-Userclass-Statement
    | Revoke-Userclass-Statement
    | Grant-Privilege-Statement
    | Revoke-Privilege-Statement
    | Alter-Password-Statement
    | Spool-Table-Statement
    | Spool-File-Statement
    | Tbmode-Statement
    | Load-Statement
    | Unload-Statement
    | Lock-Statement
    | Unlock-Statement
    ;

Storable-Statement ::=
    Insert-Statement
    | Update-Statement
    | Delete-Statement
    ;
```

Explanation: With the exception of the Select-Statement each TB/SQL statement can be written as ESQL Statement by prefixing it with EXEC SQL and terminating it with a semicolon.

All listed statements are described in the TB/SQL Reference Manual. The default rules for the Store-Clause are explained in the chapter "Stored Queries".

Note: If no current transaction exists and an Embedded-TBSQL-Statement is executed, a new transaction is started automatically.

Example:

```
EXEC SQL DROP TABLE suppliers ;

EXEC SQL CREATE INDEX quot_price
      ON quotations (price) ;

EXEC SQL INSERT INTO quotations
      VALUES (55,220,13.5,10,0) ;

EXEC SQL UPDATE quotations
      SET price = price * 1.1
      WHERE suppno = 54 ;

EXEC SQL DELETE FROM quotations
      WHERE suppno = 54 ;
```

4.5 Include-SQLCA-Statement

Supported for compatibility.

Syntax:

```
Include-SQLCA-Statement ::=
      EXEC SQL INCLUDE SQLCA ;
```

Example: This statement can appear at all places and has no effect.

Note: Note that the SQLCA Communication Area is included automatically into each ESQL program.

The include file <stdio.h> is included by default, too.

Example:

```
EXEC SQL INCLUDE SQLCA ;
```

4.6 Declare-Section-Statement

Declares C variables that can be used as host variables.

Syntax:

```
Declare-Section-Statement ::=  
    EXEC SQL BEGIN DECLARE SECTION;  
    Host-Variable-Declarations;  
    EXEC SQL END DECLARE SECTION;
```

Syntax:

```
Host-Variable-Declarations ::=  
    <arbitrarily many lines of C variable declarations>
```

Explanation: Arbitrarily many declare sections may appear in an ESQL program. No include statements, comments or static initializations are allowed in a declare section.

A declare section is allowed on all places where the declaration of C variables or parameters is allowed.

Allowed types and storage classes are described in chapter "SQL Types and Host Variable Types" (see 2.6).

Every host and indicator variable used in an ESQL statement must be declared in a declare section. The declaration must textually precede its use. The names of all host and indicator variables of all declare sections must be disjoint inside a module (regardless of the corresponding C scope).

Note: Although the Declare-Section-Statement consists of two EXEC ESQL constructs and additional C constructs, it is conceptually treated and named as one ESQL statement within this manual.

Example:

```

int i,j;
EXEC SQL BEGIN DECLARE SECTION;
Integer suppno,partno;
Numeric price;
short inds;
static Uint4 l1;
char c, *pc, arr[100];
extern char extarr[];
EXEC SQL END DECLARE SECTION;
main()
{ ...
}

```

Example:

```

int func(i,j)
int i;
EXEC SQL BEGIN DECLARE SECTION;
Integer j;
EXEC SQL END DECLARE SECTION;
{ ...
}

```

4.7 Declare-Cursor-Statement

Serves to declare a cursor.

Syntax:

```

Declare-Cursor-Statement ::=
    EXEC SQL [ Store-Clause ]
    DECLARE Cursor-Name [ SCROLL] CURSOR FOR
        { Select-Statement | Statement-Var } ;

```

```

Store-Clause ::=
    STORED
    | NOT STORED

```

```

Cursor-Name ::=
    Identifier

```

```
Select-Statement ::=
    < see TB/SQL Reference Manual >
```

```
Statement-Var ::=
    #Identifier
```

Explanation: A Declare-Cursor-Statement defines a cursor name and attaches to it a result table specified by a Select-Statement. The Select-Statement is either explicitly specified (static cursor) or is specified via a statement host variable (dynamic cursor, see chapter "Dynamic SQL Statements").

A Declare-Cursor-Statement is valid on all places where a C variable declaration is valid. The declaration of a cursor must textually precede its use. All cursor names within an SQL program module must be disjoint. Furthermore, all cursor names must be different from all Host-Var names declared within the same module.

A cursor is updatable if the corresponding Select-Statement is updatable and specified with the FOR UPDATE clause (see TB/SQL Reference Manual) The default rules for the Store-Clause are explained in chapter "Stored Queries".

The semantics of cursors are described in full detail in chapter "Cursor".

Example:

```
EXEC SQL DECLARE c1 CURSOR FOR
    SELECT suppno, count(*)
    FROM quotations
    WHERE price > 30.5
    GROUP BY suppno;
```

```
EXEC SQL DECLARE c2 CURSOR FOR
    SELECT suppno, partno, price
    FROM quotations
    WHERE suppno = 54
    FOR UPDATE;
```

```
EXEC SQL BEGIN DECLARE SECTION;
Integer sup, par;
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL NOT STORED DECLARE c3 CURSOR FOR
    SELECT suppno, partno, price
    FROM quotations
    WHERE suppno = :sup and partno = :par
    FOR UPDATE;
```

4.8 Whenever-Statement

Compiler directive for runtime event checking.

Syntax:

```
Whenever-Statement ::=
    EXEC SQL WHENEVER Event Action ;
Event ::=
    SQLERROR | SQLWARNING | NOT FOUND
Action ::=
    CONTINUE | GOTO Label | CALL Function
Label ::=
    < C program label identifier >
Function ::=
    < C program function identitfier >
```

Explanation: If GOTO Label is specified a corresponding label must be defined in the same ESQL module.

If CALL Function is specified a corresponding C function must be callable from the ESQL module.

For all executable ESQL statements which textually appear behind the Whenever-Statement and prior to another Whenever-Statement which specifies the same event, the specified action is performed at runtime whenever the event occurs.

The default action for all events is CONTINUE.

Example:

```
main()
{
    ...
    EXEC SQL WHENEVER SQLERROR CALL error ;

    EXEC SQL OPEN c1 ;
    EXEC SQL WHENEVER NOT FOUND GOTO endc1 ;

    while(1)
        EXEC SQL FETCH c1 INTO :h1, :h2 ;

    endc1:
    EXEC SQL CLOSE c1 ;
    ...
}
```

```

}
void error()
{   printf('error %ld in line %d: %s\n',
        sqlca.sqlcode, sqlca.errline, sqlca.tb_errtxt);
    exit(1);
}

```

Example:

```

/* !the following error handling does not work! */

main()
{   ...
    set_sqlerror();

    EXEC SQL OPEN c1 ;
    EXEC SQL WHENEVER NOT FOUND GOTO endc1 ;

    while(1)
        EXEC SQL FETCH c1 INTO :h1, :h2 ;

    endc1:
    EXEC SQL CLOSE c1 ;
    ...
}

set_sqlerror()
{   EXEC SQL WHENEVER SQLERROR CALL error ;
    /* no effect on the above ESQL statements !! */
}

void error()
{   printf('error %ld in line %d: %s\n',
        sqlca.sqlcode, sqlca.errline,
sqlca.tb_errtxt);
    exit(1);
}

```

4.9 Connect-Statement

Serves to connect to a database. An asynchronous 2-phase version also exists.

Syntax:

```

Connect-Statement ::=
    EXEC SQL CONNECT { Database | Host-Var };
Contactk -Statement ::=
    EXEC SQL CONTACTK{ Database | Host-Var };
Acceptk -Statement ::=
    EXEC SQL ACCEPTK{ Database | Host-Var };
Database ::=
    'Database-Name[@Host-Name]'
Database-Name ::=
    Identifier
Host-Name ::=
    Identifier

```

Effect: A Connect-Statement or a sequence of Contactk-Statement and Acceptk-Statement establishes a connection to the database specified by Database.

If successful, the database becomes the current database and a valid database identifier in the range [0 .. MAX_DB-1] is written into sqlca.dbid.

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 Connect-Statement returns the same dbid that has been returned by the first Connect-Statement for the same database (assuming that no Disconnect-Statement interfered).

A Database is identified by a character string consisting of a local database name and optionally a host name, separated by the character '@'. If no host name is given, the database is assumed to reside on the local host. If a remote host is given, it is tried to address the host in order to establish a remote connection to the required database.

If a host variable is specified its type must be appropriate to supply a character string.

Instead of issuing a CONNECT call, the application may parallelize this (sometimes time-consuming) operation into two actions CONTACTK and ACCEPTK. CONTACTK only initiates the CONNECT request but does not wait until the CONNECT has succeeded. A later ACCEPTK operation waits for the CONNECT operation to be completed. By this asynchronous CONNECT 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.

Example:

```
EXEC SQL CONNECT 'sample' ;
```

Example:

```
EXEC SQL BEGIN DECLARE SECTION;
char db[100];
EXEC SQL END DECLARE SECTION;

main()
{
    ...
    printf('connect to ');
    gets(db);
    EXEC SQL CONNECT :db ;
    ...
}
```

Example:

```
/* Connecting to more than 1 database */

EXEC SQL CONNECT 'sample@sun' ;
db1 = sqlca.dbid;

EXEC SQL CONNECT 'sample@vax' ;
db2 = sqlca.dbid;
```

Example:

```
/* Asynchronous Connect */

EXEC SQL CONTACTK 'sample@sun' ;
db = sqlca.dbid;
/* do something useful, e.g. get userid and password .. */
EXEC SQL ACCEPTK 'sample@vax' ;
/* connection ready */
```

4.10 Disconnect-Statement

Serves to disconnect from the current database.

Syntax:

```
Disconnect-Statement ::=
    EXEC SQL DISCONNECT ;
```

Explanation: Disconnects from the current database.

Error occurs if a transaction is active on the database.

Example:

```
EXEC SQL DISCONNECT ;
```

Example:

```
EXEC SQL USE DATABASE :db1 ;
EXEC SQL DISCONNECT ;
EXEC SQL USE DATABASE :db2 ;
EXEC SQL DISCONNECT ;
```

4.11 Login-Statement

Serves to login into the current database.

Syntax:

```
Login-Statement ::=
    EXEC SQL LOGIN User-Name
        [ PASSWORD Password ] ;
User-Name ::=
    'Identifier' | Host-Var
Password ::=
    'Identifier' | Host-Var
```

Explanation: If a Host-Var is specified for User-Name or Password its type must be appropriate to supply a character string.

The statement performs a login for the specified user on the current database. If the user has changed his default password (empty string) to a private password, then the password must be specified.

Example:

```
EXEC SQL BEGIN DECLARE SECTION;
char pwd[50];
EXEC SQL END DECLARE SECTION;

strcpy(pwd, getpass('Password: '));
EXEC SQL LOGIN 'smith' PASSWORD :pwd ;
```

4.12 Begin-Work-Statement

Explicitly starts a new transaction.

Syntax:

```
Begin-Work-Statement ::=  
    EXEC SQL BEGIN WORK ;
```

Explanation: A new transaction is started. The transaction becomes the current transaction and a valid transaction identifier in the range [0 .. MAX_TA-1] is written into sqlca.taid.

The maximum number of transactions an application can have active at a time, is predefined as the compile time constant MAX_TA.

Note: A transaction is started automatically if no current transaction exists and an Embedded-TBSQL-Statement or an Open-Cursor-Statement is executed.

Example:

```
EXEC SQL BEGIN DECLARE SECTION;  
Id ta1;  
EXEC SQL END DECLARE SECTION;  
  
EXEC SQL BEGIN WORK ;  
ta1 = sqlca.taid;
```

4.13 Commit-Work-Statement

Commits the current transaction.

Syntax:

```
Commit-Work-Statement ::=  
    EXEC SQL COMMIT WORK ;
```

Explanation: The current transaction is committed, i.e. all its effects are saved on disk. No current transaction exists after this statement.

All open cursors of the current transaction are automatically closed.

Example:

```
EXEC SQL COMMIT WORK ;
```

4.14 Rollback-Statement

Aborts the current transaction.

```
Rollback-Statement ::=  
    EXEC SQL ROLLBACK WORK ;
```

Explanation: The current transaction is aborted, i.e. all its effects are made undone. No current transaction exists after this statement.

All open cursors of this transaction are automatically closed.

Example:

```
EXEC SQL ROLLBACK WORK ;
```

4.15 Open-Statement

Serves to open a cursor.

Syntax:

```
Open-Statement ::=  
    EXEC SQL OPEN Cursor-Name;
```

Explanation: The input host variables in the Select-Statement of the specified cursor's Declare-Cursor-Statement 4.7 are evaluated and transferred as actual parameters into the Select-Statement of the specified cursor. Locks are requested and the cursor is opened on the query. The position of the cursor is set before the first tuple of the result set.

The specified cursor must be declared in the same ESQL module and must be in the closed state.

Example:

```
EXEC SQL OPEN c2 ;
```

4.16 Fetch-Statement

Serves to fetch the next tuple of a cursor and to assign its field values to output host variables.

Syntax:

```
Fetch-Statement ::=
    EXEC SQL FETCH [ [ row-selector ] FROM ] Cursor-Name
    [ INTO Host-Var-Ind [, Host-Var-Ind ] ... ] ;
row-selector ::=
    NEXT | PRIOR | FIRST | LAST | ABSOLUTE n | RELATIVE n
```

Explanation: The specified cursor must be in the open state.

If a row-selector is specified then the specified cursor must be a `SCROLL` cursor (see 4.7).

Assume that "card" is the cardinality of the result set (`card > 0`).

If no row-selector or `NEXT` is specified then the cursor is set to the next result tuple unless it was already on or behind the last tuple (in this case it is set behind the last result tuple and `SQLCODE` is set to 100 and the host variables if any remain unchanged).

If `PRIOR` or `LAST` or `FIRST` is specified then the cursor is set to the preceding or last or first tuple, resp.

If `ABSOLUTE` is specified with a positive number `n` then the cursor is set to the `n`-th result tuple if `n <= card` otherwise behind the last tuple.

If `ABSOLUTE` is specified with a negative number `n` then the cursor is set to the `|n|`-th result tuple counted from backward if `|n| <= card` otherwise before the first tuple.

If `RELATIVE` is specified with a number `n` then the cursor is advanced by the number of tuples specified by `n` (for negative `n` the cursor is set backwards by `|n|` tuples. The cursor might be behind the last tuple or before the first tuple after such an operation (denoted in `SQLCODE`).

If host variables are specified and the cursor is `ON` a row after the operation, then the field values of the tuple are assigned to the host variables.

If host variables are specified and the number h of host variables agrees with the number f of result tuple fields then for all i , $1 \leq i \leq h$, the value of the i -th field is assigned to the i -th host variable.

If the number h of specified host variables does not agree with the number f of result tuple fields, then `SQLWARN0` and `SQLWARN3` are set to 'W' and min field values are assigned where min is the minimum of h and f .

Rules for assignment, type conversions and null value indicators are described in chapter "SQL Types and Host Variable Types" (see 2.6).

If the cursor is dynamic then no host variables must be specified (see chapter "Dynamic ESQL Statements").

If the specified cursor refers to a database which is not the current database, an implicit switch to the database occurs where the cursor refers to.

Example:

```
EXEC SQL BEGIN DECLARE SECTION;
Integer suppno, partno;
short inds;
Real price;
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH c2 INTO :suppno:inds, :partno, :price;
```

4.17 Delete-Positioned-Statement

The Delete-Positioned-Statement deletes from a table the current tuple of a cursor.

Syntax:

```
Delete-Positioned-Statement ::=
    EXEC SQL [ Store-Clause ]
    DELETE FROM Table-Name
    WHERE CURRENT OF Cursor-Name;
```

```
Store-Clause ::=
    STORED
    | NOT STORED
```

```
Table-Name ::=
    Identifier
```

Explanation: The specified cursor must be updatable, i.e. the Select-Statement of the corresponding Declare-Cursor-Statement must include the `FOR UPDATE` clause. The specified table must be the same as the (one and only) table specified in the `FROM` clause of the Select-Statement. The cursor must be on a tuple of its result set, i.e. it must not be positioned before the first, behind the last or between two tuples.

The tuple in the specified table from which the current tuple is derived is deleted. No current tuple then is defined for the cursor.

For the delete semantics and necessary privileges see also the Delete- Statement in TB/SQL Reference Manual: all explanations there also apply to the Delete-Positioned-Statement with the exception that not a tuple set specified by a Search-Condition is deleted but only one tuple specified by the current tuple of the cursor.

If the specified cursor refers to a database which is not the current database, an implicit switch to the database occurs where the cursor refers to.

Example:

```
EXEC SQL STORED
DELETE FROM quotations
WHERE CURRENT OF c2 ;
```

4.18 Update-Positioned-Statement

The Update-Positioned-Statement updates in a table the current tuple of a cursor.

Syntax:

```
Update-Positioned-Statement ::=
    EXEC SQL [ Store-Clause ]
    UPDATE Table-Name [Correlation-Name ]
    SET Assign-List
    WHERE CURRENT OF Cursor-Name;

Store-Clause ::=
    STORED
    | NOT STORED

Table-Name ::=
    < see Update-Statement in TB/SQL Reference-Manual >
```

Correlation-Name ::=
 < see Update-Statement in TB/SQL Reference-Manual >

Assign-List ::=
 < see Update-Statement in TB/SQL Reference-Manual >

Explanation: The specified cursor must be updatable, i.e. the Select- Statement of the corresponding Declare-Cursor-Statement must include the FOR UPDATE clause. The specified table must be the same as the (one and only) table specified in the FROM clause of the Select-Statement. The cursor must be on a tuple of its result set, i.e. it must not be positioned before the first, behind the last or between two tuples.

The tuple in the specified table from which the current tuple is derived is updated according to the values of the Assign-List. The position of the cursor remains on the current tuple. Note that fields specified in the Assign- List need not necessarily be members of the result set of the cursor as shown in the example below with cursor c2.

For the update semantics and necessary privileges see also the Update- Statement in TB/SQL Reference Manual: all explanations there also apply to the Update-Positioned-Statement with the exception that not a tuple set specified by a Search-Condition is updated but only one tuple specified by the current tuple of the cursor.

If the specified cursor refers to a database which is not the current database, an implicit switch to the database occurs where the cursor refers to.

Example:

```
EXEC SQL
UPDATE quotations
SET    delivery_time = 23, qonorder = 50
WHERE CURRENT OF c2 ;
```

4.19 Close-Statement

Serves to close a cursor.

Syntax:

Close-Statement ::=
 EXEC SQL CLOSE Cursor-Name ;

Explanation: The specified cursor is closed.

The specified cursor must be in the open state.

If the specified cursor refers to a database which is not the current database, an implicit switch to the database occurs where the cursor refers to.

Example:

```
EXEC SQL CLOSE c2 ;
```

4.20 Select-Into-Statement

Executes a Select-Statement which delivers a single tuple and assigns its field values to host variables.

Syntax:

```
Select-Into-Statement ::=
    EXEC SQL [ Store-Clause ]
    SELECT [ALL | DISTINCT] { * | Expr-List }
    INTO Host-Var-Ind [, Host-Var-Ind ] ..
    FROM Table_Reference [, Table_Reference ] ...
    [ WHERE Search-Condition ]
    [ GROUP BY Field [, Field ] ...
    [ HAVING Search-Condition ] ;
```

```
Store-Clause ::=
    STORED
    | NOT STORED
```

Explanation: The syntax of a Select-Into-Statement is that of a `Select_Expression` (also called `Query_Block`, see TB/SQL Reference Manual) with an additional `INTO` clause between the `SELECT` clause and the `FROM` clause.

The Select-Into-Statement is a shortcut notation for `SELECT` queries which are known or expected to deliver at most one result tuple.

Semantically, the Select-Into-Statement is equivalent to the following sequence: A `Declare-Cursor-Statement` with a `Query-Block` where the `INTO` clause is omitted; an `Open-Cursor-Statement` with that cursor; one `Fetch-Statement` on that cursor with the same `INTO` clause as the one in the `Select-Into-Statement`. A `Close-Cursor-Statement` on that cursor.

If the Select-Statement produces no result tuple, the event NOT FOUND is triggered. If the Select-Statement would produce more than one result tuple, the event SQLERROR is triggered (see chapter "The SQL Communication Area and Error Treatment").

The default rules for the Store-Clause are explained in chapter "Stored Queries".

Note: A Select-Into-Statement runs faster than an equivalent cursor procedure. It is offered as a semantic optimization tool to the application programmer.

Example:

```
EXEC SQL
SELECT suppno, name, address
INTO :h1, :h2, :h3
FROM suppliers
WHERE suppno = 54 ; /* suppno is primary key */
```

4.21 Use-Transaction-Statement

Switches to another transaction of the connection.

Syntax:

```
Use-Transaction-Statement ::=
    EXEC SQL
    USE TRANSACTION { Host-Var | Int-Constant } ;
Int-Constant ::=
    < C integer constant >
```

Explanation: The transaction with the transaction identifier supplied by the constant or host variable becomes the current transaction.

sqlca.taid and sqlca.tastate are updated according to the supplied transaction identifier.

The type of the host variable must correspond to the type Id.

Note: This statement is only useful if multiple transactions are run in an interleaved fashion from one and the same application.

Example:

```

EXEC SQL BEGIN DECLARE SECTION;
Id ta1,ta2;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL      BEGIN WORK;
ta1 = sqlca.taid;
...
EXEC SQL      BEGIN WORK;
ta2 = sqlca.taid;
...
EXEC SQL USE TRANSACTION :ta1 ;
...
EXEC SQL USE TRANSACTION :ta2 ;
...

```

4.22 Use-Database-Statement

Switches to another database.

Syntax:

```

Use-Database-Statement ::=
    EXEC SQL
    USE DATABASE { Host-Var | Int-Constant } ;
Int-Constant ::=
    < C integer constant >

```

Explanation: The database with the database identifier supplied by the constant or host variable becomes the current database.

sqlca.dbid and sqlca.dbstate are updated according to the supplied database identifier.

The type of the host variable must correspond to the type Id.

Note: This statement is only useful if more than one database are worked upon in an interleaved fashion.

Example:

```

EXEC SQL BEGIN DECLARE SECTION;
Id db1,db2;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL CONNECT 'sample@host1' ;
db1 = sqlca.dbid;
...
EXEC SQL CONNECT 'sample@host2' ;
db2 = sqlca.dbid;
...
EXEC SQL USE DATABASE :db1 ;
...
EXEC SQL USE DATABASE :db2 ;
...

```

4.23 Set-Sortorder-Statement

Sets a User Sortorder.

Syntax:

```

Set-Sortorder-Statement ::=
    EXEC SQL SET SORTORDER :Host-Var;

```

Explanation: The call defines a character sortorder for result tuples of queries. The sortorder is defined via the input host variable which must be an array SA of type unsigned char of length 256. If SA['A'] is less than SA['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 SET_SORTORDER call.

Note: All values of SA[i] must be different from 0.

Example:

```
EXEC SQL BEGIN DECLARE SECTION
unsigned char sort_array[256];
EXEC SQL END DECLARE SECTION
... /* Setting a sortorder */
EXEC SQL SET SORTORDER :sort_array;
```

4.24 Get-Sortorder-Statement

Retrieves the actually valid Sortorder.

Syntax:

```
Get-Sortorder-Statement ::=
    EXEC SQL GET SORTORDER :Host-Var;
```

Explanation: The call retrieves the actually valid sortorder into the output host variable which must be an array SA of type unsigned char of at least length 256.

Example:

```
EXEC SQL BEGIN DECLARE SECTION
unsigned char sort_array[256];
EXEC SQL END DECLARE SECTION
... /* Retrieving a sortorder */
EXEC SQL GET SORTORDER :sort_array;
```

4.25 Set-Data-Dir-Statement

Sets the data directory for spooling external data.

Syntax:

```
Set-Data-Dir-Statement ::=
    EXEC SQL
    SET DATA DIRECTORY { Path-Name | Host-Var } ;
```

```
Path-Name ::=
    'Identifier' ;
```

Explanation: The directory specified by Path-Name or referenced by Host-Var is used to locate external files for subsequent Spool-Table-Statements and Spool-File- Statements against the current database.

It is possible to have different directories for each database connected to. The Set-Data-Dir-Statement refers to the current database.

If no Set-Data-Dir-Statement has been issued, the current directory; of the application program is used as default (for all databases connected to).

No error will be returned if pathname is an invalid directory name. Instead, this error will occur when running a Spool-Table-Statement or Spool-File- Statement against the particular database.

If a host variable is specified it must be of a type appropriate to supply a character string.

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 SET_DAT_DIR 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.

Example:

```
EXEC SQL SET DATA DIRECTORY 'usr/smith/data' ;
EXEC SQL SET DATA DIRECTORY :ddir ;
```

4.26 Set-Timeout-Statement

Sets the timeout for explicit and implicit lock requests

Syntax:

```
Set-Timeout-Statement ::=
    EXEC SQL
        SET TIMEOUT { Host-Var | Int-Constant } ;
Int-Constant ::=
    < C integer constant >
```

Explanation: A new default value for timeouts is set. The value specifies the timeout in seconds.

The value 0 means that no timeout will occur. Note that this may result in indefinite delay.

If no Set-Timeout-Statement is given a default value of 60 seconds applies. The timeout setting is valid for all databases whether they have been connected already or not. A new timeout setting overrides the previous setting.

If a host variable is specified, its type must be one of the C arithmetic types such as int or Int4.

Example:

```
EXEC SQL SET TIMEOUT 20 ;
```

4.27 Set-Consistency-Statement

Sets the level of read consistency low or high.

Syntax:

```
Set-Consistency-Statement ::=  
    EXEC SQL  
    SET CONSISTENCY { CONS_1 | CONS_2 | CONS_3 } ;
```

Explanation: The consistency-level for the application program is set to the specified level.

CONS_3 holds read locks until the end of a transaction. At level CONS_2 read locks are released automatically at the end of the statement for which they were acquired. At level CONS_1 read queries run without any locks.

Note: This statement is illegal when the application has a transaction active.

Example:

```
EXEC SQL SET CONSISTENCY CONS_2;
```

4.28 Send-Interrupt-Statement

Aborts all active transactions asynchronously

Syntax:

```
Send-Interrupt-Statement ::=  
    EXEC SQL SEND INTERRUPT ;
```

Explanation: Can be used directly in the body of a user defined interrupt function. Aborts all active transaction asynchronously. All open cursors if any are closed.

Note: In order to work correctly, the Transbase process "tserver" must be active.

Note: It is illegal to place the Send-Interrupt-Statement in the body of a function which is called by the user defined interrupt function. Only place it directly inside the user defined interrupt function.

Example: See chapter "Signal Handling".

Chapter 5

Retrieving and Inserting BLOB Objects

The TB/ESQL interface uses a data structure called Blobdesc to retrieve, insert and update 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 or UPDATE 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 application */
            unsigned mallocsize; /* for system */
            char *mmadr;      /* main mem.address */
        } mmem;
    } loc;
} Blobdesc;
```

5.1 Using Blobdesc as Host Variable

In TB/ESQL, BLOB objects are fetched using Blobdesc's as target host variables, and they are inserted using Blobdesc's as source variables.

Each Blobdesc must be declared in the DECLARE SECTION:

Example:

```
EXEC SQL BEGIN DECLARE SECTION;
Blobdesc blobdesc;
EXEC SQL END DECLARE SECTION;
```

5.2 Fetching BLOB Objects in TB/ESQL

A field of type BLOB can be fetched using a host variable of type Blobdesc. Assume "blobdesc" is a variable of structure Blobdesc. The contents of blobdesc control the storage of the BLOB, i.e. the user program must set the Blobdesc variable appropriately.

If blobdesc.mode is 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 is MMADR then the BLOB is stored in main memory. For this purpose TB/ESQL inspects the component blobdesc.loc.mmем.usrmalloc as described below.

A value of 1 in blobdesc.loc.mmем.usrmalloc indicates that the user has reserved enough space to store the BLOB and TB/ESQL takes the address in blobdesc.loc.mmем.mmadr as the target address to store the BLOB.

A value of 0 in blobdesc.loc.mmем.usrmalloc indicates that the user leaves it up to TB/ESQL to reserve storage. **In this case, the component blobdesc.loc.mmем.mmadr must be set to NULL by the user program before the variable blobdesc is used for the first time in an TB/ESQL statement.** TB/ESQL then allocates appropriate storage for the BLOBs and remembers in blobdesc.loc.mmем.mallocsize the actually allocated storage size and in blobdesc.loc.mmем.mmadr the actually allocated storage. 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 TB/ESQL sets the component blobdesc.size to the size of the retrieved and stored BLOB object.

Example:

```
/* a 1-tuple SELECT INTO .. query;
the BLOB will be stored in file BLOB001 */
```

```

EXEC SQL BEGIN DECLARE SECTION;
Blobdesc blobdesc;
EXEC SQL END DECLARE SECTION;

blobdesc.mode = FILENAME;
blobdesc.loc.filename = 'BLOB001';
EXEC SQL
    SELECT image
    INTO :blobdesc
    FROM graphik
    WHERE graphik_name = 'g002';
/* now the BLOB is stored in file BLOB001 */

```

Example:

```

/* a 1-tuple SELECT INTO .. query;
only a subrange is fetched;
BLOB is fetched into main memory;
user program allocates memory for the BLOB
*/
EXEC SQL BEGIN DECLARE SECTION;
Blobdesc blobdesc;
EXEC SQL END DECLARE SECTION;

blobdesc.mode = MMADR;
blobdesc.loc.mmem.usrmalloc = 1;
blobdesc.loc.mmem.mmadr = (char*)malloc(100);
EXEC SQL
    SELECT image(1,100)
    INTO :blobdesc
    FROM graphik
    WHERE graphik_name = 'g002';
/* now the BLOB is stored in memory */

```

Example:

```

/* a SELECT query with a CURSOR and FETCH calls;
BLOB is fetched into main memory;
TB/ESQL' allocates memory for the BLOB
*/
EXEC SQL BEGIN DECLARE SECTION;
Blobdesc blobdesc;
EXEC SQL END DECLARE SECTION;

```

```

EXEC SQL DECLARE C1 CURSOR FOR
    SELECT image
    FROM graphik
    WHERE graphik_name = 'g002';

EXEC SQL OPEN C1;
    blobdesc.mode = MMADR;
    blobdesc.loc.mmem.usrmalloc = 0;
    blobdesc.loc.mmem.mmadr = NULL;
EXEC SQL FETCH C1 INTO :blobdesc;
...

```

5.3 Inserting and Updating BLOB Objects in TB/ESQL

Insertion and Update of BLOB objects is also done using a host variable of type Blobdesc. The semantics of blobdesc is analogous to that of BLOB retrieval: the user program indicates where the BLOB to be inserted 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 program.

Theoretically, the same Blobdesc host variable can be used on more than one field position in an INSERT 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.

Example:

```

/* an INSERT query where the BLOB is fetched from a file
''BLOB001''
*/
EXEC SQL BEGIN DECLARE SECTION;
Blobdesc blobdesc;
EXEC SQL END DECLARE SECTION;
blobdesc.mode = FILENAME;
blobdesc.loc.filename = ''BLOB001'';
EXEC SQL
    INSERT INTO graphik
        VALUES('imag001', 134, :blobdesc);
/* or for example an update */
EXEC SQL

```

```
UPDATE graphik SET image = :blobdesc
WHERE graphik_name = 'imag001'
```

5.4 BLOBs in Dynamic TB/ESQL

For dynamic TB/ESQL queries, the BLOB handling is restricted. Only the Spool Statements as described in chapter 5 have full functionality in dynamic TB/ESQL. As described in the chapter "Dynamic ESQL Statements", for **SELECT** statements, the fields of the result tuples are available as pointers via a special cursor structure. BLOB values, however, are delivered by their address only (a C-structure called Blob) and cannot be accessed directly but via special calls which are available at the lower level interface TBX (see TBX manual). It is possible to mix ESQL calls with TBX calls, thus the reader is referred to the TBX manual.

Chapter 6

Dynamic ESQL Statements

TB/ESQL not only provides for parameterized queries but also supports a mechanism to run statements and queries which are completely built up at runtime. By their nature, only executable ESQL statements can be run dynamically. To construct and run dynamic statements is very easy. However, the programmer has to distinguish between two classes of dynamic statements. One class consists of Select-Statements. This class must be run using the cursor technique as for static Select-Statements but with a special cursor declaration (dynamic cursor). Most of the remaining statements can be run by one EXEC SQL call as in the static case.

Dynamic queries cannot be stored (see chapter "Stored Queries") but are dynamically compiled each time they are executed.

BLOB handling of dynamic queries is rather limited (see chapter "BLOBs in Dynamic TB/ESQL").

6.1 Dynamic Cursors and Statement Variables

A dynamic cursor is declared by a Declare-Cursor-Statement where a statement variable is written in place of a textual Select-Statement. A statement variable serves to store text and thus must be of appropriate type, e.g. a character array or a character pointer. When such a cursor is opened, the variable may contain an arbitrary Select-Statement which is thus opened for execution. In contrast to a host variable, a statement variable must be referenced by prefixing it with a #.

The following program fragment shows the principles.

```
EXEC SQL BEGIN DECLARE SECTION;
char sqlquery[MAXQUERYSIZE];      /* predefined
constant */
```

```

EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE c1 CURSOR FOR #sqlquery;
        /* note the # */

main()
{
    int i;...
    < read query text of a SELECT query into sqlquery >
    ..
    EXEC SQL WHENEVER SQLERROR CALL error;
    EXEC SQL WHENEVER NOT FOUND GOTO end;
    EXEC SQL OPEN c1;
    while(1)
    {
        EXEC SQL FETCH c1;
        < access field values >          /* see
below */
    }
    end:
    ...
}

error()
{ ...
}

```

Note the following differences to a static cursor:

- The cursor declaration has a statement variable reference instead of the statement text.
- The FETCH statement must not have an INTO clause. Instead the field values must be read in a different way as described in the sequel.
- No host variable references are allowed in the Select Statement which is stored in the statement variable.

6.1.1 Runtime Structure of a Cursor

The following rules apply both to static and dynamic cursors (however, the programmer needs to know those rules only to process dynamic cursors). The ESQL precompiler associates to each cursor a variable with the same name as that of the cursor. The type of the variable is pointer to Cursor (Cursor*) where Cursor is defined as follows:

```

typedef struct
{
    char *crsname;    /* name of cursor */
    Id taid;         /* used by ESQL */
    Id dbid;         /* used by ESQL */
    Query_descr qudescr;
} Cursor;

```

Assume that a cursor with name `c1` is declared in the ESQL program. Then at any place in the scope of the cursor, a C variable named `c1` exists and can be processed by normal C statements. The variable `c1` exhibits the following properties:

- The variable `c1` has the value `NULL` if and only if the cursor is in the closed state.
- The variable `c1` points to a structure of type `Cursor` as long as the cursor `c1` is open. It is not guaranteed that the address of the structure remains the same between two `FETCH` statements, so to safely access the associated structure, the pointer `c1` always must be used.

The structure associated to an open cursor exhibits the following properties:

- The component `crsname` is the name of the cursor.
- The components `taid` and `dbid` are for use of ESQL and of no concern for the ESQL programmer.
- The component `qudescr` contains all information directly correlated to the query itself, i.e. types and values of result fields. It is described below.

6.1.2 The Datatype `Query_Descr`

The datatype `Query_descr` is defined as follows:

```

typedef struct {
    short    query_id ; /* used by ESQL */
    short    eod ;      /* used by ESQL */
    short    qtype ;    /* used by ESQL */
    short    updatable;
    short    qattr_no;
    Field    *field;    /* heap
array */
} Query_descr;

```

```

typedef struct {
    short    fieldtype;    /* code of type
*/
    Tspec    tspec;       /* fine
typedescription */
    char     fieldname [MAXIDENTSIZE+1];
    char     *fieldpointer;
} Field;

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 CHAR / BINCHAR */
} Tspec;    /* fine type specifiers
*/

```

Meaning of the components:

query_id

eod

qtype all used by ESQL, of no concern for the ESQL programmer

updatable is a flag: 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.

field is an array on heap; it is dynamically allocated and freed by ESQL; don't malloc or free on this pointer.

field[i].fieldtype For each field i, 0 <= i < qattr_no, field[i].fieldtype is the (encoded) data type. It is one of the following encoded values:

```

TINYINT
SMALLINT
INTEGER

```

BIGINT
 NUMERIC
 FLOAT
 DOUBLE
 CHAR
 BINCHAR
 BOOL
 DATETIME
 TIMESPAN
 BLOB

These values are symbolic constants defined in `tbx.h`; don't confuse these constants with the similar Transbase keyword notation for field types used in DDL statements or in TB/SQL queries.

<code>field[i].tspec</code>	This component is a finer description of the above described <code>field[i].fieldtype</code> . It is only defined if the type is one of the following: NUMERIC the precision and scale CHAR p for CHAR(p), 0 for CHAR(*) BINCHAR analogous DATETIME range: symbolic constants YY to MS TIMESPAN same
<code>field[i].fieldname</code>	contains the name of the i-th field if the i-th field is named (see TB/SQL Reference Manual), otherwise it contains the symbolic name <code>''Column_i''</code> .
<code>field[i].fieldpointer</code>	After a <code>FETCH</code> call where a tuple has been evaluated (<code>SQLCODE</code> is 0), the entry <code>field [i].fieldpointer</code> points to the i-th field value of the delivered tuple. If the i-th field value is the <code>SQL NULL</code> value, the entry is set to the language <code>C NULL</code> pointer. If the call has not delivered a tuple (result exhausted) all entries have undefined values. Note that the type of the pointer is <code>(char*)</code> so it must be casted to the appropriate field type before dereferencing it (see below).

6.1.3 Reading Field Values of a Dynamic Cursor

For a dynamic cursor `c1`, after each `FETCH` statement that did not trigger the `NOT FOUND` event or `SQLERROR` event (i.e. `SQLCODE` is zero) the field values of the current tuple can be read via the components `c1->qudescr.field [i].fieldpointer`. These

are pointers to the field values. Note carefully that a pointer is set to NULL if the corresponding field value is the SQL null value. Note also that the pointers are defined with type `char*`. Therefore the program must check the real type of the field and cast the pointer before dereferencing it. The following programs fragment would be suitable to print the field values after the `FETCH` statement of the program fragment in the beginning of this chapter.

```
for(i = 0; i < c1->qdescr.qattr_no ; ++i){
    char *valptr = c1->qdescr.field [i].fieldpointer
    if(valptr==NULL){
        printf("<NULL>" ); continue;
    }
    switch(c1->qdescr.field [i].fieldtype ){
        case TINYINT:
            printf("%d " , *(Tinyint*) valptr);
            break;
        case SMALLINT:
            printf("%d " , *(Smallint*) valptr);
            break;
        case INTEGER:
            printf("%ld " , *(Integer*) valptr);
            break;
        case BIGINT:
            printf("%lld " , *(Bigint*) valptr);
            break;
        case NUMERIC:{
            char num[50];
            printf("%s " ,
                fix_string( (Numeric*) valptr, num));
            break;
        }
        case FLOAT:
            /* not supported in TransbaseCD */
            printf("%e " , *(Float*) valptr);
            break;
        case DOUBLE:
            /* not supported in TransbaseCD */
            printf("%le " , *(Double*) valptr);
            break;
        case CHAR:
            printf("%s " , valptr);
            break;
        case BINCHAR:
            printhex(BINCHARARR(valptr));
    }
}
```

```

        /* user routine for hexadecimal printing */
        break;
    case BOOL:
        printf("%s ", *(Bool*) valptr?"TRUE":"FALSE");
        break;
    case DATETIME:{
        char dt[40];
        printf("%s ",
            tb_dt_datetime (dt, USA,(Datetime *)valptr));
        break;
    }
    case TIMESPAN:{
        char ts[40];
        printf("%s ",
            tb_ts_timespan (ts, USA,(Timespan *)valptr));
        break;
    }
    /* ... */
}
}

```

This program fragment also shows all aspects of type handling between SQL and C types as discussed in chapter "SQL Types and Host Variable Types" (see 2.6). (note also Table 2 there). First note that the symbolic C constants for the types closely correspond to the SQL type notations. Secondly note how the SQL- like predefined C types Tinyint etc. are used here to access the field values. Finally it must be kept in mind carefully how the types correspond to basic C types: as Integer corresponds to Int4, which is integer or long depending on your hardware, the format %d or %ld must be used, analogously

For the types NUMERIC, NUMBER, DATETIME and TIMESPAN see the corresponding chapters "Routines for ...".

6.2 Dynamic Executable ESQL Statements

With few exceptions, executable ESQL statements also can be built up in statement variables and run at runtime of the program. The syntax uniformly is:

Syntax:

```

Dynamic-ESQL-Statement ::=
    EXEC SQL #Statement_Var ;

```

```
Statement_Var ::=
    Identifier
```

Analogously to the statement variable for a dynamic cursor, its type must be appropriate to store a character string.

To run a dynamic executable ESQL statement, first store the text of the statement body into the host variable and then run it using the above syntax. The body of the statement is the executable ESQL statement as defined by its syntax but without the enclosing EXEC SQL and semicolon. Examples for statement bodies are:

- DROP TABLE suppliers
- DELETE FROM quotations WHERE suppno = 54
- COMMIT WORK
- SET TIMEOUT 20

6.2.1 Restrictions and their Remedies

Analogous to the Select-Statement of a dynamic cursor, it is not allowed to use a host variable reference inside the text of a **dynamic** executable ESQL statement. It follows that all those statements whose syntax require at least one host variable cannot be run dynamically. Those statements whose syntax admits an optional host variable reference can only be run without a host variable. However, for all these restrictions there exist remedies which lastly achieve the same effect. In the sequel a list of these restrictions together with their alternatives is given.

The following Executable-Statements have optional host variables which are disallowed when executed dynamically:

- Fetch-Into-Statement:
Read field values as described in chapter "Dynamic Cursors".
- Use-Transaction-Statement :
Build the statement with the explicit transaction identifier to be supplied.
- Use-Database-Statement :
Build the statement with the explicit database identifier to be supplied.

The following Executable-Statements are not allowed as dynamic statements:

- Select-Into-Statement :
Run an equivalent Select-Statement via a dynamic cursor.
- Send-Interrupt-Statement :
Can be included statically in the program and then executed at any time.

6.2.2 Control Information for Dynamic Statements

Sophisticated programs processing dynamic statements may need to capture all available control information which implicitly arrives:

- Connecting to a database delivers a database identifier in `sqlca.dbid`.
- Explicit or implicit start of a new transaction delivers a transaction identifier in `sqlca.taid`.
- Inserting, Updating or Spooling tuples delivers a count information in `sqlerrd[2]` and `sqlerrd[3]` as described in chapter "The SQLCA Structure".

`sqlerrd[0]` and `sqlerrd[1]` can be tested to capture these events.

6.2.2.1 The `sqlerrd[0]` Flag

After each statement, `sqlerrd[0]` is set to one of three symbolic constants:

NEW_DB_ID: A connection to a database has been performed and the database identifier can be read from `sqlca.dbid`.

NEW_TA_ID: A new transaction has explicitly or implicitly been started and the transaction identifier can be read from `sqlca.taid`.

NO_INFO: None of these events.

6.2.2.2 The `sqlerrd[1]` Flag

After each statement, `sqlerrd[1]` is set to one of the two symbolic constants:

COUNT_INFO: An Insert-, Update-, or Spool-Table-Statement has been processed and `sqlerrd[2]` and `sqlerrd[3]` thus have been assigned a count result as described in chapter "The SQLCA Structure".

NO_INFO: No such event

Chapter 7

The ESQL Text Expander

ESQL offers a text replacement facility which resembles the non-parameterized macro expansion of the C preprocessor. The ESQL Text Expander runs as a separate pass before the ESQL compiler. It can be switched off by a command line option.

For ease of discussion the following terminology is used: The space, tab and newline character are called **white space** characters. Within an ESQL source line, a sequence of characters without any white space which is enclosed by white space or which starts at the beginning of the line and is followed by white space is called a **word**.

The Text Expander interprets **text definition lines**. A text definition line is a source line whose first symbol (after arbitrary many blanks or tabs) is the Text Expander's text definition symbol followed by at least one blank or tab. The text definition symbol may be a single character or a sequence of characters without blanks and tabs. By default it is the dollar character (\$). It may be redefined by a command line option.

The first word after the definition symbol is the **head** of the text definition. The **body** is the remaining sequence of characters in the line starting with the first non-white symbol after the head and ending with the last character in front of the newline character. Note that the body may be empty or may consist of several words separated by white space.

The text definition symbol itself must not appear as the head or as any word of the body.

It is allowed to use text definition heads as a word in the body of a text definition. Direct or indirect recursion, however, is illegal and leads to an error.

The text expansion mechanism is made as a single linear pass over the source code producing a target code. Each text definition line is deleted in the target code. In all other source lines, each word which is equal to the head of any **already**

encountered text definition is replaced by the corresponding body. After each replacement, the body is again recursively searched for text definition heads until no further replacements can be made.

Note that unlike the C preprocessor only text definition heads which appear as words are replaced. This means that to enforce replacement the occurrence of the head must be enclosed in white space.

Example:

```
$ ARRAYSIZE 100
char array1[ARRAYSIZE];      /* would not be replaced!
*/
char array2[ARRAYSIZE];      /* would be replaced! */
```

With the ESQL Text Expander, short hand notations for often needed text pieces can be used. The most obvious example is the EXEC SQL clause at the beginning of each ESQL statement.

Example:

```
$ ## EXEC SQL
## BEGIN DECLARE SECTION;
    int h;
## END DECLARE SECTION;

## DECLARE c1 CURSOR FOR SELECT suppno FROM suppliers;

main()
{
    ## WHENEVER NOT FOUND GOTO end;
    ## OPEN c1;
    while(1)
        ## FETCH c1 INTO :h;
end:
    ## CLOSE c1;
    ..
}
```

Note that if the dollar sign \$ is itself desired as a text definition head, the Text Expander must be started with a command line option that changes the default text definition symbol to an alternative symbol or word, (e.g. ##). See chapter Compilation of an ESQL Program.

Chapter 8

Signal Handling : Send-Interrupt-Statement

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 ESQL programs. Imagine e.g. a situation where a semantically erroneous query has been issued which would last very long. Instead of trying to abort the whole process one can abort this transaction asynchronously, using a signal and the Send- Interrupt-Statement.

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

The Send-Interrupt-Statement is an executable statement which may be executed **asynchronously** at any time. Note that this ESQL statement is the only one which may be executed asynchronously by a user-defined signal handling procedure. If other ESQL statements were executed asynchronously, the ESQL runtime system and the Transbase kernel process could get totally confused.

Assume that there exists a user-defined signal handling procedure with name "ap_sighandler".

Executing the Send-Interrupt-Statement 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. When the interrupt function ap_sighandler is called due to a signal, one of the following two cases holds:

No ESQL statement has been interrupted when receiving the signal:

All active transactions are aborted by the Send-Interrupt-Statement immediately. The control flow continues after the Send-Interrupt-

Statement and `ap_sighandler` is free to decide its subsequent actions (for example a `longjmp`).

An ESQL statement has been interrupted when receiving the signal:

All active transactions are marked for abortion by the Send-Interrupt-Statement but remain active for the moment. A signal is sent to all Transbase processes which are associated to the application. 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.

Finally, the interrupted ESQL statement finishes with `SQLERROR` and with the special error code `ABORTSIGNAL` in `sqlcode` which is reserved for this event.

Note that in this case, the control flow continues after the interrupted ESQL statement and not after the Send-Interrupt-Statement. The event that a signal has arrived and the transactions have been aborted by the Send-Interrupt-Statement therefore must be caught by the error handling of the interrupted ESQL statement. The error handling then is free to decide about further actions (e.g. a `longjmp`).

The following program shows an example how to use the signal handling feature. The `SIGINT` signal is used to abort all active transactions and to jump to a specified point in the program marked by a jump buffer.

Example:

```
EXEC SQL WHENEVER SQLERROR CALL errorhandler;
ap_sighandler()          /* interrupt routine */
{
    Id state;
    signal (SIGINT, ap_sighandler);
    EXEC SQL SEND INTERRUPT;
    /* do what ever you like, e.g. */
    longjmp (&jmpbuf, 0);
}

...

main()
{
    ...
    signal (SIGINT, ap_sighandler);
    setjmp (& jmpbuf);
```

```

    ...
    < ESQL statements and C statements >
    ...
}

errorhandler()
{
    Int4 code = sqlca.sqlcode;
    if(code==ABORTSIGNAL)
    {
        printf('Interrupted ..\n');
        longjmp (&jmpbuf, 0);
    }
    else
    {
        State tastate = sqlca.tastate;
        printf('Error: %s\n',sqlca.tb_errtxt);
        printf('Transaction %s\n',
            state==TA_ACTIVE?'continues':'aborted');
    }
    ...
}
}

```

Since the ESQL runtime system does not handle any signals directly, there are no restrictions on ESQL programs to use a predefined signal for asynchronous transaction abort nor is a signal reserved for. The 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.

Note finally that the **SEND INTERRUPT** statement can only work correctly if the Transbase process "tserver" is running (see the "Transbase System Guide") otherwise an error is returned (and is written into sqlca.tb_errtxt). Furthermore, it is illegal to place the Send-Interrupt-Statement in the body of a function which is called by the user defined interrupt function. Only place it directly inside the user defined interrupt function.

Chapter 9

Modularization of ESQL programs

9.1 ESQL Modules

Like a C program, an ESQL program may consist of more than one module. A module is the compilation unit. There are, however, the following restrictions and additional rules:

- A cursor can only be used in the module where it is declared. That means, each Open-, Fetch-, Delete-Positioned-, Update-Positioned- and Close- Statement for a cursor in a module is only valid if the cursor is declared in the same module.
- A host variable which is to be used in more than one module must be treated like a corresponding C variable: exactly one module must define the variable and all modules which also use it must declare it with storage class extern.

9.2 Mixture of ESQL and TBX Calls

It is no problem to link TBX modules with ESQL modules.

It is recommended not to use ESQL calls and TBX calls within one and the same module. In the current Version of 4.2, however, a mixture of ESQL and TBX calls works as expected provided that the following slight complication which occurs in the PC version is considered:

After an erroneous ESQL statement, the error text is stored in `sqlca.tb_errtxt`. After an erroneous TBX call the error message is, of course, not stored in `sqlca.tb_errtxt`. A TBX programmer expects the text in `tb_errtxt`, but - if preprocessed by the

ESQL preprocessor in the PC version - must be accessed via the address delivered by function call `tb_errtxt_adr()`.

Chapter 10

Compilation of an ESQL Program

10.1 Command Syntax

The ESQL compiler is started with the `tbsp` command. The `tbsp` command accepts a list of source files with `.pp` extension. For each such source file a C source file with corresponding name and `.c` extension is produced. The C source files must then be translated by the C compiler like `tbx` application programs: the C sources need the include file `"tbx.h"` and must be linked with the `"tbx.a"` library. Both are located in the Transbase directory defined by the `"TRANSBASE"` environment variable.

Syntax:

```
tbsp [ options ] infile infile ...
```

Options: The `tbsp` command accepts the following options:

- g Prepare the ESQL source for debugging.
- 1 Only run the ESQL Text Expander. Output is produced in a file named `infile.pc`.
- 2 Switch off ESQL Text Expander. If text expansion is not needed then this option makes the compilation process faster. The input file must have a `.pc` extension.
- w Switch on warnings of the ESQL compiler.

- s word Define word as the text definition symbol for the Text Expander.
- o file Name the output file to file instead of infile.c
- Output is written to stdout. No -o option may be specified.
- nv No version prompt is displayed.
- ne Echo of erroneous source lines is suppressed.

Example: The following example shows a complete compilation process for a single

```
ESQL program named appl.pp:
tbpp [ options ] appl.pp

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

rm appl.c
```

Note: On most UNIX systems, the math library must be linked as shown above.

10.1.1 Sample Makefile

An alternate method to compile an ESQL source program would be to define the following rule in a "makefile":

```
.SUFFIXES: .o .pp

# Rule to transform a .pp source into a run module
.pp:
    $(TRANSBASE)/tbpp $<
    cc $*.c -I$(TRANSBASE) \
    $(TRANSBASE)/tbx.a -lm -o $@
    rm $*.c

# Rule to transform .pp sources into .o objects
.pp.o:
    $(TRANSBASE)/tbpp $<
    cc $*.c -I$(TRANSBASE) -c
    rm $*.c
```

10.2 Command Syntax on Windows Platforms

The ESQL compiler is started with the `tbp1` and `tbp2` command.

The `tbp1` command accepts one source files with `.pp` extension. The outputfile will have a corresponding name with `.pc` extension.

The `tbp2` command accepts one source files with `.pc` extension. The outputfile will have a corresponding name with `.c` extension.

For further information see the chapter "Application Development in the Programming Interface Tbx" Manual.

Syntax:

```
tbp1 [ options ] [file]
```

Options: The `tbp1` command accepts the following options:

- : output to stdout
- h: display this message
- ne: disable echoing of erroneous line after error message
- nl: disable generation of `#line` directives (useful to debug `.c` file)
- nv: disable version prompt
- o file": output to `<file>`
- s esc_symb: set escape symbol to `<esc_symb>`
- w: enable warnings

Syntax:

```
tbp2 [ options ] [file]
```

Options: :

The `tbp2` command accepts the following options:

- : output to stdout
- g: generate a `#line` directive after each line (useful to debug `.pp` file)

- h: display this message
- ne: disable echoing of erroneous line after error message
- nl: disable generation of #line directives (useful to debug .c file)
- nv: disable version prompt
- nn: for internal use of tbpp only
- o file: output to <file>
- w: enable warnings

10.2.1 Sample Makefile

An alternate method to compile an ESQL source program would be to define the following rule in a "makefile":

```
# Rule to transform a .pp source into a run module
sample.c: sample.pc
    tbp2.exe -nv -o sample.c sample.pc
sample.pc: sample.pp
    tbp1.exe -nv -o sample.pc sample.pp
sample.obj: sample.c
    cl /c sample.c sample.obj
```

Chapter 11

Debugging an ESQL Source

In an UNIX environment with a debugger like dbx or xdb or cdb, there are two ways to debug an ESQL program. Assume that the ESQL program is stored in a file with name appl.pp.

One possibility is to debug the precompiled C source appl.c produced by the ESQL compiler. Note, however, that all ESQL statements are replaced by pure C constructs. Additionally, there are line directives produced by the ESQL compiler to adjust the error messages of the C compiler to line numbers of the original ESQL source. **These line directives must be deleted** before starting the debugging process to avoid that the debugger gets confused about the current line in the program. Of course, the C source appl.c must be compiled with the -g debug option as usual.

A more convenient way is to debug the original ESQL source. To achieve this the ESQL compiler as well as the C compiler must be supplied the -g debug option. On some UNIX systems this is all that has to be done because the debugger automatically opens the correct file appl.pp. However, it may be that additionally either the appl.c has to be removed or even the ESQL source file appl.pp must be renamed (copied or moved) to appl.c.

Chapter 12

Appendix A: Sample Programs

Two sample programs and their corresponding makefile are shown here. One program is named `create` and consists of a single module, the other one is named `phones` and consists of three modules `insert`, `list`, and `.`

The program `create` the database tables accessed by `phones`. The program `phones` is a program which allows to retrieve or insert phone numbers of persons.

`phones` is structured as follows: The module `phones` provides the function `main` which is the control program of `phones`. The module `list` provides a function `listphones(patt)` where `patt` specifies a name pattern used to restrict the set of all phone numbers by a `LIKE` predicate. The pattern `'%'` serves to retrieve all phone numbers. The module `insert` provides a function `insertphone(name, phone)` which inserts a new row into the table `Phones`.

`create.pp`

```
main()
{
    exec sql whenever sqlerror continue ;
    exec sql connect 'uni@sunsv' ;
    exec sql login 'tbadm' ;
    exec sql drop table Phones ;
    exec sql create table Phones
        ( Name CHAR(*),
          Phones CHAR(*) )
        key is Name ;
    exec sql commit work ;
    exec sql disconnect ;
}
```

makefile:

```

.SUFFIXES: .o .pp

MOBJ = phones.o insert.o list.o
COBJ = create.o

.pp.o:
    $(TRANSBASE)/tbp1 $.pp
    $(TRANSBASE)/tbp2 $.pc
    cc $.c -I$(TRANSBASE) -c
    rm $.pc $.c

all: phones create

phones: $(MOBJ)
    cc $(MOBJ) $(TRANSBASE)/tbx.a -lm -o $@

create: $(COBJ)
    cc $(COBJ) $(TRANSBASE)/tbx.a -lm -o $@

print: phones.pp insert.pp list.pp create.pp makefile
    pr $? | lp
    touch print

```

phones.pp:

```

main()
{
    char cmd [20], name [20], phone[20], yn[20];

    EXEC SQL CONNECT 'uni@sunsv';
    EXEC SQL LOGIN 'tbadm' ;

    listphones("%"); /* Use this pattern to list all */
    while (1) {
        printf ("\n? ");
        if (gets(cmd)!=cmd) break;
        switch (*cmd) {
            case 'q': goto disconn;
            case 'g': printf ("Name "); gets(name);
                    listphones(name);
                    break;
            case 'i': printf ("Name "); gets(name);

```

```

        printf ("Phone "); gets(phone);
        if (insertphone(name, phone) == 0) {
            printf ("Hit return to confirm ... ");
            if (*gets(yn) == '\0')
                EXEC SQL COMMIT WORK ;
            else
                EXEC SQL ROLLBACK WORK ;
        }
        break;
    default:    printf ("Quit / Insert / Get\n");
               break;
    }
}

disconn:
    EXEC SQL DISCONNECT ;

}

/* Note: in this module no error handling is
defined, so the default "ignore" applies.
This is important for tb_error, since the DISCONNECT
statement may find an error which should be ignored.
*/
tb_error()
{
    puts(sqlca.tb_errtxt);
    EXEC SQL DISCONNECT ;
    return -1;
}

```

list.pp:

```

                /* global error handling */
exec sql whenever sqlerror call tb_error;

listphones(patt)
exec sql begin declare section ;
char * patt;
exec sql end declare section ;
{
    exec sql begin declare section ;
    char name[80], phone[80];

```

```

exec sql end declare section ;

exec sql declare c0 cursor for
select * from Phones where Name like :patt;

exec sql open c0;
exec sql whenever not found goto endlis;
while (1) {
    exec sql fetch c0 into :name, :phone;
    printf ("%s-%s\n", name, phone);
}
endlis:
exec sql close c0;
exec sql rollback work;
}

```

insert.pp:

```

insertphone(name, phone)
EXEC SQL BEGIN DECLARE SECTION ;
char *name;
char *phone;
EXEC SQL END DECLARE SECTION ;
{
    /* special local error handling */
    EXEC SQL WHENEVER SQLERROR GOTO err4;
    EXEC SQL INSERT INTO Phones VALUES
    ( :name, :phone ) ;
    return 0;

err4:
    printf (Cannot insert %s, %s\n", name, phone);
    return -1;
}

```

Chapter 13

Appendix B: Database Schema SAMPLE

The database "SAMPLE" used in the examples throughout this manual consists of three tables named SUPPLIERS, PARTS and INVENTORY the structure and contents of which is given in the following tables:

suppliers		
suppno	name	address
51	DEFECTO PARTS	16 BUM ST., BROKEN HAND WY
52	VEUVIUS, INC.	512 ANCIENT BLVD., POMPEII NY
53	ATLANTIS CO.	8 OCEAN AVE., WASHINGTON DC
54	TITANIC PARTS	32 LARGE ST., BIG TOWN TX
57	EAGLE HARDWARE	64 TRANQUILITY PLACE, APOLLO MN
61	SKY PARTS	128 ORBIT BLVD., SIDNEY
64	KNIGHT LTD.	256 ARTHUR COURT, CAMELOT

inventory		
partno	description	qonhan
207	GEAR	75
209	CAM	50
221	BOLT	650
222	BOLT	1250
231	NUT	700
232	NUT	1100
241	WASHER	6000
285	WHEEL	350
295	BELT	85

quotations				
suppno	partno	price	delivery_time	qonorder
51	221	.30	10	50
51	231	0.10	10	0
53	222	0.25	15	0
53	232	0.10	15	200
53	241	0.08	15	0
54	209	18.00	21	0
54	221	0.10	30	150
54	231	0.04	30	200
54	241	0.02	30	200
57	285	21.00	4	0
57	295	8.50	21	24
61	221	0.20	21	0
61	222	0.20	21	200
61	241	0.05	21	0
64	207	29.00	14	20
64	209	19.50	7	7