Preface

DRIVE/WINDOWS allows you to access the SQL database system using SQL statements.

While this directory gives you a brief description of the syntax of the DRIVE SQL statements for UDS, a detailed description is provided in the "SQL for UDS/SQL V1.0 Language Reference Manual" [13]. This directory describes the DRIVE SQL statements for UDS of the DRIVE/WINDOWS version 1.1 for BS2000 and SINIX.

Complex statement sections which occur in both DRIVE and SQL statements are described separately in the chapter on "Metavariables" in the "DRIVE/WINDOWS V1.0: System Directory" [3]. Additional DRIVE SQL metavariables which are not included in this chapter are described in this statement directory under "UDS metavariables". The term "with the TP monitor" refers to UTM mode in BS2000.

SQL return codes are accepted by UDS/SQL and output by DRIVE/WINDOWS as error messages in the form DRI9xxx. For their meaning, refer to the "SQL for UDS/SQL (BS2000) V1.0: Language Reference Manual" [13].

UDS/SQL statements

CLOSE Close the cursor

CLOSE closes a cursor that you declared with the DECLARE statement for a query expression and opened with the OPEN statement.

The cursor declaration is retained. You must open the cursor using the OPEN or RESTORE statement before accessing the cursor again using the FETCH statement. An open cursor is closed at the end of a transaction.

In program mode, cursors can only be referenced in the source file in which they were declared.

```
CLOSE cursor
```

cursor	Name of the cursor you wish to close.

It makes sense to issue a CLOSE statement followed by an OPEN statement if, for instance, you have used variables in the *query-expression* when declaring the cursor. When the OPEN statement is issued, the current values for the variables are assigned to *query-expression* and the active set is filled with the current data from the database.

COMMIT WORK Terminate a transaction

COMMIT WORK terminates a transaction and commits all the updates performed on the database since the beginning of the transaction to the database. The first error-free SQL statement after the COMMIT WORK statement starts a new transaction.

COMMIT WORK closes all cursors opened during the transaction. A cursor defined with TEMPORARY in a DRIVE program is deleted at a higher program level when the next COMMIT WORK is issued.

Recommendation:
In the DRIVE program, the COMMIT WORK statement should immediately follow the declarations in order to commit the declarations to the database.
COMMIT WORK may only be used in a loop which performs cursor processing if the cursor is stored with STORE and restored with RESTORE (see the example).

```
                 ⎡display     ⎤
COMMIT WORK [WITH⎨send message⎬]
                 ⎣stop        ⎦
```

WITH	WITH allows you to specify a statement to be executed after the end of the transaction. If WITH is not specified in an interactive program of an applicattion with a TP monitor, the transaction is terminated, and the program continues to run in the successor subprogram without screen output. WITH may only be specified in program mode.
display	Displays a form. The following statements may be used: – DISPLAY screen-form (see DISPLAY screen-form statement in the "DRIVE/WINDOWS: System Directory" [3]). – DISPLAY form-name (see DISPLAY form-name statement in the "DRIVE/WINDOWS: System Directory" [3]). – DISPLAY FORM (see DISPLAY FORM statement in the "DRIVE/WINDOWS: System Directory" [3]).
send message	Sends messages (see SEND MESSAGE statement in the "DRIVE/WINDOWS: System Directory" [3]).
stop	Stops DRIVE execution (see STOP statement in the "DRIVE/WINDOWS: System Directory" [3]).

Rules

– If the programmer does not explicitly terminate the transaction with COMMIT WORK, ROLLBACK WORK or with UTM language elements, it is rolled back implicitly when the run unit ends. In this case, DRIVE outputs an error message.

– If the current transaction was opened with a SET TRANSACTION statement, COMMIT WORK resets the end-of-transaction status defined by SET TRANSACTION to the default value.

Example of cursor processing in a loop:

```
 DECLARE c1 ...
   .
   .
CYCLE c1 INTO &var.*;
   .
   .
 STORE c1;
 COMMIT WORK;
 RESTORE c1;
   .
   .
END CYCLE;
```

CREATE TEMPORARY VIEW Declare a view

CREATE TEMPORARY VIEW declares a view, i.e. a database query stored under a given name.

A view is no longer valid in the following circumstances:

- when the program is terminated
- if the program is aborted
- When DRIVE is terminated (STOP)
- if DROP TEMPORARY VIEW *view* or DROP TEMPORARY VIEWS is issued (only possible in interactive mode or within the EXECUTE statement if the view was also declared using EXECUTE).

```
CREATE TEMPORARY VIEW view [(column,...)] AS query-expression
```

view
identifies the view.
PLAM_DIRECTORY must not be specified for *view*. All view names must be unique within the compilation unit. A view must not have the same name as a base table.
No two views with the same names can be declared on a single program level or in interactive mode.

(column,...)
declares new names for the columns within the view.

The number of columns specified must match the number of columns returned by *query-expression*.
If you declare a new name for a vector, the new name must be used in the form *vector (index1[..index2])* when referring to the view. If you specify structured columns in *query-expression*, you can reference their elements in the view using the names they have in the base table.

You must specify *column* if the names of the columns returned by *query-expression* are not unique within the view (due to join fields, for example) or if no name exists, as in the case of arithmetic expressions, set functions or constants.

If no names are specified, the names of the columns returned by *query-expression* are used.

query-expression
derives the view from existing tables (see the metavariable *query-expression*).

The *query-expression* is subject to the following restrictions:

- No *variable* may be specified in a *query-expression* in the CREATE TEMPORARY VIEW statement.
- Only names of base tables may be specified in the FROM clause for *table*.

Updatable view

Only an updatable view can be used to perform updates on the underlying base table. A view is updatable if *query-expression* is updatable. The contents of an updatable view are thus the same as the corresponding section of the underlying base table.

Rule

- The view must be declared in the program text before any non-declarative statements or declarations that reference it. No view with the same name should have been released in the current transaction before the declaration of this view. The view declaration is only valid within one compilation unit.

DECLARE... CURSOR FOR... Declare cursor

DECLARE declares a cursor and assigns it a *cursor-specification* that defines the active set. The cursor must be declared in the declaration section. The declaration is valid throughout a compilation unit.

You can use a cursor to access individual rows in the active set. The current row indicated by the cursor can be read, deleted or updated. Updates or deletions performed on the active set are also performed on the underlying base table.

Recommendation:
In the DRIVE program, the COMMIT WORK statement should immediately follow the declarations in order to commit the declarations to the database. Control statements for transactions are permitted at different program levels, but can lead to seemingly erratic program behavior.

Scope of validity of a cursor:
A cursor ceases to be valid
- at the end of a program
- when a program is aborted
- when DRIVE terminates (STOP)
- DROP CURSOR *cursor* (DRIVE statement, in interactive mode, dynamically or with variable cursor)
- DROP CURSORS (DRIVE statement, only in interactive mode or dynamically)

When you switch from DRIVE interactive mode to program mode, the cursor definition remains valid, but the cursor position is lost (you can, however, store the cursor position with the STORE statement).

In DRIVE program mode, a cursor defined with PERMANENT remains valid beyond the end of a program invoked with CALL, and its position is retained.

A cursor defined with TEMPORARY is closed when the program is terminated and deleted if a COMMIT WORK statement is issued on a higher program level.

A cursor always ceases to be valid in program mode when you switch to interactive mode, if the program is aborted and when DRIVE is terminated (when the STOP or COMMIT WORK WITH STOP statement is issued).

```
                 ⎡PERMANENT⎤
DECLARE cursor  ⎨          ⎬ CURSOR [FOR cursor-specification]
                 ⎣TEMPORARY⎦
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
cursor-specification::= query-expression

                                      ⎡integer⎤  ⎡ASCENDING ⎤
                           [ORDER BY {⎨       ⎬ [⎨          ⎬]},...]
                                      ⎣column ⎦  ⎣DESCENDING⎦
```

cursor Name of the cursor.
All cursor names must be unique within a compilation unit. No two cursors with identical names may be declared at the same time at one program level or in interactive mode.

PERMANENT
PERMANENT can only be specified within programs called using CALL. The cursor's position is retained after the termination of a program invoked with CALL if no COMMIT WORK statements are permitted in the called program or in the calling program between the CALL statements. The cursor position only ceases to be retained if the cursor is closed explicitly with CLOSE, or the user switches to interactive mode.

i When the program invoked with CALL runs for the first time, the cursor must be opened with the OPEN statement. Whenever it executes subsequently, no OPEN statement may be issued for that cursor.
The calling program must not contain a COMMIT WORK statement.

<u>TEMPORARY</u>
TEMPORARY can only be specified in programs called using CALL. The cursor is closed at the end of the CALLed program and its position is lost. The cursor ceases to be valid when the next COMMIT WORK statement is issued at a higher program level.

FOR clause The FOR clause may only be omitted for a static cursor declaration in program mode. If it is omitted, a "variable cursor" is declared. A cursor of this kind is only recognized by the database system if a subsequent dynamic declaration with a FOR clause is made. In the case of a dynamic declaration with EXEC, you must specify the FOR clause. Apart from the FOR clause, the static and dynamic declaration of a cursor must be identical. All other cursor statements (OPEN, FETCH, DROP, CYCLE) may be specified statically for the variable cursor, which improves performance (see the chapter on "Dynamic SQL statements" in the "DRIVE Supplement" [50] or "DRIVE Programming Language [2]).

query-expression

defines the active set by selecting columns and rows from existing base tables or views. Variables are evaluated when the cursor is opened with OPEN.

ORDER BY

sorts the rows of the active set in ascending or descending order according to the values of the columns specified by *integer* or *column*. When rows are sorted using the ORDER BY clause, a null value is considered to be "less than" a non-null value. ORDER BY may be specified only if *query-expression* is updatable.

If two or more columns were specified, sorting is performed initially on the basis of the values in the first column specified.

The columns specified must not identify primary or foreign keys, neither may they be structured.

An ORDER BY clause may specify sorting either in ascending order only or descending order only.

column

identifies a column (see the metavariable *column*) which is to be taken as the basis for sorting. *column* must occur in the *select-list* of *query-expression* and must not be qualified with *table*.

integer

identifies the position of the column to be used as the basis for sorting within the active set (1 ≤ integer ≤ number of columns).
The columns are numbered from left to right, starting with 1, in the order of their entry in *select-list* of *query-expression*. This allows you to sort according to unnamed columns such as those containing the result of calculations.

integer must lie between 1 and the number of columns in the result table. The order of the columns is defined by *query-expression*.

<u>ASCENDING</u>

sorts the values of the specified column in ascending order.

DESCENDING

sorts the values of the specified column in descending order.

Updatable cursor

Only updatable cursors can be used with the UPDATE... WHERE CURRENT OF... or DELETE... WHERE CURRENT OF... statements to perform updates or deletions. A cursor is updatable if *cursor-specification* is of the form *query-expression* and if *query-expression* is updatable (see the metavariable *query-expression*).

Rules

- A given cursor must be declared in the DRIVE program text before all non-declarative statements and before all declarative statements that use that cursor. Also, it is not permissible to declare multiple cursors with identical names within one program level or in interactive mode. The cursor can only be used where it is valid (see above for the scope of validity for a cursor.
- A cursor declared within a transaction can be used, i.e. opened, read with FETCH and closed, only within that transaction since COMMIT WORK and ROLLBACK WORK close all open cursors (you can, however, use the STORE statement to store the cursor position for subsequent access and reopen it with the RESTORE statement).
- Columns containing the results of calculations receive no name. The data type they acquire is determined by the arithmetic operation and the data types of the operands (see the metavariable *sql-expression*).

DELETE... WHERE condition Delete rows

DELETE... WHERE *condition* deletes rows from a base table. You can also use this statement on an updatable view.

```
DELETE FROM table [WHERE condition]
```

table	identifies a base table or updatable view (see the metavariable *table*).
condition	Each row is checked to determine whether *condition* is satisfied (see the metavariable *condition*). Rows satisfying the specified *condition* are deleted. If no condition is entered, this statement deletes all the rows in the specified base table or all the rows selected by the view.

Rules

– If no row satisfies *condition*, &SQL_CODE=100 is set.

– If an error occurs at the nth row (n>1) during processing of the statement DELETE... WHERE condition, the transaction is rolled back by the database system. If the error occurs in the first row, only the current statement is rolled back.

DELETE... WHERE CURRENT OF... Delete the current row of the cursor

DELETE... WHERE CURRENT OF... deletes the current row of the cursor from the underlying base table.

The cursor is then positioned in front of the next row. To move the cursor to the next row in the active set, you must enter a FETCH statement.

```
DELETE FROM table WHERE CURRENT OF cursor
```

table	identifies a base table or updatable view (see the metavariable *table*). The specified table name must match the table name in the FROM clause of the relevant *cursor* declaration.
cursor	identifies a cursor.

Rules

- The cursor must have been declared beforehand with DECLARE.
- The cursor
  - must have been opened with an OPEN statement and positioned to a row with FETCH, or
  - must have been opened with a RESTORE statement, in which case the cursor is pointing to the same row as when the STORE statement was executed.
- The cursor must be updatable.

DROP CURSOR Dropping a cursor

The DROP CURSOR and DROP CURSORS statements release cursors. The statements are permitted in interactive mode. In program mode, they are only allowed dynamically in EXECUTE statements, and then refer to one or all dynamically declared cursors. The only exception is for a variable cursor, where DROP CURSOR *cursor* is allowed statically. A variable cursor can only be deleted explicitly with DROP CURSOR *cursor*. When a DROP is performed on a variable cursor, the variable select or insert condition (FOR clause) is deleted. This means that you can redeclare the select or insert condition of the cursor after a DROP statement. A COMMIT WORK is required between the DROP and redeclaration.
The DROP CURSOR(S) statement is subject to transaction management, i.e. if you specify ROLLBACK WORK, the cursors specified in DROP CURSOR(S) are not released.

```
     ⎧CURSOR cursor⎫
DROP ⎨             ⎬
     ⎩CURSORS      ⎭
```

cursor	The cursor specified by *cursor* is released. If the cursor is open, an implicit CLOSE is issued.
CURSORS	All the cursors defined in interactive mode or all the cursors defined at the same program level with EXECUTE are released. The DROP CURSORS statement corresponds to a sequence of DROP CURSOR *cursor* statements where the cursor names are specified explicitly.

Example

```
DCL c1 CURSOR;
EXEC ’DCL c1 CURSOR FOR’ || &SEARCH1;
...
...
DROP CURSOR c1;
COMMIT WORK;
EXEC ’DCL c1 CURSOR FOR’ || &SEARCH2;
```

DROP TEMPORARY VIEW Dropping a view

The DROP TEMPORARY VIEW and DROP TEMPORARY VIEWS statements release views in DRIVE’s interactive mode or within EXECUTE (only if the view was also declared within EXECUTE).
The DROP TEMPORARY VIEW(S) statement is subject to transaction management, i.e. if you specify ROLLBACK WORK, the temporary views specified in DROP TEMPORARY VIEW(S) are not released.

```
                ⎧VIEW view⎫
DROP TEMPORARY  ⎨         ⎬
                ⎩VIEWS    ⎭
```

view	The view specified by *view* is released together with all the cursors associated with the view.
VIEWS	All the views defined in interactive mode or all the views defined at the same program level with EXECUTE are released together with the cursors associated with them. The DROP TEMPORARY VIEWS statement corresponds to a sequence of DROP TEMPORARY VIEW *view* statements where the view names are specified explicitly.

FETCH Position the cursor and supply variables with values from columns

FETCH positions the cursor to the next row in the active set and makes that row the current row. In program mode, the column values in the current row are passed to the variable(s) specified. In interactive mode, the column values in the current row are displayed on the terminal.

The current row can be updated or deleted with subsequent UPDATE... WHERE CURRENT OF... or DELETE... WHERE statements if the cursor is updatable (see DECLARE... CURSOR FOR...).

```
FETCH cursor INTO variable,...       (Program mode)
```

```
FETCH cursor                         (Interactive mode)
```

cursor	is the name of the cursor.
INTO	passes the column values of the current row to the variables.
variable	identifies a variable (see the metavariable *variable*). The first column value is assigned to the first variable, the second column value to the second variable, and so on.

Rules

– The cursor must be declared beforehand with DECLARE.

– The cursor must have been opened with the OPEN or RESTORE statement.

– The number of variables must be equal to the number of columns in the *cursor-specification* of the cursor declaration. Also, the data types of the variables must be compatible with those of the corresponding columns.

– If an error occurs, &SQL_CODE receives a negative value. The individual variables are not assigned new values, i.e. the old contents are retained.

INSERT Insert a row

INSERT adds a row to an existing base table. You can also specify this statement for an updatable view (see CREATE TEMPORARY VIEW).

```
INSERT INTO table [(column,...)]

                 ⎡value⎤
       VALUES (⎨NULL ⎬,...)
                 ⎣*    ⎦

       [RETURN INTO variable]
```

table	identifies a base table or updatable view (see the metavariable *table*).
(column,...)	identifies the columns into which the VALUES clause is to enter values. If the entry is omitted, all the columns in the specified base table or view are included. The order of the columns is determined, in the case of a base table, by the definition of that *table*, and in the case of a view, by the order you defined in the view declaration. *column* must not be qualified. The columns must be located in the specified table. If you do not specify all the columns of the table involved, the columns of the inserted row that are not specified will receive either the default value or the null value, depending on the schema definition. The database system assigns a value to the system-defined primary key. A column may not be specified more than once, not even as an element of a structured column.
VALUES	assigns the specified values to the columns. The values are assigned to the individual columns in the order given. The number of values specified must match the number of columns specified. If no columns are specified, the number of values must match the number of columns of the highest level. In the case of a structure, this means that the values for the elements of the structure are not specified individually. An *aggregate* must be declared for *value*.
value	specifies the value which the column is to receive (see the metavariable *value*). If the column is structured, the value must have the same structure.

NULL	A keyword used to assign the null value to a column.
*	specifies that the value for a system-defined primary key is assigned by the database system. * must be specified if and only if the corresponding column is the system-defined primary key.
RETURN INTO	declares that the primary key value of the inserted row is to be passed to the variable. No significant places may be lost. RETURN INTO is permitted only if a system-defined primary key exists for the specified table.
variable	identifies a variable (see the metavariable *variable*). The *variable* must have been declared as numeric.

OPEN Open a cursor

The OPEN statement opens a cursor.

OPEN evaluates the cursor specification that you specified in the cursor declaration, using the current values of the variables specified in *cursor-specification*. OPEN positions the cursor before the first row of the active set.

One of the following statements is used to close a cursor:

- CLOSE *cursor*
- COMMIT WORK
- DROP CURSOR or DROP CURSORS
- DROP TEMPORARY VIEW or DROP TEMPORARY VIEWS, if the cursor refers to a view.

```
OPEN cursor
```

cursor	name of the cursor.

Rules

- The cursor must be declared beforehand with DECLARE.
- The cursor must be closed.
- A position for this cursor saved with STORE in a preceding transaction is lost.

PERMIT Declare a user ID

PERMIT declares the user ID for a relational schema used in a UDS data base.

If user identification and the assignment of a password-protected database are needed during the execution of a DRIVE program, the PERMIT statement must be issued before the program is invoked.

Mask-driven specification:

If PERMIT OFF is not specified, the UTM leader routine automatically outputs a PERMIT input mask on the screen following entry of the TAC. The entry fields in this screen mask are filled with blanks. If part of the mask or the whole mask is filled in, a PERMIT statement is generated internally using the specified values or default values.
The PERMIT mask can be filled any number of times as long as ’R’ (default) is set in the bottom line. If ’N’ is set, a PERMIT statement is generated and executed. Following this, the screen mask is deleted. If ’S’ is specified, the mask is terminated and no PERMIT statement generated.

Default declaration

Until you specify a PERMIT statement for a schema, the following default declaration applies:

– User group:	SQLGRP
– User name:	SQLUSR
– Password:	SQLPWD

```
                ⎡schema  ⎤
PERMIT SCHEMA=⎨         ⎬
                ⎣variable⎦

      [USERGROUP=user-group]
      [USERNAME=user-name]
      [PASSWORD=password]
```

schema	is the name of the UDS schema.
variable	Variable containing the name of the schema or the UDS database.

USERGROUP=user-group
assigns a user group (see the metavariable *value*). *user-group* must be specified as a variable, or as a character or hexadecimal literal (up to 8 characters).
user-group must conform to the UDS conventions.
In programs, *user-group* must be specified as a variable.

USERNAME=user-name
assigns a user name (see the metavariable *value*). *user-name* must be specified as a variable, or as a character or hexadecimal literal (up to 24 characters).
user-name must conform to the UDS conventions.
In programs, *user-name* must be specified as a variable.

PASSWORD=password
assigns a password (see the metavariable *value*). *password* must be specified as a variable, or as a character or hexadecimal literal.
password must not exceed 48 characters in length and must conform to the UDS or SESAM conventions for passwords.
In programs, the password must be specified as a variable.

Rules

- Only one relational schema may be referenced for each UDS database in a transaction.
- The PERMIT statement must be executed in a transaction before any other SQL statements that reference the same relational schema.
- A second PERMIT statement for a relational schema is only permitted within a transaction if the relational schema has not yet been accessed.

- If you are working with two or more relational schemas from different UDS databases and the user ID differs from the default declaration, you must enter a PERMIT statement for each schema.
- The specification remains in effect until
  - the end of the run unit or UTM conversation,
  - the next PERMIT statement is executed for the same schema, or
  - the transaction is rolled back with the ROLLBACK WORK statement. If the transaction is rolled back, either the last valid user ID for the schema is in effect, or the default declaration.

PERMIT OFF Suppress a UTM input mask

PERMIT OFF may only be used in the UTM start procedure.

The statement suppresses the input mask for user identification (PERMIT) which is displayed by default in the UTM leader routine.

```
PERMIT OFF
```

RESTORE Restore a cursor

RESTORE restores a cursor stored with the STORE statement.

This statement has the following effects:

- It opens the cursor. In contrast to the OPEN statement, it does not load the current values of the variables; the values are the same as when this cursor was last opened with OPEN.
- The cursor points to the same row as it pointed to when the STORE statement was executed in a previous transaction.
- Afterwards, no cursor position is stored for the cursor.
  RESTORE can therefore not be specified again without another STORE being entered beforehand.

```
RESTORE cursor
```

cursor	identifies the cursor. The cursor position must have been saved with STORE in a previous transaction within the same conversation.

Rules

- You cannot restore the cursor if the stored cursor was rendered invalid after the STORE statement with one of the following SQL statements:

OPEN	open a cursor
RESTORE	restore a cursor

  This is also true if the transaction in which the OPEN or RESTORE statement was executed is rolled back.
- The cursor position stored with STORE may be lost if the following occurs in the current or a foreign transaction
  - the current row is deleted
  - the current row is updated in such a way that it no longer satisfies the WHERE *condition* in *cursor-specification* of the associated cursor declaration.
- Entering a second STORE statement for the same cursor overwrites the previously stored cursor.
- The current row can be updated with UPDATE... WHERE CURRENT OF... or deleted with DELETE... WHERE CURRENT OF... following RESTORE.
  A FETCH statement can be used to read the row following the current row.

ROLLBACK WORK Roll back a transaction

ROLLBACK WORK terminates a transaction and undoes all the updates performed on the database since the start of the transaction. Under UTM, the save area is reset to the last synchronization point after a ROLLBACK WORK statement.

A cursor stored before the transaction was opened cannot be restored if it was processed with OPEN or RESTORE during the transaction reset with ROLLBACK WORK. In this case, the cursor must be reopened with OPEN if it is to be used.

Any cursors opened within the transaction are closed.

If an open transaction is not terminated before the end of a program which executes dynamically, DRIVE performs a ROLLBACK WORK. The program aborts and outputs an error message.

The first error-free SQL statement after the ROLLBACK WORK statement opens a new transaction.

```
ROLLBACK WORK [WITH RESET]
```

WITH RESET	is only permitted in program mode and only applies to DRIVE system control. The program rolls back to the status prior to the last COMMIT WORK and continues with the statement which follows that COMMIT WORK. The contents of the DRIVE variables are reset to the values valid at the last COMMIT WORK. Moreover, with the DRIVE/WINDOWS(SINIX) version all windows which are open after the COMMIT WORK statement are closed (see the section on transaction management or Window-4GL applications in the chapter on "Transaction procesing" in the manual "DRIVE/WINDOWS: Programming Language [2]). If no COMMIT WORK has been issued since the start of the program, the program aborts and outputs an error message. As far as the database is concerned, the two statements ROLLBACK WORK and ROLLBACK WORK WITH RESET are identical.

If the database system reports that it has rolled back the current transaction implicitly (automatically), both the database transaction and the DRIVE transaction are rolled back.

Rule

- If a PERMIT statement was specified in the current transaction, either the user ID that was valid for the schema beforehand or the default declaration is again valid following a rollback.

i If the ROLLBACK WORK WITH RESET statement is specified without a condition, there is a potential risk of an endless loop, since the DRIVE program is continued with the statement that follows the last COMMIT WORK statement.

SELECT Retrieve data

SELECT retrieves a row from base tables or views. You can link rows from two base tables (join). In program mode, SELECT transfers the values of the columns to the variables specified with INTO. In interactive mode, SELECT displays the values on screen.

The number of rows found by the SELECT statement must not exceed 1. If you want to read more than one row, you must use a cursor.

```
SELECT [ALL] select-list

       [INTO variable,...]

       FROM {table-specification},...

       [WHERE condition]
```

<u>ALL</u>	Default value. This operand returns all the rows selected with WHERE *condition*. Even duplicate rows can be selected. Please note that an error occurs each time duplicates are present as the result set of the SELECT statement may contain only one row.
select-list	Use a *select-list* to specify the columns in the result table (see the metavariable *query-expression*).
INTO	You use INTO to assign the column values of the (single-row) result table to variables. INTO is only valid in program mode.

variable	identifies a variable (corresponds to the term *variable* in the "SQL for ISO/SQL (BS2000) Language Reference Manual" [11]). The first variable is assigned the first column value in the result set, the second variable the second column value, and so on. The number of columns in the *select-list* must match the number of variables. Structured columns must be assigned to variables with the same structure. The data types of the variables and the corresponding columns in the *select-list* must be compatible (see the "DRIVE/WINDOWS: Programming Language Reference Manual" [2], chapter on "Using variables and constants", section on "Data type conversion compatibility"). If no row could be determined, &SQL_CODE receives the value 100 and no values are assigned to the variables.
FROM	You use the FROM clause to specify the base tables or view from which data is to be selected for the result table (see the metavariable *query-expression*).
WHERE	In the WHERE clause, you specify the conditions to be applied in selecting rows for the result table. The result table contains only rows which fulfill the specified conditions (see the metavariable *query-expression*).

SET TRANSACTION Set consistency level

SET TRANSACTION sets the consistency level and status for a transaction.
The higher the consistency level, the lower the degree of transaction concurrency.

```
                                  ⎡2⎤
SET TRANSACTION CONSISTENCY LEVEL {  }
                                  ⎣3⎦
```

If the SET TRANSACTION statement is not specified, the transaction has the consistency level 2.

The following applies in transaction processing:

– A transaction is either executed in its entirety or not at all.

– No updates can be "lost". The following situation could result in "lost updates": if transaction-1 updates a row that is then updated by transaction-2, the second update could also be lost if transaction-1 is rolled back.

– Updates performed in a transaction that is still open are not accessible by any other transaction, i.e. "dirty read" is not possible.

Depending on the consistency level set, the following phenomena may occur if two transactions access a row concurrently:

– "Non-repeatable read"
One transaction accesses a row "reading without locking" (consistency level 0 or 2) that is then updated by a second transaction. The second transaction is terminated and the update committed to the database. This row is then no longer in its original state if reread by the first transaction.

– "Phantoms" One transaction reads rows from a table in which additional rows are then inserted by a second transaction. If the first transaction performs the same query again, the number of rows found may be greater.

The following table shows which phenomena can occur at the different consistency levels:

Consistency level	non-repeatable read	phantoms
2	x	x
3		x

Rule

– The SET TRANSACTION statement may only be preceded by PERMIT in the static statements of a transaction.

SHOW Display information on metadata

SHOW can be used to display information on base tables, views, cursors, columns and schema names.

```
      ⎧               ⎧SCHEMA [schema-name]⎫⎫
      ⎪TABLES FROM    ⎨VIEW view-name      ⎬⎪
      ⎪               ⎩CURSOR cursor       ⎭⎪
      ⎪                                     ⎪
      ⎪VIEWS                                ⎪
      ⎪                                     ⎪
      ⎪VIEW view-name                       ⎪
      ⎪                                     ⎪
      ⎪CURSORS                              ⎪
      ⎪                                     ⎪
SHOW  ⎨CURSOR cursor                        ⎬
      ⎪                                     ⎪
      ⎪            ⎧TABLE table       ⎫     ⎪
      ⎪            ⎪VIEW view-name    ⎪     ⎪
      ⎪ITEMS FROM  ⎨                  ⎬     ⎪
      ⎪            ⎪CURSOR cursor     ⎪     ⎪
      ⎪            ⎩ITEM column       ⎭     ⎪
      ⎪                                     ⎪
      ⎪ITEM column                          ⎪
      ⎪                                     ⎪
      ⎩SCHEMA                               ⎭
```

TABLES FROM
: The table name is displayed for all base tables of the specified schema, view or cursor.

  If *schema-name* is not specified, *schema-name* from PARAMETER DYNAMIC is assumed.

VIEWS
: The view name and "updatable (AE)/not updatable" is displayed for all views.

VIEW view-name
: The name of the view and "updatable (AE)/not updatable" is displayed for *view-name*.

CURSORS
: The cursor name and "updatable (AE)/not updatable" is displayed for all cursors.

CURSOR cursor
: The name of the cursor and "updatable (AE)/not updatable" is displayed for *cursor-name*.

ITEMS FROM
: The name and data type, with number of digits and decimal places, is displayed for all columns of the specified table, view or cursor, plus:

  – the repetition factor, for vectors or repeating groups
  – the key identification
  – the null value constraint
  – the name of the referenced table

ITEM column
: The same information is displayed for *column* as in ITEMS FROM, however only for the column specified.

  column must be specified with *prefix* and, in the case of tables, with *schema-name* (see *column*).

SCHEMA
: In UDS databases, all schema names are displayed for which access rights are in effect at this time.

STORE Save the cursor position

STORE saves a cursor position beyond transaction boundaries.

All cursors are closed at the end of a transaction.
STORE can be used to save a cursor position before the end of a transaction. The cursor can then be restored with RESTORE in a later transaction. The cursor has the same position after RESTORE as when STORE was executed.

```
STORE cursor
```

cursor	identifies a cursor.

Rules

- The cursor must be declared beforehand with DECLARE.
- The cursor position can be saved until the end of the UTM conversation or the end of the application program run, but no longer than that.
- The cursor must have been opened with the OPEN or RESTORE statement and positioned to a row.
- Entering a second STORE statement for the same cursor overwrites the cursor position stored earlier.

UPDATE... WHERE condition
Update column values in selected rows

UPDATE ... WHERE *condition* updates column values of selected rows in a base table. You can also specify this statement for an updatable view.

```
                                ⎡sql-expression⎤
UPDATE table SET {column=⎨              ⎬},...[ WHERE condition]
                                ⎣NULL          ⎦
```

table	identifies a base table or updatable view (see the metavariable *table*).
SET	assigns the columns the new values specified in *sql-expression* or the keyword NULL. The data types of the expression and the column must be compatible.
column	identifies the column to be updated (see the metavariable *column*). The column must be located in the table specified. *column* may not be qualified with *table*. A column may not be specified more than once, even as an element in a structured column. The column must not identify the primary key. The values of any columns not specified remain unchanged.
sql-expression	identifies an expression (see the metavariable *sql-expression*) whose value is assigned to the corresponding column. If a *column* of *table* occurs in *sql-expression*, the values of this column are valid before any update with UPDATE. If the column is structured, either a *variable* with the same structure or an *aggregate* must be specified for *sql-expression* (see the metavariable *value*).
NULL	assigns the null value to a column.
condition	selects the rows that are to be updated (see the metavariable *condition*). If this entry is omitted, all the rows in the base table or view specified are updated.

Rules

- If no row satisfies the condition, &SQL_CODE is set to 100.
- If an error occurs in the nth row (n>1) during execution of the statement UPDATE... WHERE condition, the transaction is rolled back by the database system. If the error occurred in the first row, only the current statement is rolled back.
- An *sql-expression* must not contain a *set-function*.

UPDATE... WHERE CURRENT OF... Update column values in the current row of the cursor

UPDATE... WHERE CURRENT OF... updates the values of the columns in the row to which the cursor is positioned.

The position of the cursor remains unchanged.

```
                             ⌈sql-expression⌉
UPDATE table SET {column={                    }},... WHERE CURRENT OF cursor
                             ⌊NULL          ⌋
```

table	identifies a base table or updatable view (see the metavariable *table*). The specified table name must match the table name specified in the FROM clause of the underlying cursor declaration.
SET	assigns the columns the new values specified in *sql-expression* or NULL. The data types of the expression and the corresponding column must be compatible.
column	identifies the column to be updated (see the metavariable *column*). The column must be located in the base table specified. *column* may not be qualified with *table*. A column may not be specified more than once, even as an element in a structured column. The column must not identify the primary key. The values of any columns not specified remain unchanged.

sql-expression
identifies an expression (see the metavariable *sql-expression*) whose value is assigned to the corresponding column. If a *column* of *table* occurs in *sql-expression*, the values of this column are valid before any update with UPDATE.

If the column is structured, either a *variable* with the same structure or an *aggregate* must be specified for *sql-expression* (see the metavariable *value*).

NULL assigns the null value to a column.

cursor identifies a cursor. The cursor must be updatable.

Rules

- The cursor must be declared beforehand with DECLARE.
- The cursor
  - must have been opened with an OPEN statement and positioned to a row with FETCH, or
  - must have been opened a RESTORE statement, in which case the cursor is pointing to the same row as when the STORE statement was executed.
- *sql-expression* must not contain a *set-function*.

column Specifying columns

For *column*, you can specify a:

- simple column
- vector
- structure
- vector with structured elements

```
column ::=

                                                           ⎧simple-column                           ⎫
  ⎧table      ⎫    ⎧structure1                ⎫            ⎪                                        ⎪
[ ⎨           ⎬.][ ⎨                          ⎬.]...       ⎨structure2                              ⎬
  ⎩correlation⎭    ⎩vector-structured1(index1)⎭            ⎪ ⎧vector[(index1[..index2])]    ⎫       ⎪
                                                           ⎪ ⎨                              ⎬       ⎪
                                                           ⎩ ⎩vector-structured2[(index1)]] ⎭       ⎭
```

table	identifies a base table or view (see the metavariable *table*). If a statement includes identical names for columns from different tables, you must specify *table* to uniquely identify these columns.
correlation	Correlation for a table. *correlation* can be declared in the FROM clause of the SELECT statement or *query-expression*. If a statement includes identical names for columns from different tables and you have declared correlations for these tables, you must also specify *correlation* to uniquely identify the columns.
structure1	identifies a structure.
vector-structured1(index1)	structured element of a vector; *index1* must be greater than 0.
simple-column	identifies a non-structured column.
structure2	identifies a structure.

vector[(index1[..index2])]

identifies a vector.
You can specify only one element of a vector (indexed vector) with *index1* or a partial vector with *index1..index2*.
The elements specified must be contained in the vector.
For *index1* or *index2*:
index1 and *index2* must be integer constants.
index1 must be greater than 0 and
index2 must be greater than *index1*.

vector-structured2[(index1)]

identifies a vector with structured elements.
You can specify only one element of the vector with *index1*. The element specified must be contained in the vector.
index1 must be greater than 0.

Rules

– *table* and *correlation* must not be specified for *column* in an ORDER BY clause.

– If you specify a *column*, *structure*, *vector* or *vector-structured* as a lower-level column, the higher-level column must be a *structure* or *vector-structured*. The individual column names must be separated by periods ".".

– If the column is an element of a vector with structured elements, you must use *vector-structured1(index1)* to specify which occurrence of the vector with structured elements the specified column belongs to.

– If the name specified is not unique within the statement, you must prefix the column with the name of the table or the correlation for the table.

– If you prefix a column name with its table name in a SELECT query, you must also specify this table name in the relevant FROM clause.

condition Specifying a condition

A condition consists of one or more logical expressions and the logical operators AND, OR and NOT.
A row is included the result table if it satisfies the specified condition (CREATE, DECLARE) or else the appropriate action is carried out (DELETE, UPDATE).

There is a distinction between join conditions and selection conditions; you use a join condition to link base tables together, and a selection condition to select rows from a table.

The following selection conditions can be formulated:

- Comparison of a column with an expression
- Comparison of a column with a value range
- Comparison of a column with a list of values
- Comparison of a column with the null value
- Comparison of a column with a pattern

Null values in conditions

If the null value occurs in a condition, the result of the condition may be either satisfied, not satisfied or unknown. Refer to the section on the *condition* syntax element in the "SQL for UDS/SQL V1.0 Language Reference Manual" [13] for a detailed description of when a condition is satisfied, not satisfied or unknown.
If the result of *WHERE condition* is unknown, the database system reacts as if the result of the condition were not satisfied.

The syntax for *condition*:

```
             ⎡join-condition [AND selection-condition]...         ⎤
             ⎢                                                    ⎥
condition := ⎨                        ⎡AND⎤                       ⎬
             ⎢selection-condition1 [  ⎨   ⎬ selection-condition2]...⎥
             ⎣                        ⎣OR ⎦                       ⎦
```

```
join-condition := column = column [AND column = column]...
```

```
                                ⎡                  ⎡= ⎤                                                ⎤
                                ⎢                  ⎢< ⎥                                                ⎥
                                ⎢                  ⎢> ⎥                                                ⎥
                                ⎢sql-expression    ⎨<>⎬ sql-expression                                 ⎥
                                ⎢                  ⎢<=⎥                                                ⎥
                                ⎢                  ⎣>=⎦                                                ⎥
                                ⎢                                                                      ⎥
                                ⎢column[ NOT] BETWEEN sql-expression1 AND sql-expression2              ⎥
                                ⎢                                                                      ⎥
selection-condition := [NOT]    ⎨column[ NOT] IN (value,...)                                           ⎬
                                ⎢                                                                      ⎥
                                ⎢column IS[ NOT] NULL                                                  ⎥
                                ⎢                                                                      ⎥
                                ⎢column[ NOT] LIKE pattern                                             ⎥
                                ⎢      [ ESCAPE escape-character]                                      ⎥
                                ⎢                                                                      ⎥
                                ⎢                          ⎡AND⎤                                       ⎥
                                ⎢(selection-condition[     ⎨   ⎬ selection-condition]...)              ⎥
                                ⎣                          ⎣OR ⎦                                       ⎦
```

The following applies to conditions combined with AND or OR, or negated with NOT:

AND	both conditions combined by AND must be satisfied for the overall condition to be satisfied.
OR	at least one of the two conditions combined by OR must be satisfied for the overall condition to be satisfied.
NOT	negation: the condition negated with NOT must be not satisfied for the overall condition to be satisfied. If the result of the negated condition is unknown, the result of the overall condition is also unknown.

join-condition

You use a join condition to link base tables together (a join). The columns you specify in a join condition are called join columns.

To create a join condition, you formulate a comparison predicate with the "=" comparison operator to compare a column from one base table with a column from a second base table.

The data types of the columns being compared must be compatible.

If one of the columns is a foreign key, the other must be the primary key of the base table to which the foreign key refers.
In this case, you join the base tables via a foreign key and its associated primary key.

You can join several base tables by logically ANDing two or more comparison predicates in a join condition. This is referred to as a multiple join.

The first comparison predicate in a join condition defines the two base tables to be joined.
Each subsequent comparison predicate specifies a new base table to be joined with one and only one base table defined in one of the preceding comparison predicates. This means that a comparison predicate must involve only one column from a base table already referenced in a previous comparison predicate.

A join of three base tables involves two comparison predicates and AND. A join of four base tables is formulated with three comparison conditions, and so on.

There is a special type of join which involves the same base table being joined with itself. In this case, both columns come from the same base table. As base tables have to have different names in the FROM clause, however, a correlation name must be declared for at least one of these base tables.

selection-condition

You use a selection condition to select rows from a table.

selection-condition1 OR selection-condition2

All the columns must be in a single table. If the FROM clause refers to a join view, only columns that were assigned in the view definition for the same base table may be specified.

selection-condition1 AND selection-condition2 [AND...]

All the columns within *selection-condition1*, *selection-condition2* etc. must come from the same table. If the FROM clause refers to a join view, only columns that were assigned in the view definition for the same base table may be specified in *selection-condition1*, *selection-condition2* etc.

You can combine selection conditions using the logical operators AND, OR and NOT, and insert parentheses to group conditions together.

Rule

If you combine the logical operators AND, OR and NOT, the usual precedence rules apply:
NOT before AND before OR.
If you want to change this order, you must insert parentheses at the appropriate positions. Operators within parentheses take precedence.

The following pages describe how to use *selection-condition* in the individual functions and explain the entries.

Not described in *condition* are the entries for:

– *sql-expression*	see the metavariable *sql-expression*
– *column*	see the metavariable *column*

Comparing expressions using comparison operators

You can compare the values of two expressions using comparison operators.

```
                     ⎧=  ⎫
                     ⎪<  ⎪
                     ⎪>  ⎪
sql-expression1      ⎨   ⎬ sql-expression2
                     ⎪<= ⎪
                     ⎪>= ⎪
                     ⎩<> ⎭
```

Comparison operator	Meaning
=	equal to
<	smaller than
>	greater than
<=	smaller than or equal to
>=	greater than or equal to
<>	not equal to

The condition is satisfied if the comparison is true.
The result of the condition is unknown if at least one expression has the null value.

Rules

– At least one expression must be a column. The second expression may be either a column or a value.

– Only columns from the same base table may be used to formulate a comparison value within a selection condition.

– If an expression identifies a foreign key or primary key, the comparison operator must be "=" or "<>".

– The expressions must be either both numeric or both character-string, or have the same structure.

– If an expression identifies a structured column, the comparison operator must be "=" or "<>", and the second expression must have the same structure.

– Each character in a character string condition is compared individually. If the expressions are of unequal length, the shorter expression is padded with blanks.

Comparing a column with a value range

This condition determines whether the column value lies within the specified value range or not.

```
column [NOT] BETWEEN sql-expression1 AND sql-expression2
```

BETWEEN ... AND
: The result of the condition is the same as for the condition
sql-expression1 <= column AND column <= sql-expression2

The condition is satisfied if the value of *column* lies within the value range.

NOT BETWEEN ... AND
: The result of the condition is the same as for the condition
column < sql-expression1 OR column > sql-expression2

The condition is satisfied if the value of *column* does not lie within the value range.

Rules

– The column and the expressions must not be structured and must be either all character string or all numeric.

– Each expression identifies a *value*.

Comparing a column with a list of values

This condition compares a column with a list of specified values.

```
column [NOT] IN (value,...)
```

IN

- The condition is satisfied if the value of *column* is not the null value and matches at least one of the values listed after IN.
- The condition is not satisfied if neither the value of *column* nor one of the values listed after IN is the null value and the value of *column* does not match any of the values listed after IN.

Otherwise, the result of the condition is unknown.

NOT IN

- The condition is satisfied if neither the value of *column* nor any of the values listed after IN is the null value and the value of *column* does not match any of the values listed after IN.
- The condition is not satisfied if the value of *column* is not the null value and matches at least one of the values listed after IN.

Otherwise, the result of the condition is unknown.

Rules

- The column and the values must be either all numeric, all character string or have the same structure.
- You must specify a list of values after IN, i.e. more than one expression.

Comparing a column with the null value

```
column IS [NOT] NULL
```

IS NULL
: The condition is satisfied if *column* contains the null value. Otherwise the condition is not satisfied.

IS NOT NULL
: The condition is satisfied if *column* does not contain the null value. Otherwise the condition is not satisfied.

Rule

– The column must refer to a foreign key.

Comparing a column with a pattern

This condition compares the value of a column with a specified pattern.

```
column [NOT] LIKE pattern
       [ESCAPE escape-character]
```

LIKE	The condition is satisfied if the value of *column* matches *pattern*.
NOT LIKE	The condition is satisfied if the value of *column* does not match *pattern*.

The result of the condition is unknown if *column* or *pattern* contains the null value.

pattern You can specify a character constant *const* or a *variable* for *pattern*.
The pattern may consist of the following characters:

Pattern contains	The condition is satisfied if the *column* contains at the same place ...
character	exactly the same character
_ (underscore)	any character
%	any character string or nothing
escape-character%	%
escape-character_	_
escape-character-escape-character	escape-character

escape-character

declares a character with which you can disable the characters "%", "_" or the *escape-character* as pattern characters. In this way, you can determine whether the column contains these characters.
You can specify a character constant *const* or *variable* with a length of 1 for *escape-character*.

Rules

- The column must be alphanumeric.
- Unless preceded by an *escape-character*, "%" may be specified only at the end of *pattern*.
- *escape-character* must immediately precede "%", "_" or a second *escape-character*.
- *escape-character* and the following character are handled as a single character in *pattern*.
- The length of a pattern that contains no "%" special characters and no *escape-character* must match the defined length of the corresponding column.

query-expression
Specifying SELECT within SQL statements

A *query-expression* selects rows and columns from base tables or views and allows you to join rows from two or more base tables.

The result is again a table, the result table.
The result table contains the columns specified in the select list and which are part of the rows selected by the WHERE clause.

If the view or the cursor is used in further SQL statements, the result table is the basis for retrieving data from the database and for updates.

Updatable query expression

The *query-expression*, and hence the result table, can be updatable or not updatable. A *query-expression* is updatable if

– only columns are specified in the *select-list* and each column is referenced only once. A column must not be specified once as an element of a structured column and then specified again directly under its own name.
For example, you are not allowed to specify the structured column ADDRESS and then also specify CITY as an element of ADDRESS.
Partial vectors must not overlap.

– only one table is specified in the FROM clause. The table must be a base table or an updatable view.

A view or a cursor is updatable if *cursor-specification* is in the form *query-expression* and *query-expression* is updatable.

```
query-expression::=  SELECT[ ALL] select-list

                            FROM table-specification,...

                            [ WHERE condition]
```

The clauses must be specified in the order shown above.

[ALL] (default value)
returns all the rows selected with WHERE *condition*. Duplicate rows are also selected. The ALL entry is syntactically permissible, but has no effect on the result.

select-list Selecting columns

Using the *select-list*, you specify the columns you want to include in the result table.

```
                ⎡*                 ⎤
select-list := ⎨                    ⎬
                ⎣sql-expression,...⎦
```

* The result table contains all the columns of the table(s) specified in the FROM clause. The order of the columns is determined

- by the order of the tables in the FROM clause, and
- within a table, by the order defined in the schema.

sql-expression
The result table contains the columns specified with *sql-expression*,... in the order of their specification (see the metavariable *sql-expression*).

Rules

- If an *sql-expression* contains a *set-function*, each column in the *select-list* must be part of a set function.
- All column names must be unique. If you want join base tables that have columns with the same name, you have to prefix the columns with the name of the base tables involved or their correlations in order to assure unique identification.
- The characteristics of result table columns (data type, length, precision, scale factor) are either adopted from the underlying table or result from the expression specified.

FROM Selecting tables

You use the FROM clause to specify the base tables or view from which data is to be selected for the result table.

```
FROM table-specification,...
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
table-specification::= table [correlation]
```

table	specifies a base table or view whose columns are used in the query expression (see the metavariable *table*). If you specify more than one table, they must be base tables. Only base tables can be joined. You then use the WHERE clause to select rows from this table. If two or more tables are specified in the FROM clause (in which case the WHERE clause must be specified), the result table consists of the columns specified in *select-list* and result rows from all possible combinations of the rows of the two tables (a "Cartesian product"). Since a join condition must be specified in the WHERE clause whenever more than one table is listed in the FROM clause, the result table specified by the FROM clause does not, as a rule, contain all the rows of the result table.
correlation	*correlation* assigns a new name to *table*. The new table name can then be used within the SELECT query. The *correlation* must conform with the naming conventions.

Rules

– A FROM clause may refer only to base tables or a view.

– If identical table names occur in the FROM clause, correlations must be specified to distinguish the table names.

– All correlations within the FROM clause must be unique and must not be the same as any table name for which no *correlation* has been declared.

WHERE Selecting rows

In the WHERE clause, you specify the conditions to be applied when selecting rows for the result table. The result table contains only rows which fulfill the specified conditions.

```
WHERE condition
```

WHERE condition

selects rows which satisfy the specified condition from the result table defined by the FROM clause (see the metavariable *condition*).

You must include a WHERE clause if you have specified more than one table in the FROM clause. In this case, you must specify a join condition.

If you specify only one table in the FROM clause and no WHERE clause, the result table will contain all the rows of the table specified.

Each *column* specified in *condition* must occur in a *table* referenced in the FROM clause.

set-function Specifying set functions

A set function is used to calculate a value from a set of rows.

```
                  ⎡COUNT(*)            ⎤
                  ⎢                    ⎥
                  ⎢⎡SUM⎤               ⎥
set-function:=   {⎢AVG⎥               }
                  ⎢{   }([ ALL] column)⎥
                  ⎢⎢MAX⎥               ⎥
                  ⎣⎣MIN⎦               ⎦
```

COUNT(*)
: returns the number of rows, including duplicates.

SUM([ALL] column)
: adds the values of *column*. *column* must be numeric.

AVG([ALL] column)
: calculates the average of the values of *column*.
column must be numeric.

MAX([ALL] column)
: returns the greatest value of *column*.
column must be numberic or character string.

MIN([ALL] column)
: returns the lowest value of *column*.
column must be numeric or character string.

The results of the set functions have the following data types:

Set function	Data type of result
`COUNT(*)`	`DECIMAL data type with a precision of 15, no decimal places.`
`MIN and MAX`	`Same data type as the` *`column`* `specified.`
`SUM`	`DECIMAL data type with a precision of 15.` `The number of decimal places corresponds to the number of decimal places in the specified` *`column`*`.`
`AVG`	`DECIMAL data type with a precision of 15.` `The number of decimal places is equal to 15 minus the places before the decimal point in the specified` *`column`*`.`

Rules

- Set functions are only permitted within the *select-list* of a SELECT statement or *query-expression*.
- The column must not be structured.
- Null values in the column are not taken into account in the calculation of SUM, AVG, MAX or MIN.
- If the result of WHERE *condition* in the *query-expression* or the SELECT statement returns no rows, the result of SUM, AVG, MIN or MAX is the value, and COUNT(*) has the result "0".

sql-expression Specifying expressions

Expressions consist of columns, constant, variables or set functions. You can also combine several expressions to form one expression using arithmetic operators (+,- ,*,/). This is called an arithmetic expression.

```
                     ⎡column                               ⎤
                     ⎢value                                ⎥
                     ⎢set-function                         ⎥
                     ⎢                                     ⎥
                     ⎢⎡+⎤                                  ⎥
                     ⎢{ }sql-expression                    ⎥
                     ⎢⎣-⎦                                  ⎥
sql-expression :=   {                                       }
                     ⎢               ⎡+⎤                   ⎥
                     ⎢               ⎢-⎥                   ⎥
                     ⎢sql-expression{   }sql-expression    ⎥
                     ⎢               ⎢*⎥                   ⎥
                     ⎢               ⎣/⎦                   ⎥
                     ⎢                                     ⎥
                     ⎣(sql-expression)                     ⎦
```

column identifies a column name (see the metavariable *column*). The column must be contained in a table specified in the FROM clause of the SELECT query or in the DELETE, INSERT or UPDATE statements.

value identifies a value (see the metavariable *value*).

set-function

identifies a set function (see the metavariable *set-function*). *sql-expression* is the value returned by this function. *set-function* may only occur in the *select-list* of a SELECT statement or *query-expression*.

-sql-expression, +sql-expression

"-" causes a change in sign. "+" leaves the value of *sql-expression* unchanged. *sql-expression* must be numeric. *sql-expression* must not begin with "+" or "-".

The following can be specified for values or expressions:

```
     ⎡column          ⎤
⎡+⎤  ⎢value           ⎥
⎣-⎦ {⎢set-function    ⎥}
     ⎣(sql-expression)⎦
```

```
              ⎡+⎤
              ⎢-⎥
sql-expression⎨ ⎬sql-expression
              ⎢*⎥
              ⎣/⎦
```

identifies the arithmetic operations addition, subtraction, multiplication and division. Both operands must be numeric.

(sql-expression)

Using parentheses, you can group parts of expressions together to form a unit in order to change the order in which arithmetic operations are performed. The parentheses must be set according to the precedence rules used in algebra.

The result of an arithmetic expression has the numeric data type.
The precision (number of places to the left and right of the decimal point) and the scale factor (number of places to the right of the decimal point) of the result of the arithmetic expression depend on

– the precision and scale factor of the arithmetic operands, and

– the arithmetic operation involved.

The following table informs you how to determine the precision and scale factor in each case:

Operation	Result
Addition x+y Subtraction x-y	The precision P is equal to the sum of the maximum number of digits to the left of the decimal point and the maximum number to the right plus one; maximum 15. $P = MIN (15, MAX(left_x,left_y) + MAX(S_x,St_y) + 1)$ 1) The scale factor S is equal to the number of digits to the right of the decimal point in the operand that has the maximum number of digits to the right. $S = MAX(S_x,S_y)$ 1)
Multiplication x*y	The precision P is equal to the sum of the precisions of the factors; maximum 15. $P = MIN (15, P_x+P_y))$ 1) The scale factor is equal to the sum of the scale factors of the factors; maximum 15. $S = MIN(15, S_x+S_y)$ 1)
Division x/y	The precision P is equal to 15. P = 15 The scale factor S is calculated using the following formula: $S = MAX(15-P_x+S_x-S_y, 0)$ 1)

1) $left_x$	–	Number of digits to the left of the decimal point in the first arithmetic operand
$left_y$	–	Number of digits to the left of the decimal point in the second arithmetic operand
S_x	–	Scale factor (number of digits to the right of the decimal point) in the first operand
S_y	–	Scale factor (number of digits to the right of the decimal point) in the second arithmetic operand
P_x	–	Precision of the first arithmetic operand
P_y	–	Precision of the second arithmetic operand
Where:		$P_x = left_x + S_x$ $P_y = left_y + S_y$

If the result of the arithmetic operation after removal of digits to the right of the decimal point does not fit into the result column described in the table above, a value overflow occurs (SQLCODE -340). Division by 0 also results in SQLCODE -340.
If the scale factor of the result is greater than 15, digits to the right of the decimal point are lost.

Rules

- If a null value occurs in an arithmetic expression, the whole expression has the null value.
- The usual rules of algebra apply to the use of arithmetic operators.
- If an *sql-expression* contains a *set-function*, each *column* in the *select-list* must be part of a *set-function*.

sql-literal Specifying literals

Literals are character strings that represent a constant value.

A distinction is made between character and numeric literals. Numeric literals can be assigned only to numeric columns, and character literals only to character columns. DRIVE internally converts hexadecimal literals to character literals. In the "SQL for UDS/SQL V1.0 Language Reference Manual" [13], the term literal has been used for sql-literal.

```
                ⎡char_literal       ⎤
sql-literal ::= ⎨numeric_literal    ⎬
                ⎣hexadecimal_literal⎦
```

Character literals

```
char_literal ::= ’character...’ [(n)]
```

character	Any ASCII character is permitted. *char_literal* can contain up to 256 characters. If the character ’ (apostrophe) is used within *char_literal*, it must be entered twice: ’ ’. The double apostrophe is regarded as a single character.
n	Repetition factor. The string preceding *n* is repeated *n* times.

Numeric literals

```
                       ⎧+⎫ ⎧integer          ⎫
numeric_literal ::= [⎨ ⎬]⎨                 ⎬
                       ⎩-⎭ ⎩fixed-point-number⎭
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                          ⎧integer[.integer]⎫
fixed-point-number ::= ⎨integer.         ⎬
                          ⎩.integer         ⎭
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
integer ::= digit...
```

digit	Digit from 0 to 9. A fixed point number can have up to 15 digits, of which up to 14 can appear after the decimal point.

Hexadecimal literals

```
hexadecimnal_literal := X’hexadecimal_digit ...’  [(n)]
```

hexadecimal_digit

Hexadecimal digit, i.e. 0, 1, ...9, A, B, ... F.
Each hexadecimal digit represents the bits of a half byte. A full byte must always be supplied with a value, i.e. ’hexadecimal_digit ...’ must always contain an even number of hexadecimal digits.
A maximum of 512 hexadecimal digits (i.e. 256 bytes) can be specified.

table Specifying tables

Tables specified in SQL statements may be base tables or views. Tables can be qualified with the name *schema*.

```
                    ⎡base-table⎤
table::= [schema.] ⎨          ⎬
                    ⎣view      ⎦
```

schema

identifies the relational schema where the base table is located. If this specification is missing, the schema used in OPTION SCHEMA or PARAMETER DYNAMIC SCHEMA is used. If neither OPTION SCHEMA nor PARAMETER DYNAMIC SCHEMA has been specified, the table name or view name must be unique.
If the fact that *schema* has not been specified means that a base table or view cannot be identified uniquely, DRIVE issues an error message. If a transaction accesses base tables located in different schemas, the schemas must be defined in different databases.

view

identifies a view. A view must be declared in the program text before any statements that reference it.

value Specifying values

A value can be declared by means of a variable, literal or aggregate.
An aggregate can be specified for *value* only if it is to be assigned to, or compared with, a structured column.

```
              ⎡sql-literal⎤
value ::= ⎨variable   ⎬
              ⎣aggregate  ⎦
```

```
                  ⎡value⎤
aggregate ::= <⎨     ⎬,...>
                  ⎣NULL ⎦
```

sql-literal
: identifies a numeric or character literal.

variable
: identifies a variable.

aggregate
: identifies a compound value for representing values in a structured column. A value must be specified for each lower-level column of the structured column.

value
: identifies another literal, variable or *aggregate*.

NULL
: specifies the null value.

variable Specifying variables

You must use the following syntax when specifying data elements as variables within SQL statements.

```
variable::=  &data-name
```

variable

is used in an SQL statement to include values from the database (output variable), store values in the database or make available values required in calculations or conditions (input variable).
Thus, output variables include all the variables that occur in the following places:

- in the RETURN INTO clause of the INSERT statement,
- in the INTO clause of the SELECT statement or
- in the FETCH statement after INTO.

All other variables are input variables.

The data types of columns and any variables associated with them must be compatible with one another. Compatible combinations are: numeric columns and numeric variables or character columns and character variables. Structured variables and columns are compatible with one another if both have the same number of components and the pairs of corresponding components are compatible with one another.

data-name

Name of the variable.
DRIVE requires that variables are specified in the form "&data-name".
data-name can be up to 31 characters in length. Refer to the metavariable *variable* for a detailed description of the syntax. DRIVE also provides you with the option of using DECLARE VARIABLE ... LIKE to create variables whose structure and name correspond to those of the table and cursor.

References

[1] **DRIVE/WINDOWS** (SINIX)
Software Production Environment (SPE)
User Guide

Target group
Application programmers
Contents
The functions available in the software production environment (desktop) and in expert mode. Setting up DRIVE/WINDOWS, including remote access to BS2000 databases and generating applications for BS2000.

DRIVE/WINDOWS V1.1 (BS2000)
Programming System
User Guide

Target group
Application programmers
Contents
- Introduction to the programming system DRIVE/WINDOWS
- Explanation of the functions available in interactive mode
- Installation
- DRIVE/WINDOWS generation and administration

[2] **DRIVE/WINDOWS** (SINIX)
Programming Language
Reference Manual

Target group
Application programmers
Contents
The creation of programs, including graphical and alpha screen forms, as well as list forms using DRIVE and the report generator.

DRIVE/WINDOWS V1.1 (BS2000)
Programming Language

Target group
Application programmers
Contents
– Program creation
– Transaction concept
– Distributed transaction processing
– Screen and list forms
– Reports
– Examples

[3] **DRIVE/WINDOWS** (SINIX)
System Directory
Reference Manual

Target group
Application programmers
Contents
The syntax and scope of functions of all DRIVE statements, as well as all DRIVE messages and keywords.

DRIVE V6.0A (BS2000)
System Directory
User Guide

Target group
Applications programmers
Contents
– Syntax and range of functions of all DRIVE statements
– DRIVE messages and keywords

[4] **DRIVE/WINDOWS** (SINIX)
Directory of DRIVE SQL Statements for INFORMIX
Reference Manual

Target group
Application programmers
Contents
A concise description of the syntax and scope of functions of all the DRIVE SQL statements for INFORMIX.

[5] **DRIVE/WINDOWS** (SINIX)
Directory of DRIVE SQL Statements for ISO/SQL
Reference Manual

Target group
Application programmers

Contents
A concise description of the syntax and scope of functions of all the DRIVE SQL statements for ISO/SQL.

[6] **DRIVE/WINDOWS** (SINIX)
Directory of DRIVE SQL Statements for SESAM
Reference Manual

Target group
Application programmers
Contents
A concise description of the syntax and scope of functions of all the DRIVE SQL statements for SESAM.

[7] **DRIVE/WINDOWS** (SINIX)
Directory of DRIVE SQL Statements for UDS
Reference Manual

Target group
Application programmers
Contents
A concise description of the syntax and scope of functions of all the DRIVE SQL statements for UDS.

[8] **DRIVE/WINDOWS V1.1**
(SINIX)
Supplement
User Guide

Target group
Application programmers
Contents
The manual contains the functional changes included in DRIVE/WINDOWS (SINIX) V1.1. If this supplement is to be used, the manuals of version 1.0 are also required.

[9] **DRIVE/WINDOWS V1.1** (BS2000)
Programming Language

Target group
Application programmers

Contents
- Program creation
- Transaction concept
- Distributed transaction processing
- Screen and list forms
- Reports
- Examples

[10] **DRIVE V6.0A** (BS2000)
System Directory
User Guide

Target group
Applications programmers
Contents
- Syntax and range of functions of all DRIVE statements
- DRIVE messages and keywords

[11] System Interfaces for Applications **SQL for ISO/SQL** (BS2000)
Portable SQL Applications for BS2000 and SINIX Language Reference Manual

Target group
Users wanting to access SESAM or UDS databases using SQL or DRIVE
Contents
This manual describes the language elements of the product ISO/SQL V1.0. It also enables the creation of portable SQL applications in BS2000 and SINIX, as the common language elements of ISO/SQL, INFORMIX and the SQL Standard are emphasized.

[12] **SQL for SESAM/SQL**
Language Reference Manual

Target group
Programmers who want to access SESAM databases using SQL statements.
Contents
SQL statements available for accessing SESAM databases.

[13] **SQL for UDS/SQL**
Language Reference Manual

Target group
Programmers who want to access UDS databases using SQL statements.
Contents
SQL statements available for accessing UDS databases.

[14] **INFORMIX** (SINIX)
SQL
Language Reference Manual

Target group
INFORMIX users
Contents
Complete description of the INFORMIX SQL database query language for all INFORMIX products which provide an SQL interface.
Deviations from and extensions of the ANSI standard are also described.

[15] **INFORMIX**
Supplement

Target group
INFORMIX users.
Contents
The supplement for INFORMIX V4.1 contains the changes in the functions of the INFORMIX products ’SQL Language Reference Guide’, ’SQL Reference Guide’, ’ESQL/COBOL’, ’ESQL/C’, ’C-ISAM’ and ’Interactive Debugger’. The version 4.0 manuals are required.

[16] **INFORMIX Error Messages** (SINIX)
Reference Manual

Target group
INFORMIX users
Contents
This manual contains the INFORMIX error message texts together with the corresponding corrective action to be taken.

[17] **SESAM/SQL** (BS2000)
Creation and Maintenance
User’s Guide

Target group
Database administrators
Contents
– Creation and maintenance of SESAM databases using the database administration monitor SESASB
– Shadow database operation

[18] **Dialog Builder V2.0**

[19] **OSF/Motif**
Programmer’s Reference
Release 1.2

Target group
- Application designers
- Widget designers

Contents
Description of all the commands, functions and file formats of the OSF/Motif widget set

[20] **SINIX/windows User Environment**
Guide for Experts and System Administrators

Target group
SINIX/windows experts and system administrators
Contents
This manual discusses the concepts underlying the product and explains how to configure the user interface. It presents the primary clients for display and window management and for managing desktools and files.

[21] **SINIX/windows User Environment**
Clients Reference Manual

Target group
SINIX/windows experts and system administrators
Contents
This manual provides a comprehensive survey of the clients: invocation, options and resources governing client appearance and behavior. It explains the order and precedence for interpretation of resource definitions.

[22] **X Window System**
Xlib Reference Manual

Target group
- Application designers
- Widget designers

Contents
Full description of the C interface to Xlib.

[23] **FORMANT** (SINIX)
Reference Manual

Target group
- C programmers
- COBOL programmers
- Application designers

Contents
Formant is a mask control program for all SINIX systems. The manual contains:
- Introduction to FORMANT
- Description of FORMANTGEN
- Description of user interface
- Program interfaces in C and COBOL
- Programming examples

[24] **FHS** (TRANSDATA)
User Guide

Target group
Programmers
Contents
Program interfaces of FHS for TIAM, DCAM and UTM applications. Generation, application and management of formats.

[25] **IFG for FHS** (TRANSDATA)
User Guide

Target group
Terminal users, application engineers and programmers
Contents
The Interactive Format Generator (IFG) is a system that permits simple, user-friendly generation and management of formats at a terminal. In conjunction with FHS, these formats can be used on the host computer. This user guide describes how formats are generated, modified and managed, plus also the new functions of IFG.

[26] **UTM**(SINIX)
Formatting System

Target group
UTM(SINIX) users who wish to use formats, C programmers and COBOL programmers
Contents
How to use the FORMANT format handler in UTM(SINIX) program units, create formats, convert formats from BS2000 to/from SINIX.

[27] **UTM** (SINIX)
Generating and Administering Applications
User Guide

Target group
- System administrators
- UTM administrators

Contents
- Creation, generation and operation of UTM applications under SINIX
- Working with UTM messages and error codes

Applications
SINIX transaction processing

[28] **UTM** (SINIX)
Planning and Design
User Guide

Target group
- DP managers
- Application planners
- Programmers

Contents
- Introduction to UTM(SINIX); description of the program memory and interface concept, handling of data, files and databases.
- Notes on the design, optimization and performance of UTM applications under SINIX, as well as details of data protection.

Applications
SINIX transaction processing

[29] **UPIC**
Client-Server Communication with UTM
User Guide

Target group
Dp managers, application planners Programmers of UPIC programs
Contents
UPIC permits program-to-program communication between a UPIC application and a UTM application. This works both locally with a UTM (SINIX) application on the same computer and remotely with UTM applications on other SINIX or BS2000 computers. UPIC enables you to link modern presentation systems such as Motif and X Window System to UTM(SINIX) and UTM(BS2000). The manual describes how you program a UTM application in C.

[30] **ERMS** (SINIX)
Command Interface

Target group
- End users
- Database administrators

Contents
- ERMS concepts for the end user
- Description and explanation of the end-user commands

[31] **RADAR V1.0**

[32] **QUERY** (BS2000)
Reference Manual

Target group
- End users
- QUERY administrator

Contents
- Retrieving information from SESAM databases
- Printing out search results in reports
- Administering users, access authorizations and tables

Applications
Non-dp department

[33] **TOM-REF** (BS2000)
Data-Dictionary-System
Reference Manual

[34] **C** (SINIX)
Programmer's Reference Manual
Reference Manual

Target group
C programmers working under SINIX V5.41 with C-DS V1.0.
Contents
Description of the commands for program development, library functions and system calls, and a description of a number of header files and C-specific file formats.

[35] **C** (SINIX)
Guide to Tools for Programming in C
User Guide

Target group
C programmers working under SINIX V5.41 with C-DS V1.0.
Contents
The manual describes the C compilation system (preprocessor, link editor, header files, libraries), and utilities for developing, managing, maintaining and generating C programs.

[36] SINIX V5.41
MX300 Installation Guide

Target group
Service engineers and system administrators
Contents
Comprehensive guide to installing the SINIX V5.41 operating system. Descriptions of starting up a preinstalled MX300, of preparatory work to be performed before installing SINIX V5.41, and of software package installation and removal.

[37] MX500 (SINIX V5.40)
MX300 (SINIX V5.41)
System Administrator’s Reference Manual

Target group
System administrators
Contents
Describes commands and application programs for system maintenance, file formats and special system administration files and provides notes on diagnostics

[38] **Commands** (SINIX V5.41)
Part 1, A - K
Reference Manual

Target group
SINIX shell users
Contents
Alphabetically arranged description of the SINIX command set

[39] **Commands** (SINIX V5.41)
Part 2, L - Z
Reference Manual

Target group
SINIX shell users
Contents
Alphabetically arranged description of the SINIX command set

[40] SINIX V5.41
User’s Guide

Target group
Users
Contents
Description of the principal features of the SINIX operating system. Among the topics it covers are basics for SINIX system users, the file system, process control and the shell.

[41] case/4/0 Methodenhandbuch
microTOOL GmbH Berlin

[42] case/4/0 Referenzhandbuch
microTOOL GmbH Berlin

[43] case/4/0 Bedienerhandbuch
microTOOL GmbH Berlin

[44] **Widget Set**
Programmer’s Reference

Target group
– Application designers
– Widget designers

Contents
Descriptions of all the commands, functions and file formats for the XmSniBrowser, XmSniFormat, XmSniHelp, XmSniTable and XmSniTree widgets in the Siemens Nixdorf extension to the OSF/Motif widget set.

[45] **SINIX SPOOL**
Operation - Administration - Programming
User Guide

Target group
Users, administrators and programmers of the SINIX SPOOL system

Contents
– Description of commands
– Administration functions
– C interface to the SINIX SPOOL system

[46] **Style Guide**
Guidelines on the Design of User Interfaces
User’s Guide

Target group
Developers of application programs

Contents
The Style Guide contains rules and recommendations for the development of uniform user interfaces. It describes their structure and contents, and how they are used.

[47] SINIX V5.40 (MX500)
SINIX V5.41 (MX300)
System Administrator’s Guide

Target group
System administrators

Contents
– Introduction to system administration on SINIX systems
– Instructions for SINIX system configuration and maintenance

[48] **Commands** (SINIX V5.41)
Part 3, Reference Section
Reference Manual

Target group
SINIX shell users
Contents
Tables and reference section for commands described in Parts 1 and 2 -
Table of contents – Command overview
– Regular expressions
– Bourne shell metacharacters
– Data media special files
– SINIX V5.23 and V5.40 SPOOL system files
– ISO 646 character set
– References and index

[49] **File Transfer in SINIX**
FT-SINIX (SINIX) V5.0A
FTOS-SINIX (SINIX) V2.0A
User Guide

Target group
This manual is intended for SINIX users who want to work with FT/FTOS-SINIX and for SINIX system administrators.
Contents
The manual describes the functions of FT-SINIX/FTOS-SINIX. FT-SINIX is used for transferring files and for file management on the basis of the FTNEA protocols. The add-on product FTOS-SINIX permits the use of functions based on the FTAM protocol.

Ordering manuals

The manuals listed above and the corresponding order numbers can be found in the Siemens Nixdorf *List of Publications*. New publications are described in the *Druckschriften-Neuerscheinungen (New Publications)*.

You can arrange to have both of these sent to you regularly by having your name placed on the appropriate mailing list. Please apply to your local office, where you can also order the manuals.

Index

Contents

DRIVE/WINDOWS V1.1 (BS2000/SINIX)

Directory of DRIVE SQL Statements for UDS Reference Manual

Target group
Application programmers
Contents
A concise description of the syntax and scope of functions of all the DRIVE SQL statements for UDS.

Edition: **December 1993**

File: **DRV_UDS.PDF**