ISO Working Draft

SQL Call-Level Interface (CLI)

«Addendum 1 to SQL-92»

January 1993

Working Draft

ISO CLI Addendum Working Draft—January 6, 1993

Working Draft

Digital Equipment Corporation
Maynard, Massachusetts

Contents

Page

Foreword v

Introduction vii

1 Scope 1

2 Normative references 1

3 Definitions, notations, and conventions 1

4 Concepts 1

5 Call Level Interface specifications 1

5.1 Introduction 2

5.2 <CLI routine> 4

5.3 <CLI routine> invocation 8

5.4 CLI common elements 11

5.4.1 Implicit set connection 11

5.4.2 Implicit cursor 11

5.4.3 Implicit using clause 12

5.4.4 Tables associated with CLI 15

5.4.5 Return codes and status records 16

5.4.6 CLI-specific status codes 17

5.4.7 Miscellaneous characteristics 18

5.4.7.1 Handles 18

5.4.7.2 Null terminated strings 18

5.4.7.3 Null pointers 19

5.5 CLI routines 20

5.5.1 AllocConnect 20

5.5.2 AllocEnv 21

5.5.3 AllocStmt 21

5.5.4 BindCol 22

5.5.5 BindParam 23

5.5.6 Cancel 25

5.5.7 ColAttribute 26

5.5.8 Connect 27

5.5.9 DescribeCol 30

5.5.10 Disconnect 31

5.5.11 Error 32

5.5.12 ExecDirect 33

5.5.13 Execute 35

5.5.14 Fetch 36

5.5.15 FreeConnect 37

5.5.16 FreeEnv 37

5.5.17 FreeStmt 38

Working Draft

5.5.18 GetCol . . . 39
5.5.19 GetCursorName . . . 41
5.5.20 NumResultCols . . . 42
5.5.21 Prepare . . . 43
5.5.22 RowCount . . . 44
5.5.23 SetCursorName . . . 45
5.5.24 SetParamValue . . . 46
5.5.25 Transact . . . 48

Annex A Typical header files . . . 51
A.1 C Header File SQLCLI.H . . . 51
A.2 COBOL Header File SQLCLI.CBH . . . 53

Annex B Sample C programs . . . 55
B.1 Create Table, Insert, Select . . . 55
B.2 Interactive Query . . . 57

Index

Working Draft

TABLES

Table		Page
1	Abbreviated CLI generic names	15
2	Codes used for SQL data types in CLI	16
3	Codes used for buffer data types	16
4	SQLSTATE class and subclass values for CLI-specific conditions	17

Working Draft

Foreword

To be supplied.

Working Draft

Introduction

The organization of this International Standard is as follows:

1) To be supplied.

Working Draft

Information Technology — Database Languages — SQL Addendum 1 — SQL Call-Level Interface (SQL CLI)

1 Scope

To be supplied

2 Normative references

The following standards contain provisions that, through reference in this text, constitute provisions of this National Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this National Standard are encouraged to investigate the possibility of applying the most recent editions of the standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

— ISO/IEC 9075:1992, *Information Technology — Database Languages — SQL.*

More to be supplied as required

3 Definitions, notations, and conventions

To be supplied

4 Concepts

To be supplied

5 Call Level Interface specifications

Working Draft

5.1 Introduction

The Call Level Interface (CLI) is an alternative mechanism for executing SQL statements. It consists of a number of routines that:

1) Allocate and deallocate resources.

2) Control connections to SQL-servers.

3) Execute SQL statements using similar mechanisms to dynamic SQL.

4) Obtain diagnostic information.

5) Control transactions.

The AllocEnv, AllocConnect, and AllocStmt routines allocate the resources to manage an SQL-environment, an SQL-connection, and SQL-statement processing, respectively. An SQL-connection is allocated within the context of an allocated SQL-environment and an SQL-statement is allocated within the context of an allocated SQL-connection. The FreeEnv, FreeConnect, and FreeStmt routines deallocate the corresponding resources.

The Connect routine establishes an SQL-connection. The Disconnect routine terminates an established SQL-connection. Switching between established SQL-connections occurs automatically whenever the application switches processing to a dormant SQL-connection.

The ExecDirect routine is used for a one-time execution of an SQL-statement. The Prepare routine is used to prepare an SQL-statement for subsequent execution using the Execute routine. A description of the resultant columns of a <dynamic select statement> or <dynamic single row select statement> is obtained either one attribute at a time by using the ColAttribute routine or all attributes at a time, except for the column count, by using the DescribeCol routine. The column count can be independently obtained by using the NumResultCols routine.

Executed SQL-statements can contain dynamic parameters. For each dynamic parameter, either a value source must be established by using the BindParam routine or a value must be set by using the SetParamValue routine. When a value source is established, the value is not retrieved until the SQL-statement is executed. When a value is set, that value is used when the SQL-statement is executed. When a <dynamic select statement> or <dynamic single row select statement> is executed, a cursor is implicitly declared and opened. The cursor name can be preset by using the SetCursorName routine. If a cursor name is not preset, an implementation-dependent cursor name is generated. The current cursor name can be retrieved by using the GetCursorName routine. The Fetch routine is used to position an open cursor on the next row and retrieve the values of certain columns of that row. The column values retrieved by the Fetch routine are for those columns for which a corresponding result target has previously been established by use of the BindCol routine. Values for columns without a corresponding result target can be individually retrieved by using the GetCol routine. The GetCol routine also enables the values of character string columns to be retrieved piece by piece. The current row of a cursor can be deleted by the execution of a <preparable dynamic delete statement: positioned> for that cursor. The current row of a cursor can be updated by the execution of a <preparable dynamic update statement: positioned> for that cursor. An option of the FreeStmt routine enables a cursor to be closed.

The Error routine is used to obtain diagnostic information about the execution of the most recent routine operating on a particular resource. As it obtains information about only a single condition, it needs to be used repeatedly to retrieve all the available diagnostic information. Information on the number of rows affected by the last executed SQL-statement is obtained by using the RowCount routine.

Working Draft

An SQL-transaction is terminated by using the Transact routine.

Note: Neither a <commit statement> nor a <rollback statement> may be executed using the ExecDirect or Execute routines.

The Cancel routine is used to cancel the execution of a concurrently executing CLI routine.

Working Draft

5.2 <CLI routine>

Function

Describe a generic CLI routine.

Format

```
<CLI routine> ::=
      <CLI routine name>
      <CLI parameter list>
      [ <CLI returns clause> ]

<CLI routine name> ::= <CLI name prefix><CLI generic name>

<CLI name prefix> ::=
        <CLI by-reference prefix>
      | <CLI by-value prefix>

<CLI by-reference prefix> ::= SQLR

<CLI by-value prefix> ::= SQL

<CLI generic name> ::=
        AllocConnect
      | AllocEnv
      | AllocStmt
      | BindCo
      | BindParam
      | Cancel
      | ColAttribute
      | Connect
      | DescribeCol
      | Disconnect
      | Error
      | ExecDirect
      | Execute
      | Fetch
      | FreeConnect
      | FreeEnv
      | FreeStmt
      | GetCol
      | GetCursorName
      | NumResultCols
      | Prepare
      | RowCount
      | SetCursorName
      | SetParamValue
      | Transact

<CLI parameter list> ::=
      <left paren> <CLI parameter declaration>
        [ { <comma> <CLI parameter declaration> }... ] <right paren>

<CLI parameter declaration> ::=
      <CLI parameter name> <CLI parameter mode> <CLI parameter data type>

<CLI parameter name> ::= !! See the individual CLI routine definitions

<CLI parameter mode> ::=
```

Working Draft

```
          IN
        | OUT
        | DEFIN
        | DEFOUT

<CLI parameter data type> ::=
          INTEGER
        | SMALLINT
        | POINTER
        | CHARACTER <left paren> <length> <right paren>

<CLI returns clause> ::= RETURNS SMALLINT
```

Syntax Rules

1) <CLI routine> is a pre-defined routine written in a standard programming language that is invoked by a compilation unit of the same standard programming language. Let *HL* be that standard programming language. *HL* shall be one of Ada, C, COBOL, Fortran, MUMPS, Pascal and PL/I.

2) <CLI routine> that contains a <CLI returns clause> is called a *CLI function*. A <CLI routine> that does not contain a <CLI returns clause> is called a *CLI procedure*.

3) For each CLI function *CF*, there is a corresponding CLI procedure *CP*, with the same <CLI routine name>. The <CLI parameter list> for *CP* is the same as the <CLI parameter list> for *CF* but with the following additional <CLI parameter declaration>:

   ReturnCode OUT SMALLINT

4) *HL* shall support either the invocation of *CF* or the invocation of *CP*. It is implementation-defined which is supported.

5) Case:

   a) If <CLI parameter mode> is IN, then the parameter is an *input parameter*. The value of an input argument is established when a CLI routine is invoked.

   b) If <CLI parameter mode> is OUT, then the parameter is an *output parameter*. The value of an output argument is established when a CLI routine is executed.

   c) If <CLI parameter mode> is DEFIN, then the parameter is a *deferred input parameter*. The value of a deferred input argument for a CLI routine *R* is not established when *R* is invoked, but subsequently during the execution of a related CLI routine.

   d) If <CLI parameter mode> is DEFOUT, then the parameter is a *deferred output parameter*. The value of a deferred output argument for a CLI routine *R* is not established by the execution of *R* but subsequently by the execution of a related CLI routine.

   Note: CLI routines that have deferred parameters are not provided by implementations that do not support deferred parameters. This means that these implementations do not provide the option of binding dynamic parameter value sources and result targets.

6) An output, deferred output or deferred input argument provides an address. It is either a non-pointer host variable passed by reference or a pointer host-variable passed by value.

Working Draft

7) A *by-value version* of a CLI routine is a version that expects each of its input parameters to be provided as an actual value. A *by-reference version* of a CLI routine is a version that expects each of its input parameters to be provided as an address.

   Case:

   a) If the parameter passing paradigm for *HL* is (only) by-value, then both by-value and by-reference versions of each CLI routine shall be supported.

   b) Otherwise, only by-reference versions of each CLI routine shall be supported.

8) If a <CLI routine> is a by-reference routine, then its <CLI routine name> contains a <CLI by-reference prefix>. Otherwise, its <CLI routine name> contains a <CLI by-value prefix>.

9) The <CLI routine name> for a CLI function shall be different from the <CLI routine name> of any other CLI function. The <CLI routine name> for a CLI procedure shall be different from the <CLI routine name> of any other CLI procedure.

10) <CLI routine name>s are unique within the first 10 characters. Any implementation that requires routine names to be unique within less than 10 characters effectively replaces the <CLI routine name> with an abbreviated name according to the following rules:

    a) Any <CLI by-value prefix> remains unchanged.

    b) Any <CLI by-reference prefix> is replaced by SQR.

    c) The <CLI generic name> is replaced by an abbreviated version according to Table 1, "Abbreviated CLI generic names".

11) Let *CR* be a <CLI routine> and let *RN* be its <CLI routine name>. Let *RNU* be the value of UPPER(*RN*).

    Case:

    a) If an implementation supports case sensitive routine names, then the name used for the invocation of *CR* shall be *RN*.

    b) If an implementation does not support <simple Latin lower case letter>s, then the name used for the invocation of *CR* shall be *RNU*.

    c) If an implementation does not support case sensitive routine names, then the name used for the invocation of *CR* shall be *RN* or *RNU*.

12) Let ''operative data type correspondence table'' be the data type correspondence table for *HL* as specified in <REFERENCE>(typecorr\FULL). Refer to the two columns of the operative data type correspondence table as the "SQL data type column" and the "host data type column".

13) Let *TI*, *TS*, *TC*, and *TV* be the types listed in the host data type column for the rows that contains INTEGER, SMALLINT, CHARACTER(*L*) and CHARACTER VARYING(*L*), respectively, in the SQL data type column.

    a) If *TS* is "None", then let *TS* = *TI*.

    b) If *TC* is "None", then let *TC* = *TV*.

Working Draft

c) For each parameter *P*:

i) If *HL* is *C* and the CLI routine is a by-value routine and *P* is an input parameter, then let *TI* = "long" and *TS* = "short".

ii) Case:

1) If the CLI parameter data type is INTEGER, then the type of the corresponding argument shall be *TI*.

2) If the CLI parameter data type is SMALLINT, then the type of the corresponding argument shall be *TS*.

3) If the CLI parameter data type is CHARACTER(*L*), then the type of the corresponding argument shall be *TC*.

4) If the CLI parameter data type is POINTER, then:

Case:

A) If *HL* is *C*, then the type of the corresponding argument shall be "void *".

B) Otherwise, the type of the corresponding argument shall be *TI*, *TS*, or *TC*.

d) If the CLI routine is a CLI function, then:

Case:

i) If *HL* is *C*, then the type of the returned value is "short".

ii) Otherwise, the type of the returned value is *TS*.

Access Rules:

None.

General Rules:

1) The rules for invocation of the <CLI routine> are specified in Subclause 5.3, "<CLI routine> invocation".

Working Draft

5.3 <CLI routine> invocation

Function

Specify the rules for invocation of a <CLI routine>.

Syntax Rules

1) Let *HL* be the standard programming language of the invoking host program.

2) A CLI function or CLI procedure is invoked by the *HL* mechanism for invoking functions or procedures, respectively.

3) Let *RN* be the <CLI routine name> of the <CLI routine> invoked by the host program. The number of arguments provided in the invocation shall be the same as the number of <CLI parameter declaration>s for *RN*.

4) Let *DA* be the data type of the *i*-th argument in the invocation and let *DP* be the <CLI parameter data type> of the *i*-th <CLI parameter declaration> of *RN*. *DA* shall be the *HL* equivalent of DP as specified by the rules of Subclause 5.2, "<CLI routine>".

General Rules

1) If the value of any input argument provided by the host program falls outside the set of allowed values of the data type of the parameter, or if the value of any output argument resulting from the execution of the <CLI routine> falls outside the set of values supported by the host program for that parameter, then the effect is implementation-defined.

2) Let *GRN* be the <CLI generic name> of *RN*.

3) When the <CLI routine> is called by the host program:

   Case:

   a) If *RN* is a CLI routine with a statement handle as an input parameter and *GRN* is not 'Error', then:

      i) The values of all input arguments to *RN* are established.

      ii) If the statement handle does not identify an allocated SQL-statement, then an exception condition is raised: *invalid handle*. Otherwise, let *S* be the allocated SQL-statement identified by the statement handle.

      iii) If *GRN* is not 'Cancel', then any status records associated with S are destroyed.

      iv) Let *C* be the allocated SQL-connection with which *S* is associated.

      v) If there is no established SQL-connection associated with *C*, then an exception condition is raised: *connection exception—connection does not exist*. Otherwise, let *EC* be the established SQL-connection associated with *C*.

      vi) If *EC* is not the current connection, then the General Rules of Subclause 5.4.1, "Implicit set connection", are applied to *EC* as the dormant connection.

      vii) *RN* is invoked.

Working Draft

b) Otherwise:

i) The values of all input arguments to *RN* are established.

ii) *RN* is invoked.

4) Case:

a) If the <CLI routine> is a CLI function, then:

i) The values of all output arguments are established.

ii) Let *RC* be the return value.

b) If the <CLI routine> is a CLI procedure, then:

i) The values of all output arguments are established except for the argument associated with the ReturnCode parameter.

ii) Let *RC* be the argument associated with the ReturnCode parameter.

5) Case:

a) If *RN* executed successfully, then:

i) Either a completion condition is raised: *successful completion*, or a completion condition is raised: *warning*, or a completion condition is raised: *no data*, as determined by the rules of this and other Subclauses of this International Standard.

ii) Case:

1) If a completion condition is raised: *successful completion*, then *RC* is set to indicate Success .

2) If a completion condition is raised: *warning*, then *RC* is set to indicate Success with information .

3) If a completion condition is raised: *no data*, then *RC* is set to indicate No data found .

iii) If *RN* did not execute successfully, then:

iv) All changes made to SQL data or schemas by the execution of *RN* are canceled.

v) An exception condition is raised as determined by the rules in this and other Subclauses of this International Standard.

vi) Case:

1) If an exception condition is raised: *invalid handle*, then *RC* is set to indicate Invalid Handle .

2) Otherwise, *RC* is set to indicate Error .

6) Case:

a) If *GRN* is not 'Error' and *RC* indicates either Error or Success with information , then status information resulting from the execution of *RN* is placed into status records as specified in Subclause 5.4.5.

Working Draft

b) Otherwise, no status records are generated.

Working Draft

5.4 CLI common elements

5.4.1 Implicit set connection

Function

Specify the rules for an implicit SET CONNECTION statement.

General Rules

1) Let *DC* be a dormant SQL-connection specified in an application of this Subclause.

2) If an SQL-transaction is active for the current SQL-connection and the implementation does not support transactions that affect more than one SQL-server, then an exception condition is raised: *feature not supported—multiple server transactions*.

3) If *DC* cannot be selected, then an exception condition is raised: *connection exception—connection failure*.

4) The current SQL-connection and current SQL-session become a dormant SQL-connection and a dormant SQL-session, respectively. The SQL-session context information is preserved and is not affected in any way by operations performed over the selected SQL-connection.

   Note: The SQL-session context information is defined in Subclause 4.29, "SQL-sessions".

5) *DC* becomes the current SQL-connection and the SQL-session associated with *DC* becomes the current SQL-session. All SQL-session context information is restored to the same state as at the time *DC* became dormant.

   Note: The SQL-session context information is defined in Subclause 4.29, "SQL-sessions".

6) The SQL-server for the subsequent execution of SQL-statements via CLI routine invocations is set to that of the current SQL-connection.

5.4.2 Implicit cursor

Function

Specify the rules for an implicit DECLARE CURSOR and OPEN statement.

General Rules

1) Let *SS* and *AS* be a *SELECT SOURCE* and *ALLOCATED STATEMENT* specified in an application of this Subclause.

2) Case:

   a) If there is no cursor associated with *AS*, then a cursor is associated with *AS*. If a cursor name is associated with *AS*, then that is used as the name of the cursor; otherwise the name of the cursor is a unique implementation-dependent name that has the prefix ''SQLCUR''. Let *CN* be the name of the cursor.

   b) Otherwise, let *CN* be the name of the cursor associated with *AS*.

3) Let *CR* be the following <declare cursor>:

   DECLARE *CN* CURSOR FOR *SS*

Working Draft

4) The General Rules of Subclause 5.4.3, "Implicit using clause", are applied to 'OPEN', *SS*, and *AS* as *TYPE*, *SOURCE*, and *ALLOCATED STATEMENT*, respectively.

5) Cursor *CN* is opened in the following steps:

   a) A copy of *SS* is effectively created in which:

      i) Each <dynamic parameter specification> is replaced by the value of the corresponding dynamic parameter.

      ii) Each <value specification> generally contained in *SS* that is USER, CURRENT_USER, SESSION_USER or SYSTEM_USER is replaced by the value resulting from evaluation of USER, CURRENT_USER, SESSION_USER or SYSTEM_USER, respectively, with all such evaluations effectively done at the same instant in time; and

      iii) Each <datetime value function> generally contained in *SS* is replaced by the value resulting from evaluation of that <datetime value function>, with all such evaluations effectively done at the same instant in time.

   b) Let *T* be the table specified by the copy of *SS*.

   c) A table descriptor for *T* is effectively created.

   d) The General Rules of Subclause 13.1, "<declare cursor>", are applied to *CR*.

   e) Cursor *CN* is placed in the open state and its position is before the first row of *T*.

5.4.3 Implicit using clause

Function

Specify the rules for an implicit <using arguments>.

General Rules

1) *T*, *S*, and *AS* be a *TYPE*, *SOURCE*, and *ALLOCATED STATEMENT* specified in an application of this Subclause.

2) If *T* = 'EXECUTE' or *T* = 'OPEN', then

   a) Let *DP* be the number of <dynamic parameter specification>s in *S*.

   b) Let *BP* be the number of <dynamic parameter specification> value sources that are currently bound to *AS* and let *SP* be the number of <dynamic parameter specification> values that are currently set for *AS*. Let *PN* be the parameter number associated with a bound <dynamic parameter specification> value source or a set <dynamic parameter specification> value and let *HP* be the value of MAX(*PN*).

   c) If *HP* does not equal $(BP + SP)$ or if *DP* does not equal *HP*, then an exception condition is raised: *dynamic SQL error— using clause does not match dynamic parameter specifications*.

   d) If *BP* is not zero, then for each bound <dynamic parameter specification> value source *BPVS*:

      i) The <dynamic parameter specification> value and its associated indicator value are established. Let these values be *SV* and *IV*, respectively.

Working Draft

ii) Let *BPN* be the parameter number associated with *BPVS*.

iii) Let *VT* be the value type associated with *BPVS*.

iv) If *VT* indicates CHARACTER, then:

1) Case:

A) If *IV* is greater than zero, then let *L* be *IV*.

B) If *IV* is -3, then let *L* be the number of characters of *SV* that precede the implementation-defined null character that terminates a C character string. If *SV* does not contain the implementation-defined null character that terminates a C character string, then an exception condition is raised: *data exception—unterminated string*.

C) If *IV* is less than 1 but is neither -1 nor -3, then an exception condition is raised: *data exception—error in assignment*.

2) *L* is the length of *SV*.

v) Case:

1) If *IV* is -1, then the value of dynamic parameter *BPN* is the general null value.

2) If *VT* is 99, then *SV* is the value of dynamic parameter *BPN*.

3) Otherwise:

A) *VT* indicates the data type of *SV*.

B) Let *TDT* be the target data type indicated by the target SQL type associated with *BPVS* and its associated length, precision, and scale as applicable.

C) If the <cast specification>

CAST (*SV* AS *TDT*)

violates the General rules of Subclause 6.10, "<cast specification>", then an exception condition is raised in accordance with the General Rules of Subclause 6.10, "<cast specification>".

D) The <cast specification>

CAST (*SV* AS *TDT*)

is effectively performed, and is the value of dynamic parameter *BPN*.

e) If *SP* is not zero, then for each set <dynamic parameter specification> value *SPV*:

i) Let *SPN* be the parameter number associated with *SPV*.

ii) *SPV* is the value of dynamic parameter *SPN*.

f) If the data type of any dynamic parameter value does not match the data type of the corresponding dynamic parameter, then an exception condition is raised: *data exception—error in assignment*.

Working Draft

3) If *T* = 'FETCH', then:

a) Let *BT* be the number of bound result targets associated with *AS*. If *BT* is zero, then no further rules of this Subclause are applied.

b) Let *D* be the degree of the table described by *S*.

c) If *BT* is greater than *D*, then an exception condition is raised: *dynamic SQL error—using clause does not match target specifications*.

d) For each <select list> column for which there is a bound result target *BRT* associated with *AS*:

i) Let *SV* and *SDT* be the value and data type of the <select list> column.

ii) Let *IT* be the indicator target associated with *BRT*.

iii) Let *TT* be the target type associated with *BRT*.

iv) Case:

1) If *TT* is 99, then let *TV* = *SV*.

2) If *TT* indicates CHARACTER, then:

A) Let *L* be the length value associated with *BRT*.

B) If the <cast specification>

CAST (*SV* AS CHARACTER(*L*))

violates the General Rules of Subclause 6.10, "<cast specification>", then an exception condition is raised in accordance with the General Rules of Subclause 6.10, "<cast specification>".

C) The <cast specification>

CAST (*SV* AS CHARACTER(*L*))

is effectively performed, and is the target value *TV*.

3) If *TT* indicates the same data type as *SDT*, then let *TV* = *SV*.

4) Otherwise, it is implementation-defined whether *SV* can be converted to the data type indicated by *TT*.

Case:

A) If *SV* cannot be converted to the data type indicated by *TT*, then an exception condition is raised: *dynamic SQL error—restricted data type attribute violation*.

B) Otherwise, *TV* is derived from *SV* according to implementation-defined rules.

v) If *TV* is a null value, then *IT* is set to the negative number associated with that null value.

vi) If *TV* is not a null value, then:

1) *BRT* is set to *TV*.

Working Draft

2) Case:

   A) If *SDT* is CHARACTER or CHARACTER VARYING, then *IT* is set to the length of *SV*.

   B) Otherwise, *IT* is set to zero.

5.4.4 Tables associated with CLI

Table 1—Abbreviated CLI generic names

Generic Name	Abbreviation
AllocConnect	AC
AllocEnv	AE
AllocStm	AS
BindCol	BC
BindPara	BP
Cancel	CAN
ColAttribute	COL
Connect	CON
DescribeCol	DC
Disconnect	DIS
Error	ERR
ExecDirect	ED
Execute	EX
Fetch	FT
FreeConnect	FC
FreeEnv	FE
FreeStmt	FS
GetCol	GC
GetCursorName	GCN
NumResultCol	NRC
Prepare	PR
RowCount	RC
SetCursorName	SCN
SetParamValue	SPV
Transact	TR

Working Draft

Table 2—Codes used for SQL data types in CLI

SQL data type	Code
CHARACTER	1
CHARACTER VARYING	12
DECIMAL	3
DOUBLE PRECISION	8
FLOAT	6
INTEGER	4
NUMERIC	2
REAL	7
SMALLINT	5

Table 3—Codes used for buffer data types

Buffer data type	Code
CHARACTER	1
DOUBLE PRECISION	8
INTEGER	4
REAL	7
SMALLINT	5

5.4.5 Return codes and status records

The execution of a CLI routine causes one or more conditions to be raised as determined by the rules of other Subclauses of this International Standard. The basic result of the execution is indicated by a code that is returned either as the result of a CLI routine that is a CLI function or as the value of the ReturnCode argument of a CLI routine that is a CLI procedure.

The values and meanings of the return codes are as follows:

— A value of zero indicates Success. The CLI routine executed successfully.

— A value of 1 indicates Success with information. The CLI routine executed successfully but a completion condition was raised: *warning*.

— A value of 100 indicates No data found. The CLI routine executed successfully but a completion condition was raised: *no data*.

— A value of -1 indicates Error. The CLI routine did not execute successfully because an exception condition other than *invalid handle* was raised.

Working Draft

— A value of -2 indicates Invalid Handle. The CLI routine did not execute successfully because an exception condition was raised: *invalid handle*.

Note: If the CLI routine did not execute successfully, then the values of all output arguments are implementation-dependent unless explicitly defined by this International Standard.

In addition to providing the return code, for all CLI routines other than Error, the implementation records information about conditions that result in Success with information or Error. For each such condition information is recorded in a status record associated with the resource being utilised. The application accesses the status records one at a time by invoking the CLI routine Error. The application identifies which set of status records is to be accessed by providing the handle of the relevant resource as an input argument.

The order in which status records are generated is implementation-dependent. The status record accessed by Error is the first status record, if any, in the relevant set of status records.

At the beginning of the execution of any CLI routine other than Error, all existing status records for the resource being utilised are destroyed. After Error has been used to access a status record, that status record is destroyed.

Refer to a condition that causes a status record to be generated as a status condition. A status record contains the following information about a status condition:

— The SQLSTATE value corresponding to the status condition.

— An integer that is the implementation-defined native error code corresponding to the status condition.

— A character string that is the implementation-defined message text that describes the status condition.

Note: This may be set to <space>s or to a zero-length string if there is no applicable message text.

5.4.6 CLI-specific status codes

Some of the conditions that can occur during the execution of CLI routines are CLI-specific. The corresponding status codes are listed in the table below rather than in Subclause 22.1, "SQLSTATE".

Table 4—SQLSTATE class and subclass values for CLI-specific conditions

Condition	Class	Subcondition	Subclass
CLI-specific condition	HY	(*no subclass*)	000
		function sequence error	010
		invalid argument value	009
		invalid column number	002
		invalid transaction operation code	012

Working Draft

Table 4—SQLSTATE class and subclass values for CLI-specific conditions (Cont.)

Condition	Class	Subcondition	Subclass
		memory allocation error	001
		memory management error	013
		no cursor name available	015
		operation canceled	008
		program type out of range	003
		SQL data type out of range	004

5.4.7 Miscellaneous characteristics

5.4.7.1 Handles

The allocation routines return an identifier, known as a handle, that uniquely identifies the allocated resource. Although the CLI parameter data type for a handle parameter is INTEGER, its value has no meaning in any other context and should not be used as a numeric operand or modified in any way.

If the related resource cannot be allocated, then a zero handle value is returned. However, even if a resource has been successfully allocated, processing of that resource can subsequently fail due to insufficient memory. When this situation occurs, an exception condition is raised: *CLI-specific condition—memory management error*.

The validity of a handle in a compilation unit other than the one in which the identified resource was allocated is implementation-defined.

Note: Specifying (the address of) a valid handle as the output argument of the allocation routine for that resource type does not have the effect of reinitializing the identified resource. Instead a new resource is allocated and a new handle value overwrites the old one.

5.4.7.2 Null terminated strings

An input character string provided by the application may be terminated by the implementation-defined null character that terminates C character strings. If this technique is used, the application may set the associated length argument to either the length of the string excluding the null terminator or to -3 to indicate that the string is null terminated.

All output character strings returned by the implementation are terminated by the implementation-defined null character that terminates C character strings.

Working Draft

5.4.7.3 Null pointers

If the standard programming language of the invoking host program supports pointers, then the application may provide a zero-valued pointer, referred to as a *null pointer*, in lieu of an output argument that is to receive the length of a returned character string. This indicates that the application does not wish to be informed of the length of the returned character string.

If the application provides a null pointer in any other circumstances, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

Working Draft

5.5 CLI routines

Subclause 5.2, "<CLI routine>", defines a generic CLI routine. This Subclause describes the individual CLI routines in alphabetical order.

For convenience, the variable <CLI name prefix> is omitted and the <CLI generic name> is used for the descriptions. For presentation purposes (and purely arbitrarily), the routines are presented as functions rather than as procedures.

5.5.1 AllocConnect

Function

Allocate an SQL-connection.

Definition

```
AllocConnect (
       EnvironmentHandle   IN   INTEGER,
       ConnectionHandle    OUT  INTEGER )
       RETURNS SMALLINT
```

Description

Case:

a) If EnvironmentHandle does not identify an allocated SQL-environment or if it identifies an allocated skeleton SQL-environment, then ConnectionHandle is set to zero and an exception condition is raised: *CLI-specific condition—invalid handle*.

b) Otherwise:

i) Let *E* be the allocated SQL-environment identified by EnvironmentHandle.

ii) Any status records associated with *E* are destroyed.

1) Case:

a) If the memory requirements to manage an SQL-connection cannot be satisfied, then ConnectionHandle is set to zero and an exception condition is raised: *CLI-specific condition—memory allocation error*.

b) If the resources to manage an SQL-connection cannot be allocated for implementation-defined reasons, then ConnectionHandle is set to zero and an implementation-defined exception condition is raised.

c) Otherwise, the resources to manage an SQL-connection are allocated and are referred to as an *allocated SQL-connection*. The allocated SQL-connection is associated with *E* and is assigned a unique value that is returned in ConnectionHandle.

2) The maximum number of SQL-connections that can be allocated at one time is implementation-defined.

Working Draft

5.5.2 AllocEnv

Function

Allocate an SQL-environment.

Definition

```
AllocEnv (
       EnvironmentHandle   OUT  INTEGER )
       RETURNS SMALLINT
```

Description

1) Case:

   a) If the memory requirements to manage an SQL-environment cannot be satisfied, then EnvironmentHandle is set to zero and an exception condition is raised: *CLI-specific condition—memory allocation error*.

      Note: No status record is generated in this case as there is no valid environment handle to pass to Error in order to obtain status information.

   b) If the resources to manage an SQL-environment cannot be allocated for implementation-defined reasons, then an implementation-defined exception condition is raised. A skeleton SQL-environment is allocated and is assigned a unique value that is returned in EnvironmentHandle.

   c) Otherwise, the resources to manage an SQL-environment are allocated and are referred to as an allocated SQL-environment. The allocated SQL-environment is assigned a unique value that is returned in EnvironmentHandle.

2) The maximum number of SQL-environments that can be allocated at one time is implementation-defined.

5.5.3 AllocStmt

Function

Allocate an SQL-statement.

Definition

```
AllocStmt (
       ConnectionHandle    IN   INTEGER,
       StatementHandle     OUT  INTEGER )
       RETURNS SMALLINT
```

Working Draft

Description

1) Case:

   a) If ConnectionHandle does not identify an allocated SQL-connection, then StatementHandle is set to zero and an exception condition is raised: *CLI-specific condition—invalid handle*.

   b) Otherwise:

      i) Let *C* be the allocated SQL-connection identified by ConnectionHandle.

      ii) Any status records associated with *C* are destroyed.

2) If there is no established SQL-connection associated with *C*, then StatementHandle is set to zero and an exception condition is raised: *connection exception—connection does not exist*. Otherwise, let *EC* be the established SQL-connection associated with *C*.

3) If *EC* is not the current SQL-connection, then the General Rules of Subclause 5.4.1, "Implicit set connection", are applied to *EC* as the dormant SQL-connection.

4) Case:

   a) If the memory requirements to manage an SQL-statement cannot be satisfied, then StatementHandle is set to zero and an exception condition is raised: *CLI-specific condition—memory allocation error*.

   b) If the resources to manage an SQL-statement cannot be allocated for implementation-defined reasons, then StatementHandle is set to zero and an implementation-defined exception condition is raised.

   c) Otherwise, the resources to manage an SQL-statement are allocated and are referred to as an *allocated SQL-statement*. The allocated SQL-statement is associated with *C* and is assigned a unique value that is returned in StatementHandle.

5) The maximum number of SQL-statements that can be allocated at one time is implementation-defined.

5.5.4 BindCol

Function

Define a result target.

Definition

```
BindCol (
       StatementHandle  IN        INTEGER,
       ColumnNumber     IN        SMALLINT,
       TargetType       IN        SMALLINT,
       TargetValue      DEFOUT    POINTER,
       BufferLength     IN        INTEGER,
       IndicatorValue   DEFOUT    INTEGER )
       RETURNS SMALLINT
```

Working Draft

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) Let *CN* be the value of ColumnNumber. If *CN* is less than 1, then an exception condition is raised: *CLI-specific condition—invalid column number*.

3) If there is a select source associated with *S*, then let *D* be the degree of the table described by the select source. If *CN* is greater than *D*, then an exception condition is raised: *CLI-specific condition—invalid column number*.

4) Let *TT* be the value of TargetType.

5) Let *HL* be the standard programming language of the invoking host program. Let ''operative data type correspondence table'' be the data type correspondence table for *HL* as specified in <REFERENCE>(typecorr\FULL). Refer to the two columns of the operative data type correspondence table as the ''SQL data type column'' and the ''host data type column''.

6) If any of the following are true, then an exception condition is raised: *CLI-specific condition—program type out of range*.

   a) *TT* is not greater than zero.

   b) *TT* is greater than zero but is neither 99 nor one of the code values in Table 3, "Codes used for buffer data types".

   c) *TT* is one of the code values in Table 3, "Codes used for buffer data types", but the row that contains the corresponding data type in the SQL data type column of the operative data type correspondence table contains 'None' in the host data type column.

7) Let *BL* be the value of BufferLength. If *TT* indicates CHARACTER and BL is not greater than zero, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

8) If the column number associated with any result target currently bound for *S* is *CN*, then that result target is unbound.

9) TargetValue becomes the result target for column *CN* bound for *S* and has the following associated attributes: IndicatorValue is its associated indicator target; *TT* is its associated coded target type; and *BL* is its associated length value.

5.5.5 BindParam

Function

Define a value source for a <dynamic parameter specification>.

Definition

```
BindParam (
       StatementHandle  IN     INTEGER,
       ParameterNumber  IN     SMALLINT,
       ValueType        IN     SMALLINT,
       SqlType          IN     SMALLINT,
       LengthPrecision  IN     INTEGER,
       Scale            IN     SMALLINT,
       ParameterValue   DEFIN  POINTER,
       IndicatorValue   DEFIN  INTEGER )
```

Working Draft

```
RETURNS SMALLINT
```

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) Let *PN* be the value of ParameterNumber. If *PN* is less than 1, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

3) Let *VT* be the value of ValueType.

4) Let *HL* be the standard programming language of the invoking host program. Let ''operative data type correspondence table'' be the data type correspondence table for *HL* as specified in <REFERENCE>(typecorr\FULL). Refer to the two columns of the operative data type correspondence table as the ''SQL data type column'' and the ''host data type column''.

5) If any of the following are true, then an exception condition is raised: *CLI-specific condition—program type out of range*.

   a) *VT* is not greater than zero.

   b) *VT* is greater than zero but is neither 99 nor one of the code values in Table 3, "Codes used for buffer data types".

   c) *VT* is one of the code values in Table 3, "Codes used for buffer data types", but the row that contains the corresponding data type in the SQL data type column of the operative data type correspondence table contains 'None' in the host data type column.

6) Let *ST* be the value of SqlType. If *ST* is not one of the code values in Table 2, "Codes used for SQL data types in CLI", then an exception condition is raised: *CLI-specific condition—SQL data type out of range*.

7) Case:

   a) If *ST* indicates a character string type, then LengthPrecision is the length associated with *ST*.

   b) If *ST* indicates NUMERIC or DECIMAL or FLOAT, then LengthPrecision is the precision associated with *ST*.

   c) If *ST* indicates NUMERIC or DECIMAL, then Scale is the scale associated with *ST*.

8) If *VT* is greater than zero, then:

   a) Let *SV* be a <value expression> whose data type is the data type indicated by *VT*.

   b) Let *TDT* be the target data type indicated by *ST* and its associated length, precision, and scale as applicable.

   c) If the <cast specification>

      CAST (*SV* AS *TDT*)

      violates the Syntax Rules of Subclause 6.10, "<cast specification>", then an exception condition is raised: *dynamic SQL error—restricted data type attribute violation*.

Working Draft

9) Case:

   a) If the parameter number associated with any <dynamic parameter specification> value source currently bound for *S* is *PN*, then that <dynamic parameter specification> value source is unbound.

   b) If the parameter number associated with any <dynamic parameter specification> value currently set for *S* is *PN*, then that <dynamic parameter specification> value is unset.

10) ParameterValue becomes the <dynamic parameter specification> value source for parameter *PN* bound for *S* and has the following associated attributes:

   a) IndicatorValue is its associated indicator value source.

   b) *VT* is its associated coded value type.

   c) *ST* is its associated coded target SQL type and has an associated length, precision, and scale, as applicable.

5.5.6 Cancel

Function

Attempt to cancel execution of a CLI routine.

Definition

```
Cancel (
       StatementHandle IN   INTEGER )
       RETURNS SMALLINT
```

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no CLI routine concurrently operating on *S*, then a completion condition is raised: *success*, and no further rules of this Subclause are applied. Otherwise, let *RN* be the routine name of the concurrent CLI routine.

   Note: The method of passing control between concurrently operating programs is implementation-dependent.

3) The SQL-server is requested to cancel the execution of *RN*.

4) If the SQL-server rejects the cancelation request, then an exception condition is raised: *Remote Database Access exception condition*.

5) If the SQL-server accepts the cancelation request, then a completion condition is raised: *success*.

   Note: Acceptance of the request does not guarantee that the execution of *RN* will be canceled.

6) If the SQL-server succeeds in canceling the execution of *RN*, then an exception condition: *CLI-specific condition—operation canceled*, is raised for RN.

   Note: Canceling the execution of *RN* does not destroy any status records already generated by its execution.

Working Draft

5.5.7 ColAttribute

Function

Get a column attribute.

Definition

```
ColAttribute (
       StatementHandle     IN   INTEGER,
       ColumnNumber        IN   SMALLINT,
       AttributeIdentifier IN   SMALLINT,
       CharacterAttribute  OUT  CHARACTER(L),
       BufferLength        IN   SMALLINT,
       AttributeLength     OUT  SMALLINT,
       NumericAttribute    OUT  INTEGER )
       RETURNS SMALLINT
```

where *L* is the value of BufferLength and has a maximum value of 128.

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) Let *AN* be the value of AttributeNumber. If *AN* is less than 1 or greater than 8, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

3) If there is a select source associated with *S*, then let *SS* be the select source associated with *S*, let *T* be the table specified by SS, and let *D* be the degree of *T*. Otherwise, let *D* be zero.

4) Case:

   a) If *AN* = 1, then:

      Case:

      i) If there is no prepared or directly executed statement associated with *S*, then an exception condition is raised: *CLI-specific condition—function sequence error*.

      ii) Otherwise, NumericAttribute is set to *D* and no further rules of this Subclause are applied.

   b) Otherwise, if *D* is zero, then an exception condition is raised: *CLI-specific condition—function sequence error*.

5) Let *N* be the value of ColumnNumber. If *N* is less than 1 or greater than *D*, then an exception condition is raised: *CLI-specific condition—invalid column number*.

6) Let *BL* be the value of BufferLength. If *BL* is not greater than zero, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

7) Let *C* be the *n*-th column of *T*. A single attribute of *C* is retrieved.

Working Draft

Case:

a) If *AN* = 2, then the name of *C* is retrieved as follows:

   i) Let *CN* be the implementation-dependent or <derived column> name of *C* and let *L* be the length of *CN*.

   ii) If *L* is not greater than *BL*, then CharacterAttribute is set to *CN* and AttributeLength is set to *L*.

   iii) If *L* is greater than *BL*, then CharacterAttribute is set to the first *BL* characters of *CN*, AttributeLength is set to *L* and a completion condition is raised: *string data, right truncation*.

b) If *AN* = 3, then NumericAttribute is set to a code, as shown in Table 2, "Codes used for SQL data types in CLI", to indicate the data type of *C*.

c) If *AN* = 4 and *C* is a character string column, then NumericAttribute is set to the length or maximum length of *C*.

d) If *AN* = 5 and *C* is a numeric column, then NumericAttribute is set to the precision of *C*.

e) If *AN* = 6 and *C* is an exact numeric column, then NumericAttribute is set to the scale of *C*.

f) If *AN* = 7, then NumericAttribute is set to indicate the nullability of *C*. If *C* is possibly nullable, then NumericAttribute is set to 1; otherwise it is set to zero.

g) If *AN* = 8, then NumericAttribute is set to indicate *C*'s name type. If the name is implementation-dependent, then NumericAttribute is set to 1; otherwise it is set to zero.

5.5.8 Connect

Function

Establish a connection.

Definition

```
Connect (
       ConnectionHandle     IN    INTEGER,
       ServerName           IN    CHARACTER(L1),
       NameLength1          IN    SMALLINT,
       UserName             IN    CHARACTER(L2),
       NameLength2          IN    SMALLINT )
       RETURNS SMALLINT
```

where:

— *L1* is determined by the value of NameLength1 and has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

— *L2* is determined by the value of NameLength2 and has a maximum value of 128.

Working Draft

Description

1) Case:

   a) If ConnectionHandle does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition—invalid handle*.

   b) Otherwise:

      i) Let *C* be the allocated SQL-connection identified by ConnectionHandle.

      ii) Any status records associated with *C* are destroyed.

2) If an SQL-transaction is active for the current SQL-connection and the implementation does not support transactions that affect more than one SQL-server, then an exception condition is raised: *feature not supported—multiple server transactions*.

3) If there is an established SQL-connection associated with *C*, then an exception condition is raised: *connection exception—connection name in use*.

4) Let *NL1* be the value of NameLength1.

   Case:

   a) If *NL1* is greater than zero, then let *L1* be *NL1*.

   b) If *NL1* is -3, then let *L1* be the number of characters of ServerName that precede the implementation-defined null character that terminates a C character string. If ServerName does not contain the implementation-defined null character that terminates a C character string, then an exception condition is raised: *data exception—unterminated C string*.

   c) Otherwise, if *NL1* is not zero, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

5) Case:

   a) If *NL1* is zero, then let 'DEFAULT' be the value of *SN*.

   b) Otherwise, let *SV* be the first *L1* characters of ServerName and let *SN* be the value of

      TRIM (BOTH ' ' FROM *SV*)

6) Let *E* be the allocated SQL-environment with which *C* is associated and let *NL2* be the value of NameLength2.

7) If the value of *SN* is 'DEFAULT', then:

   a) If an established default SQL-connection is associated with an allocated SQL-connection associated with *E*, then an exception condition is raised: *connection exception—connection name in use*.

   b) If *NL2* is not zero, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

8) If the value of *SN* is not 'DEFAULT', then:

   a) If *NL2* is greater than zero, then let *L2* be *NL2*.

Working Draft

b) If *NL2* is -3, then let *L2* be the number of characters of UserName that precede the implementation-defined null character that terminates a C character string. If UserName does not contain the implementation-defined null character that terminates a C character string, then an exception condition is raised: *data exception—unterminated C string*.

c) If *NL2* is less than zero but not -3, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

d) Case:

i) If *NL2* is zero, then let *UN* be an implementation-defined <authorization identifier>.

ii) Otherwise:

1) Let *UV* be the first *L2* characters of UserName and let *UN* be the result of

TRIM (BOTH ' ' FROM *UV*)

2) If *UN* does not conform to the Format and Syntax Rules of a <regular identifier>, then an exception condition is raised: *invalid authorization identifier*.

3) If *UN* violates any implementation-defined restrictions on its value, then an exception condition is raised: *invalid authorization identifier*.

9) Case:

a) If the value of *SN* is 'DEFAULT', then the default SQL-session is initiated and associated with the default SQL-server. The method by which the default SQL-server is determined is implementation-defined.

b) Otherwise, an SQL-session is initiated and associated with the SQL-server identified by *SN*. The method by which *SN* is used to determine the appropriate SQL-server is implementation-defined.

10) If an SQL-session is successfully initiated, then:

a) The current SQL-connection and current SQL-session, if any, become a dormant SQL-connection and a dormant SQL-session respectively. The SQL-session context information is preserved and is not affected in any way by operations performed over the initiated SQL-connection.

Note: The SQL-session context information is defined in Subclause 4.29, "SQL-sessions".

b) The initiated SQL-session becomes the current SQL-session and the SQL-connection established to that SQL-session becomes the current SQL-connection and is associated with *C*.

Note: If an SQL-session is not successfully initiated, then the current SQL-connection and current SQL-session, if any, remain unchanged.

11) If the SQL-client cannot establish the SQL-connection, then an exception condition is raised: *connection exception— SQL-client unable to establish SQL-connection*.

12) If the SQL-server rejects the establishment of the SQL-connection, then an exception condition is raised: *connection exception— SQL-server rejected establishment of the SQL- connection*.

13) The SQL-server for the subsequent execution of SQL-statements via CLI routine invocations is set to the SQL-server identified by *SN*.

Working Draft

14) The SQL-session <authorization identifier> is set to *UN*.

5.5.9 DescribeCol

Function

Get column attributes.

Definition

```
DescribeCol (
       StatementHandle IN   INTEGER,
       ColumnNumber    IN   SMALLINT,
       ColumnName      OUT  CHARACTER(L),
       BufferLength    IN   SMALLINT,
       NameLength      OUT  SMALLINT,
       DataType        OUT  SMALLINT,
       LengthPrecision OUT  INTEGER,
       Scale           OUT  SMALLINT,
       Nullable        OUT  SMALLINT )
       RETURNS SMALLINT
```

where *L* is the value of BufferLength and has a maximum value of 128.

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no select source associated with *S*, then an exception condition is raised: *CLI-specific condition—function sequence error*. Otherwise, let *SS* be the select source associated with *S*.

3) Let *T* be the table specified by *SS*.

4) Let *N* be the value of ColumnNumber. If *N* is less than 1 or greater than the degree of *T*, then an exception condition is raised: *CLI-specific condition—invalid column number*.

5) Let *BL* be the value of BufferLength. If *BL* is not greater than zero, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

6) Let *C* be the *N*-th column of *T*. Information is retrieved about *C*.

   a) Let *CN* be the name of *C* and let *L* be the length of *CN*.

      Case:

      i) If *L* is not greater than *BL*, then ColumnName is set to *CN* and NameLength is set to *L*.

      ii) If *L* is greater than *BL*, then ColumnName is set to the first *BL* characters of *CN*, NameLength is set to *L* and a completion condition is raised: *string data, right truncation*.

   b) DataType is set to a code, as shown in Table 2, "Codes used for SQL data types in CLI", indicating the data type of *C*.

Working Draft

c) Case:

i) If the data type of *C* is character string, then LengthPrecision is set to the length or maximum length of *C*.

ii) If the data type of *C* is numeric, then LengthPrecision is set to the precision of *C*.

iii) If the data type of *C* is exact numeric, then Scale is set to the scale of *C*.

d) If *C* is possibly nullable, then Nullable is set to 1; otherwise Nullable is set to zero.

5.5.10 Disconnect

Function

Terminate an established connection.

Definition

```
Disconnect (
       ConnectionHandle    IN   INTEGER )
       RETURNS SMALLINT
```

Description

1) Case:

a) If ConnectionHandle does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition—invalid handle*.

b) Otherwise:

i) Let *C* be the allocated SQL-connection identified by ConnectionHandle.

ii) Any status records associated with *C* are destroyed.

2) Case:

a) If there is no established SQL-connection associated with *C*, then an exception condition is raised: *connection exception—connection does not exist*.

b) Otherwise, let *EC* be the established SQL-connection associated with *C*.

3) If *EC* is active, then an exception condition is raised: *invalid transaction state*.

4) Let *CC* be the current SQL-connection.

5) The SQL-session associated with *EC* is terminated. *EC* is terminated, regardless of any exception conditions that might occur during the disconnection process, and is no longer associated with *C*.

6) If any error is detected during the connection process, then a completion condition is raised: *warning—disconnect error*.

Working Draft

7) If *EC* and *CC* were the same SQL-connection, then there is no current SQL-connection. Otherwise, *CC* remains the current SQL-connection.

5.5.11 Error

Function

Return status information.

Definition

```
Error (
       EnvironmentHandle   IN   INTEGER,
       ConnectionHandle    IN   INTEGER,
       StatementHandle     IN   INTEGER,
       Sqlstate            OUT  CHARACTER(5),
       NativeError         OUT  INTEGER,
       MessageText         OUT  CHARACTER(L),
       BufferLength        IN   SMALLINT,
       TextLength          OUT  SMALLINT )
       RETURNS SMALLINT
```

where *L* is the value of BufferLength and has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

Description

1) Case:

   a) If StatementHandle identifies an allocated SQL-statement, then let *IH* be the value of StatementHandle.

   b) If StatementHandle is zero and ConnectionHandle identifies an allocated SQL-connection, then let *IH* be the value of ConnectionHandle.

   c) If ConnectionHandle is zero and EnvironmentHandle identifies an allocated SQL-environment, then let *IH* be the value of EnvironmentHandle.

   d) Otherwise, an exception condition is raised: *CLI-specific condition—invalid handle*.

2) Let *R* be the most recently executed CLI routine, other than Error, for which *IH* was passed as the value of an input handle.

   Note: If *IH* is an environment handle, then a Transact routine qualifies as *R* if, in addition, the value of its input connection handle was zero.

   Note: The Error routine may cause exception or completion conditions to be raised, but it does not cause status records to be generated.

3) Let *N* be the number of still existing status records generated by the execution of *R*. If *N* is zero, then a completion condition is raised: *no data*, and no further rules of this Subclause are applied. Sqlstate is set to '00000' and the values of NativeError, MessageText, and TextLength are implementation-dependent.

4) Let *BL* be the value of BufferLength. If *BL* is not greater than zero, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

Working Draft

5) Let *SR* be the first still-existing status record generated by the execution of *R*. Information is retrieved from *SR* as follows:

   a) Sqlstate is set to the SQLSTATE value.

   b) NativeError is set to the value of the native error code.

   c) Let *TL* be the length of the message text.

      Case:

      i) If *TL* is not greater than *BL*, then MessageText is set to the message text and TextLength is set to *TL*.

      ii) If *TL* is greater than *BL*, then MessageText is set to the first *BL* characters of the message text and TextLength is set to *TL*.

6) *SR* is destroyed.

5.5.12 ExecDirect

Function

Execute a statement directly.

Definition

```
ExecDirect (
       StatementHandle  IN   INTEGER,
       StatementText    IN   CHARACTER(L),
       TextLength       IN   SMALLINT)
       RETURNS SMALLINT
```

where *L* is determined by the value of TextLength and has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is a fetched row associated with *S*, then an exception condition is raised: *CLI-specific condition—invalid cursor state*.

3) Let *TL* be the value of TextLength.

   Case:

   a) If *TL* is greater than zero, then let *L* be *TL*.

   b) If *TL* is -3, then let *L* be the number of characters of StatementText that precede the implementation-defined null character that terminates a C character string. If StatementText does not contain the implementation-defined null character that terminates a C character string, then an exception condition is raised: *data exception—unterminated C string*.

Working Draft

c) Otherwise, an exception condition is raised: *CLI-specific condition—invalid argument value*.

4) Let *P* be the first *L* characters of StatementText.

5) If *P* is a <preparable dynamic delete statement: positioned> or a <preparable dynamic update statement: positioned>, then let *CN* be the name of the cursor referenced by *P*. Let *C* be the allocated SQL-connection with which *S* is associated. If *CN* is not the name of a cursor associated with another allocated SQL-statement associated with *C*, then an exception condition is raised: *invalid cursor name*.

6) If one or more of the following are true, then an exception condition is raised: *syntax error or access rule violation in dynamic SQL statement*.

a) *P* does not conform to the Format, Syntax Rules or Access Rules for a <preparable statement) or *P* is a <commit statement> or a <rollback statement>.

b) *P* contains a <simple comment>.

c) *P* contains a <dynamic parameter specification> in an invalid position as determined by the rules specified in Subclause 17.6, "<prepare statement>".

7) The data type of any <dynamic parameter specification> contained in *P* is determined by the rules specified in Subclause 17.6, "<prepare statement>".

8) *P* is executed.

Case:

a) If *P* is a <dynamic select statement> or a <dynamic single row select statement>, then *P* becomes the select source associated with *S* and the General Rules of Subclause 5.4.2, "Implicit cursor", are applied to *P* and *S* as *SELECT SOURCE* and *ALLOCATED STATEMENT*, respectively.

b) Otherwise:

i) The General Rules of Subclause 5.4.3, "Implicit using clause", are applied to 'EXECUTE', *P*, and *S* as *TYPE*, *SOURCE*, and *ALLOCATED STATEMENT* respectively.

Case:

1) If *P* is a <preparable dynamic delete statement: positioned>, then all General Rules in Subclause 17.19, "<preparable dynamic delete statement: positioned>", apply to *P*.

2) If *P* is a <preparable dynamic update statement: positioned>, then all General Rules in Subclause 17.20, "<preparable dynamic update statement: positioned>", apply to *P*.

3) Otherwise, the results of the execution are the same as if the statement were contained in a <routine> and executed; these are described in <REFERENCE>(external_invocation\FULL).

9) If *P* executed successfully, then:

a) The following objects associated with *S* are destroyed:

i) Any prepared statement.

Working Draft

ii) Any cursor.

iii) Any select source.

b) If a cursor associated with *S* is destroyed, then so are any prepared statements that reference that cursor.

c) Any executed statement associatd with *S* is destroyed and *P* becomes the directly executed statement associated with *S*.

5.5.13 Execute

Function

Execute a prepared statement.

Definition

```
Execute (
       StatementHandle  IN   INTEGER )
       RETURNS SMALLINT
```

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no prepared statement associated with *S*, then an exception condition is raised: *CLI-specific condition—function sequence error*. Otherwise, let *P* be the statement that was prepared.

3) If there is a fetched row associated with *S*, then an exception condition is raised: *invalid cursor state*.

4) *P* is executed.

Case:

a) If *P* is a <dynamic select statement> or a <dynamic single row select statement>, then the General Rules of Subclause 5.4.2, "Implicit cursor", are applied to *P* and *S* as *SELECT SOURCE* and *ALLOCATED STATEMENT*, respectively.

b) Otherwise:

i) The General Rules of Subclause 5.4.3, "Implicit using clause", are applied to 'EXECUTE', *P*, and *S* as *TYPE*, *SOURCE*, and *ALLOCATED STATEMENT*, respectively.

ii) Case:

1) If *P* is a <preparable dynamic delete statement: positioned>, then all General Rules in Subclause 17.19, "<preparable dynamic delete statement: positioned>", apply to *P*.

2) If *P* is a <preparable dynamic update statement: positioned>, then all General Rules in Subclause 17.20, "<preparable dynamic update statement: positioned>", apply to *P*.

Working Draft

3) Otherwise, the results of the execution are the same as if the statement were contained in a <routine> and executed; these are described in <REFERENCE>(external_invocation\FULL).

5) If *P* executed successfully, then any executed statement associated with *S* is destroyed and *P* becomes the indirectly executed statement associated with *S*.

5.5.14 Fetch

Function

Fetch the next row of a cursor.

Definition

```
Fetch (
       StatementHandle  IN   INTEGER )
       RETURNS SMALLINT
```

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no executed statement associated with *S*, then an exception condition is raised: *CLI-specific condition—function sequence error*.

3) If there is no open cursor associated with *S*, then an exception condition is raised: *invalid cursor state*. Otherwise, let *CR* be the open cursor associated with *S* and let *T* be the table associated with the open cursor.

4) If *T* is empty or if the position of *CR* is on or after the last row of *T*, then:

   a) *CR* is positioned after the last row of *T*.

   b) A completion condition is raised: *no data*, no database values are assigned to bound result targets, and no further rules of this Subclause are applied.

5) Case:

   a) If the position of *CR* is before a row *NR*, then *CR* is positioned on row *NR*.

   b) If the position of *CR* is on a row *OR* other than the last row, then *CR* is positioned on the row immediately after *OR*. Let *NR* be the row immediately after *OR*.

6) *NR* becomes the current row of *CR*.

7) If an exception condition is raised during derivation of any <derived column> associated with *NR*, then there is no fetched row associated with *S*, but *NR* remains the current row of *CR*. Otherwise, *NR* becomes the fetched row associated with *S*.

8) Let *SS* be the select source associated with *S*. The General Rules of Subclause 5.4.3, "Implicit using clause", are applied with 'FETCH', *SS*, and *S* as *TYPE*, *SOURCE*, and *ALLOCATED STATEMENT*, respectively.

Working Draft

9) If an exception condition occurs during the derivation of any target value, then the values of all the bound result targets are implementation-dependent and *CR* remains positioned on the current row.

5.5.15 FreeConnect

Function

Deallocate an SQL-connection.

Definition

```
FreeConnect (
       ConnectionHandle    IN   INTEGER )
       RETURNS SMALLINT
```

Description

1) Case:

   a) If ConnectionHandle does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition—invalid handle*.

   b) Otherwise:

      i) Let *C* be the allocated SQL-connection identified by ConnectionHandle.

      ii) Any status records associated with *C* are destroyed.

   c) If an established SQL-connection is associated with *C*, then an exception condition is raised: *CLI-specific condition—function sequence error*.

   d) *C* is deallocated and all its resources are freed.

5.5.16 FreeEnv

Function

Deallocate an SQL-environment.

Definition

```
FreeEnv (
       EnvironmentHandle   IN   INTEGER )
       RETURNS SMALLINT
```

Working Draft

Description

1) Case:

   a) If EnvironmentHandle does not identify an allocated SQL-environment, then an exception condition is raised: *CLI-specific condition—invalid handle*.

   b) Otherwise:

      i) Let *E* be the allocated SQL-environment identified by EnvironmentHandle.

      ii) Any status records associated with *E* are destroyed.

2) If an allocated SQL-connection is associated with *E*, then an exception condition is raised: *CLI-specific condition—function sequence error*.

3) *E* is deallocated and all its resources are freed.

5.5.17 FreeStmt

Function

Deallocate an SQL-statement.

Definition

```
FreeStmt (
       StatementHandle  IN   INTEGER,
       Option           IN   SMALLINT )
       RETURNS SMALLINT
```

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If Option is less than 0 or greater than 3, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

3) If Option = 0 or 1 and there is an open cursor associated with *S*, then:

   a) The open cursor associated with *S* is placed in the closed state and its copy of the select source is destroyed.

   b) Any fetched row associated with *S* is destroyed.

   c) If there is a directly executed statement associated with *S*, then the select source associated with *S* is destroyed.

   d) The executed statement associated with *S* is destroyed.

4) If Option = 1, then *S* is deallocated and all its resources are freed.

5) If Option = 2, then all result targets that are currently bound for *S* are unbound.

Working Draft

6) If Option = 3, then all <dynamic parameter specification> value sources that are currently bound for *S* are unbound and all <dynamic parameter specification> values that are currently set for *S* are unset.

5.5.18 GetCol

Function

Retrieve a column value.

Definition

```
GetCol (
       StatementHandle  IN   INTEGER,
       ColumnNumber     IN   SMALLINT,
       TargetType       IN   SMALLINT,
       TargetValue      OUT  POINTER,
       BufferLength     IN   INTEGER,
       IndicatorValue   OUT  INTEGER )
       RETURNS SMALLINT
```

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no fetched row associated with *S*, then an exception condition is raised: *CLI-specific condition—function sequence error*. Otherwise, let *FR* be the fetched row associated with *S* and let *T* be the table associated with the open cursor associated with *S*.

3) Let *CN* be the value of ColumnNumber. If *CN* is less than 1 or greater than the degree of *T*, then an exception condition is raised: *CLI-specific condition—invalid column number*.

4) If there is at least one result target currently bound for *S*, then let *BCN* be the column number associated with a bound result target and let *HBCN* be the value of MAX(*BCN*). Otherwise, let *HBCN* be zero.

5) If *CN* is not greater than *HBCN*, then an exception condition is raised: *CLI-specific condition—invalid column number*.

6) If there is a fetched column number associated with *FR*, then let *FCN* be that column number; otherwise, let *FCN* be zero.

7) Case:

   a) If *FCN* is greater than zero and *CN* is not greater than *FCN*, then an exception condition is raised: *CLI-specific condition—invalid column number*.

   b) If *FCN* is less than zero, then:

      i) Let *AFCN* be the absolute value of *FCN*.

      ii) Case:

         1) If *CN* is less than *AFCN*, then an exception condition is raised: *CLI-specific condition—invalid column number*.

Working Draft

2) If *CN* is greater than *AFCN*, then let *FCN* = *AFCN*.

8) Let *TT* be the value of TargetType.

9) Let *HL* be the standard programming language of the invoking host program. Let ''operative data type correspondence table'' be the data type correspondence table for *HL* as specified in <REFERENCE>(typecorr\FULL). Refer to the two columns of the operative data type correspondence table as the ''SQL data type column'' and the ''host data type column''.

10) If any of the following are true, then an exception condition is raised: *CLI-specific condition—program type out of range*.

a) *TT* is not greater than zero.

b) *TT* is greater than zero but is neither 99 nor one of the code values in Table 2, "Codes used for SQL data types in CLI".

c) *TT* is one of the code values in Table 2, "Codes used for SQL data types in CLI", but the row that contains the corresponding data type in the SQL data type column of the operative data type correspondence table contains 'None' in the host data type column.

11) Let *BL* be the value of BufferLength. If *TT* indicates CHARACTER and BL is not greater than zero, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

12) Let *CV* and *CDT* be the value and data type, respectively, of column CN of *FR*.

13) Case:

a) If *FCN* is less than zero, then:

Case:

i) If *TT* does not indicate CHARACTER, then an exception condition is raised: *data exception—error in assignment*.

ii) Otherwise, let *FL* be the fetched length associated with *FCN* and let SV and *SDT* be the value and data type, respectively, of the result of the <string value function>:

SUBSTRING(*CV* FROM (*FL*+1))

Otherwise, let *FL* be zero, let *SV* be *CV* and let *SDT* be *CDT*.

14) Case:

a) If *TT* is 99, then let *TV* = *CV*.

b) If *TT* indicates CHARACTER, then:

i) If the <cast specification>

CAST (*SV* AS CHARACTER(*BL*))

violates the General Rules of Subclause 6.10, "<cast specification>", then an exception condition is raised in accordance with the General Rules of Subclause 6.10, "<cast specification>".

ii) The <cast specification>

CAST (*SV* AS CHARACTER(*L*))

Working Draft

is effectively performed, and is the target value *TV*.

c) If *TT* indicates the same data type as *SDT*, then let *TV* = *SV*.

d) Otherwise, it is implementation-defined whether *SV* can be converted to the data type indicated by *TT*.

Case:

i) If *SV* cannot be converted to the data type indicated by *TT*, then an exception condition is raised: *dynamic SQL error—restricted data type attribute violation*.

ii) Otherwise, *TV* is derived from *SV* according to implementation-defined rules.

15) *CN* becomes the fetched column number associated with the fetched row associated with *S*.

16) If *TV* is a null value, then IndicatorValue is set to the negative number associated with that null value.

17) If *TV* is not a null value, then:

a) TargetValue is set to *TV*.

b) Case:

i) If *SDT* is CHARACTER or CHARACTER VARYING, then IndicatorValue is set to the length of *SV*.

ii) Otherwise, IndicatorValue is set to zero.

c) If *TT* indicates CHARACTER and *SDT* is CHARACTER or CHARACTER VARYING, then:

i) Let *LCV* be the length of *CV*.

ii) Let *FL* = $FL + BL$.

iii) If *FL* is less than *LCV*, then $FL - CN$ becomes the fetched column number associated with the fetched row associated with *S* and *FL* becomes the fetched length associated with the fetched column number.

5.5.19 GetCursorName

Function

Get a cursor name.

Definition

```
GetCursorName (
      StatementHandle  IN   INTEGER,
      CursorName       OUT  CHARACTER(L),
      BufferLength     IN   SMALLINT,
      NameLength       OUT  SMALLINT )
      RETURNS SMALLINT
```

where *L* is the value of BufferLength and has a maximum value of 128.

Working Draft

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no open cursor associated with *S* and there is no cursor name associated with *S*, then an exception condition is raised: *CLI-specific condition—no cursor name available*.

3) Case:

   a) If there is an open cursor associated with *S*, then let *CN* be the name of the open cursor.

   b) Otherwise, let *CN* be the cursor name associated with *S*.

4) Let *L* be the length of *CN* and let *BL* be the value of BufferLength. If *BL* is not greater than zero, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

5) Case:

   a) If *L* is not greater than *BL*, then CursorName is set to *CN* and NameLength is set to *L*.

   b) If *L* is greater than *BL*, then CursorName is set to the first *BL* characters of *CN*, NameLength is set to *L* and a completion condition is raised: *string data, right truncation*.

5.5.20 NumResultCols

Function

Get the number of result columns.

Definition

```
NumResultCols (
       StatementHandle  IN   INTEGER,
       ColumnCount      OUT  SMALLINT )
       RETURNS SMALLINT
```

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no prepared or directly executed statement associated with *S*, then an exception condition is raised: *CLI-specific condition—function sequence error*.

3) Case:

   a) If there is a select source associated with *S*, then let *D* be the degree of the table specified by the select source.

   b) Otherwise, let *D* be zero.

4) ColumnCount is set to *D*.

Working Draft

5.5.21 Prepare

Function

Prepare a statement.

Definition

```
Prepare (
       StatementHandle  IN   INTEGER,
       StatementText    IN   CHARACTER(L),
       TextLength       IN   SMALLINT )
       RETURNS SMALLINT
```

where *L* is determined by the value of TextLength and has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If an open cursor is associated with *S*, then an exception condition is raised: *invalid cursor state*.

3) Let *TL* be the value of TextLength.

   Case:

   a) If *TL* is greater than zero, then let *L* be *TL*.

   b) If *TL* is -3, then let *L* be the number of characters of StatementText that precede the implementation-defined null character that terminates a C character string. If StatementText does not contain the implementation-defined null character that terminates a C character string, then an exception condition is raised: *data exception—unterminated C string*.

   c) Otherwise, an exception condition is raised: *CLI-specific condition—invalid argument value*.

4) Let *P* be the first *L* characters of StatementText.

5) If *P* is a <preparable dynamic delete statement: positioned> or a <preparable dynamic update statement: positioned>, then let *CN* be the name of the cursor referenced by *P*. Let *C* be the allocated SQL-connection with which *S* is associated. If *CN* is not the same as the name of a cursor associated with another allocated SQL-statement associated with *C*, then an exception condition is raised: *invalid cursor name*.

6) If one or more of the following are true, then an exception condition is raised: *syntax error or access rule violation in dynamic SQL statement*.

   a) *P* does not conform to the Format, Syntax Rules, or Access Rules for a <preparable statement> or *P* is a <commit statement> or a <rollback statement>.

   b) *P* contains a <simple comment>.

Working Draft

c) *P* contains a <dynamic parameter specification> in an invalid position as determined by the rules specified in Subclause 17.6, "<prepare statement>".

7) The data type of any <dynamic parameter specification> contained in *P* is determined by the rules specified in Subclause 17.6, "<prepare statement>".

8) The following objects associated with *S* are destroyed:

a) Any prepared statement.

b) Any cursor.

c) Any select source.

d) Any executed statement.

If a cursor associated with *S* is destroyed, then so are any prepared statements associated with that cursor.

9) *P* is prepared and the prepared statement is associated with *S*.

10) If *P* is a <dynamic select statement> or a <dynamic single row select statement>, then P becomes the select source associated with *S*.

11) The validity of a prepared statement in an SQL-transaction different from the one in which the statement was prepared is implementation-dependent.

5.5.22 RowCount

Function

Get the row count.

Definition

```
RowCount (
       StatementHandle  IN   INTEGER,
       RowCount         OUT  INTEGER )
       RETURNS SMALLINT;
```

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no executed statement associated with *S*, then an exception condition is raised: *CLI-specific condition—function sequence error*. Otherwise, let *ES* be the executed statement associated with *S*.

3) Case:

a) If *ES* is not a <delete statement: searched>, an <insert statement>, or an <update statement: searched>, then RowCount is set to an implementation-dependent value.

b) Otherwise, RowCount is set to the number of rows affected by the execution of *ES*. Let *T* be the table identified by the <table name> directly contained in *ES*.

Working Draft

Case:

i) If *ES* is an <insert statement>, then RowCount is set to the number of rows inserted into *T*.

ii) If *ES* is not an <insert statement> and does not contain a <search condition>, then RowCount is set to the cardinality of *T* before the execution of *ES*.

iii) Otherwise, let *SC* be the <search condition> directly contained in *ES*. RowCount is set to the value effectively derived by executing the statement:

SELECT COUNT(*) FROM *T* WHERE *SC*

before the execution of *ES*.

5.5.23 SetCursorName

Function

Set a cursor name.

Definition

```
SetCursorName (
       StatementHandle     IN   INTEGER,
       CursorName          IN   CHARACTER(L),
       NameLength          IN   SMALLINT )
       RETURNS SMALLINT
```

where *L* is determined by the value of NameLength and has a maximum value of 128.

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is a prepared statement associated with *S*, then an exception condition is raised: *CLI-specific condition—function sequence error*.

3) Let *NL* be the value of NameLength.

Case:

a) If *NL* is greater than zero, then let *L* be *NL*.

b) If *NL* is 3, then let *L* be the number of characters of CursorName that precede the implementation-defined null character that terminates a C character string. If CursorName does not contain the implementation-defined null character that terminates a C character string, then an exception condition is raised: *data exception—unterminated C string*.

c) Otherwise, an exception condition is raised: *CLI-specific condition—invalid argument value*.

4) Let *CV* be the first *L* characters of CursorName and let *CN* be the value of

TRIM (BOTH ' ' FROM *CV*)

Working Draft

5) If *CN* does not conform to the Format and Syntax Rules of a <regular identifier>, then an exception condition is raised: *invalid cursor name*.

6) Let *C* be the allocated SQL-connection with which *S* is associated and let *SC* be the <search condition>:

   CN LIKE 'SQLCUR%'

   If *SC* is true or if *CN* is identical to the value of any cursor name associated with an allocated SQL-statement associated with *C*, then an exception condition is raised: *invalid cursor name*.

7) *CN* becomes the cursor name associated with *S*.

5.5.24 SetParamValue

Function

Set the value for a <dynamic parameter specification>.

Definition

```
SetParamValue (
       StatementHandle  IN   INTEGER,
       ParameterNumber  IN   SMALLINT,
       ValueType        IN   SMALLINT,
       SqlType          IN   SMALLINT,
       LengthPrecision  IN   INTEGER,
       Scale            IN   SMALLINT,
       ParameterValue   IN   POINTER,
       IndicatorValue   IN   INTEGER )
       RETURNS SMALLINT
```

Description

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) Let *PN* be the value of ParameterNumber. If *PN* is less than 1, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

3) Let *VT* be the value of ValueType.

4) Let *HL* be the standard programming language of the invoking host program. Let ''operative data type correspondence table'' be the data type correspondence table for *HL* as specified in <REFERENCE>(typecorrFULL). Refer to the two columns of the operative data type correspondence table as the ''SQL data type column'' and the ''host data type column''.

5) If any of the following are true, then an exception condition is raised: *CLI-specific condition—program type out of range*.

   a) *VT* is not greater than zero.

   b) *VT* is greater than zero but is neither 99 nor one of the code values in Table 3, "Codes used for buffer data types".

Working Draft

c) *VT* is one of the code values in Table 3, "Codes used for buffer data types", but the row that contains the corresponding data type in the SQL data type column of the operative data type correspondence table contains 'None' in the host data type column.

6) Let *ST* be the value of SqlType. If *ST* is not one of the code values in Table 2, "Codes used for SQL data types in CLI", then an exception condition is raised: *CLI-specific condition—SQL data type out of range*.

7) Case:

a) If *ST* indicates a character string type, then LengthPrecision is the length associated with *ST*.

b) If *ST* indicates NUMERIC or DECIMAL or FLOAT, then LengthPrecision is the precision associated with *ST*.

c) If *ST* indicates NUMERIC or DECIMAL, then Scale is the scale associated with *ST*.

8) Let *SV* and *IV* be the values of ParameterValue and IndicatorValue respectively.

9) If *VT* indicates CHARACTER, then:

a) Case:

i) If *IV* is greater than zero, then let *L* be *IV*.

ii) If *IV* is -3, then let *L* be the number of characters of *SV* that precede the implementation-defined null character that terminates a C character string. If *SV* does not contain the implementation-defined null character that terminates a C character string, then an exception condition is raised: *data exception—unterminated C string*.

iii) If *IV* is less than 1 but is neither -1 nor -3, then an exception condition is raised: *CLI-specific condition—invalid argument value*.

b) *L* is the length of *SV*.

10) Case:

a) If *IV* is -1, then let *SPV* be the general null value.

b) If *VT* is 99, then let *SPV* be *SV*.

c) Otherwise:

i) *VT* indicates the data type of *SV*.

ii) Let *TDT* be the target data type indicated by *ST* and its associated length, precision, and scale, as applicable.

iii) If the <cast specification>

CAST (*SV* AS *TDT*)

violates the Syntax Rules of Subclause 6.10, "<cast specification>", then an exception condition is raised: *dynamic SQL error—restricted data type attribute violation*.

iv) If the <cast specification>

CAST (*SV* AS *TDT*)

Working Draft

violates the General Rules of Subclause 6.10, "<cast specification>", then an exception condition is raised in accordance with the General Rules of Subclause 6.10, "<cast specification>".

v) The <cast specification>

CAST (*SV* AS *TDT*)

is effectively performed, and is the value of *SPV*. LE>Case:

a) If the parameter number associated with any <dynamic parameter specification> value source currently bound for *S* is *PN*, then that <dynamic parameter specification> value source is unbound.

b) If the parameter number associated with any <dynamic parameter specification> value currently set for *S* is *PN*, then that <dynamic parameter specification> value is unset.

11) *SPV* becomes the <dynamic parameter specification> value set for parameter *PN* for *S*.

5.5.25 Transact

Function

Terminate an SQL-transaction.

Definition

```
Transact (
       EnvironmentHandle    IN    INTEGER,
       ConnectionHandle     IN    INTEGER,
       TerminationType      IN    SMALLINT )
       RETURNS SMALLINT
```

Description

Case:

a) If ConnectionHandle identifies an allocated SQL-connection, then:

i) Let *C* be the allocated SQL-connection identified by ConnectionHandle.

ii) Let *E* be the allocated SQL-environment with which *C* is associated.

iii) Let *MS* be 0.

iv) Any status records associated with *C* are destroyed.

b) If ConnectionHandle is zero and EnvironmentHandle identifies an allocated SQL-environment that is not a skeleton SQL- environment, then:

i) Let *E* be the allocated SQL-environment identified by EnvironmentHandle.

ii) Let *MS* be 1.

iii) Any status records associated with *E* are destroyed.

c) Otherwise, an exception condition is raised: *CLI-specific condition—invalid handle*.

Working Draft

1) Let *TT* be the value of TerminationType. If *TT* is neither zero nor 1, then an exception condition is raised: *CLI-specific condition—invalid transaction operation code*.

2) Let *L* be a list of allocated SQL-connections associated with *E*. Let *L1* be a list of the allocated SQL-connections in *L* that have an associated established SQL-connection that is active.

3) If *MS* = 0, then:

   a) If there is no established SQL-connection associated with *C*, then an exception condition is raised: *CLI-specific condition—function sequence error*. Otherwise, let *EC* be the established SQL-connection associated with *C*.

   b) If *EC* is not active, then no further rules of this Subclause are applied.

   c) If there is more than one allocated SQL-connection in *L1*, then it is implementation-defined whether or not an exception condition is raised.

   d) Let *L2* be a list of open cursors associated with allocated SQL-statements associated with *C*.

4) If *MS* = 1, then:

   a) If *L1* is empty, then no further rules of this Subclause are applied.

   b) Let *L2* be a list of open cursors associated with allocated SQL-statements associated with allocated SQL-connections in *L1*.

5) If the current SQL-transaction is part of an encompassing transaction that is controlled by an agent other than the SQL-agent, then an exception condition is raised: *invalid transaction termination*.

6) For every open cursor *OC* in *L2*:

   a) Let *SH* be the statement handle that identifies the allocated SQL-statement with which *OC* is associated.

   b) FreeStmt is implicitly invoked with *SH* as the value of StatementHandle and with zero as the value of Option.

7) If *TT* = 0, then:

   a) Case:

      i) If any error preventing commitment of the SQL-transaction has occurred, then any changes to SQL-data or schemas that were made by the current SQL-transaction are canceled and an exception condition is raised: *transaction rollback* with an implementation-defined subclass value.

      ii) Otherwise, any changes to SQL-data or schemas that were made by the current SQL-transaction are made accessible to all concurrent and subsequent SQL-transactions.

   b) The current SQL-transaction is terminated.

8) If *TT* = 1, then:

   a) Any changes to SQL-data or schemas that were made by the current SQL-transaction are canceled.

Working Draft

b) The current SQL-transaction is terminated.

Working Draft

Annex A
(Informative)

Typical header files

A.1 C Header File SQLCLI.H

```
/* SQLCLI.H  Header File for SQL CLI.
* The actual header file should contain at least the information specified
* here, except that the comments may vary.
*/

/* host language data types */
typedef unsigned char   SQLCHAR;
typedef long  SQLINTEGER;
typedef short SQLSMALLINT;
typedef double     SQLDOUBLE;
typedef float SQLREAL;
typedef void *     SQLPOINTER;

/* function return type */
typedef SQLSMALLINTSQLRETURN;

/* handles */
typedef SQLINTEGER SQLHENV;     /* environment handle */
typedef SQLINTEGER SQLHDBC;     /* connection handle */
typedef SQLINTEGER SQLHSTMT;    /* statement handle */

/* null value indication */
 define SQL_NULL_DATA     -1

/* return values from routines */
 define SQL_SUCCESS 0
 define SQL_SUCCESS_WITH_INFO 1
 define SQL_NO_DATA 100
 define SQL_ERROR   -1
 define SQL_INVALID_HANDLE    -2

/* flag for null-terminated string */
 define SQL_NTS     -3

/* maximum message length */
 define SQL_MAX_MESSAGE_LENGTH255

/* maximum identifier length */
 define SQL_MAX_IDENTIFIER_LENGTH  128
```

Working Draft

A.1 C Header File SQLCLI.H

```
/* codes for SQL data types */
 define SQL_CHAR    1
 define SQL_NUMERIC 2
 define SQL_DECIMAL 3
 define SQL_INTEGER 4
 define SQL_SMALLINT5
 define SQL_FLOAT   6
 define SQL_REAL    7
 define SQL_DOUBLE  8
 define SQL_VARCHAR 12

/* Application buffer types  */
 define SQLBUF_CHAR SQL_CHAR
 define SQLBUF_LONG SQL_INTEGER
 define SQLBUF_SHORTSQL_SMALLINT
 define SQLBUF_FLOATSQL_REAL
 define SQLBUF_DOUBLE    SQL_DOUBLE
 define SQLBUF_DEFAULT   99

/* DescribeCol description of NULLABLE attributes */
 define SQL_NULLABLE1
 define SQL_NO_NULLS0

/* FreeStmt options */
 define SQL_CLOSE   0
 define SQL_DROP    1
 define SQL_UNBIND  2
 define SQL_RESET_PARAMS 3

/* Transact options */
 define SQL_COMMIT  0
 define SQL_ROLLBACK1

/* Null handles for Error */
 define SQL_NULL_HENV    0
 define SQL_NULL_HDBC    0
 define SQL_NULL_HSTMT   0

/* ColAttribute options */
 define SQL_COLUMN_COUNT1
 define SQL_COLUMN_NAME 2
 define SQL_COLUMN_TYPE 3
 define SQL_COLUMN_LENGTH    4
 define SQL_COLUMN_PRECISION 5
 define SQL_COLUMN_SCALE 6
 define SQL_COLUMN_NULLABLE  7
 define SQL_COLUMN_UNNAMED   8

/* by-value prototypes */

SQLRETURN  SQLAllocConnect(SQLHENV henv, SQLHDBC * hdbc);
SQLRETURN  SQLAllocEnv(SQLHENV * henv);
SQLRETURN  SQLAllocStmt(SQLHDBC hdbc, SQLHSTMT * hstmt);
SQLRETURN  SQLBindCol(SQLHSTMT hstmt, SQLSMALLINT icol,
      SQLSMALLINT fBufType, SQLPOINTER rgbValue,
      SQLINTEGER cbValueMax, SQLINTEGER * cbValue);
SQLRETURN  SQLBindParam(SQLHSTMT hstmt, SQLSMALLINT ipar,
      SQLSMALLINT fBufType, SQLSMALLINT fSQLType,
      SQLINTEGER cbParamDef, SQLSMALLINT ibScale,
      SQLPOINTER rgbValue, SQLINTEGER * cbValue);
SQLRETURN  SQLCancel(SQLHSTMT hstmt);
SQLRETURN  SQLColAttribute(SQLHSTMT hstmt, SQLSMALLINT icol,
```

Working Draft

```
      SQLSMALLINT fDescType, SQLCHAR * rgbDesc,
      SQLSMALLINT cbDescMax, SQLSMALLINT * cbDesc,
      SQLINTEGER * fDesc);
SQLRETURN  SQLConnect(SQLHDBC hdbc, SQLCHAR * szServer,
      SQLSMALLINT cbServer, SQLCHAR * szUID, SQLSMALLINT cbUID);
SQLRETURN  SQLDescribeCol(SQLHSTMT hstmt, SQLSMALLINT icol,
      SQLCHAR * szColName, SQLSMALLINT cbColNameMax,
      SQLSMALLINT * cbColName, SQLSMALLINT * fSQLType,
      SQLINTEGER * cbColDef, SQLSMALLINT * ibScale,
      SQLSMALLINT * fNullable);
SQLRETURN  SQLDisconnect(SQLHDBC hdbc);
SQLRETURN  SQLError(SQLHENV henv, SQLHDBC hdbc, SQLHSTMT hstmt,
      SQLCHAR * szSqlState, SQLINTEGER * fNativeError,
      SQLCHAR * szErrorMsg, SQLSMALLINT cbErrorMsgMax,
      SQLSMALLINT * cbErrorMsg);
SQLRETURN  SQLExecDirect(SQLHSTMT hstmt, SQLCHAR * szSqlStr,
      SQLSMALLINT cbSqlStr);
SQLRETURN  SQLExecute(SQLHSTMT hstmt);
SQLRETURN  SQLFetch(SQLHSTMT hstmt);
SQLRETURN  SQLFreeConnect(SQLHDBC hdbc);
SQLRETURN  SQLFreeEnv(SQLHENV henv);
SQLRETURN  SQLFreeStmt(SQLHSTMT hstmt, SQLSMALLINT fOption);
SQLRETURN  SQLGetCol(SQLHSTMT hstmt, SQLSMALLINT icol,
      SQLSMALLINT fBufType, SQLPOINTER rgbValue,
      SQLINTEGER cbValueMax, SQLINTEGER * cbValue);
SQLRETURN  SQLGetCursorName(SQLHSTMT hstmt, SQLCHAR * szCursor,
      SQLSMALLINT cbCursorMax, SQLSMALLINT * cbCursor);
SQLRETURN  SQLNumResultCols(SQLHSTMT hstmt, SQLSMALLINT * ccol);
SQLRETURN  SQLPrepare(SQLHSTMT hstmt, SQLCHAR * szSqlStr,
      SQLSMALLINT cbSqlStr);
SQLRETURN  SQLRowCount(SQLHSTMT hstmt, SQLINTEGER * crow);
SQLRETURN  SQLSetCursorName(SQLHSTMT hstmt, SQLCHAR * szCursor,
      SQLSMALLINT cbCursor);
SQLRETURN  SQLSetParamValue(SQLHSTMT hstmt, SQLSMALLINT ipar,
      SQLSMALLINT fBufType, SQLSMALLINT fSQLType,
      SQLINTEGER cbParamDef, SQLSMALLINT ibScale,
      SQLPOINTER rgbValue, SQLINTEGER * cbValue);
SQLRETURN  SQLTransact(SQLHENV henv, SQLHDBC hdbc, SQLSMALLINT fType);
```

A.2 COBOL Header File SQLCLI.CBH

Here is a typical SQLCLI.CBH file. COBOL applications include this prolog by containing the following statement:

```
COPY SQLCLI.CBH
```

The following file does not include prototypes of the CLI routines because COBOL applications are not required to specify them.

```
* NULL VALUE INDICATION
01 SQL-NULL-DATA   PIC S9(4) BINARY VALUE IS -1.

* RETURN VALUES FROM ROUTINES
01 SQL-SUCCESS     PIC S9(4) BINARY VALUE IS 0.
01 SQL-SUCCESS-WITH-INFO PIC S9(4) BINARY VALUE IS 1.
01 SQL-NO-DATA     PIC S9(4) BINARY VALUE IS 100.
01 SQL-ERROR  PIC S9(4) BINARY VALUE IS -1.
01 SQL-INVALID-HANDLE   PIC S9(4) BINARY VALUE IS -2.

* FLAG FOR NULL-TERMINATED STRING
```

Working Draft

```
01 SQL-NTS    PIC S9(9) BINARY VALUE IS -3.

* CODES FOR SQL DATA TYPES
01 SQL-CHAR   PIC S9(4) BINARY VALUE IS 1.
01 SQL-NUMERIC     PIC S9(4) BINARY VALUE IS 2.
01 SQL-DECIMAL     PIC S9(4) BINARY VALUE IS 3.
01 SQL-INTEGER     PIC S9(4) BINARY VALUE IS 4.
01 SQL-SMALLINT    PIC S9(4) BINARY VALUE IS 5.
01 SQL-FLOAT  PIC S9(4) BINARY VALUE IS 6.
01 SQL-REAL   PIC S9(4) BINARY VALUE IS 7.
01 SQL-DOUBLE PIC S9(4) BINARY VALUE IS 8.
01 SQL-VARCHAR     PIC S9(4) BINARY VALUE IS 12.

* APPLICATION BUFFER TYPES
01 SQLBUF-CHAR     PIC S9(4) BINARY VALUE IS 1.
01 SQLBUF-INTEGER  PIC S9(4) BINARY VALUE IS 4.
01 SQLBUF-SMALLINT PIC S9(4) BINARY VALUE IS 5.
01 SQLBUF-DEFAULT  PIC S9(4) BINARY VALUE IS 99.

* DESCRIBECOL DESCRIPTION OF NULLABLE ATTRIBUTE
01 SQL-NO-NULLS    PIC S9(4) BINARY VALUE IS 0.
01 SQL-NULLABLE    PIC S9(4) BINARY VALUE IS 1.

* FREESTMT OPTIONS
01 SQL-CLOSE  PIC S9(4) BINARY VALUE IS 0.
01 SQL-DROP   PIC S9(4) BINARY VALUE IS 1.
01 SQL-UNBIND PIC S9(4) BINARY VALUE IS 2.
01 SQL-RESET-PARAMSPIC S9(4) BINARY VALUE IS 3.

* TRANSACT OPTIONS
01 SQL-COMMIT PIC S9(4) BINARY VALUE IS 0.
01 SQL-ROLLBACK    PIC S9(4) BINARY VALUE IS 1.

* NULL HANDLES FOR ERROR
01 SQL-NULL-HENV   PIC S9(9) BINARY VALUE IS 0.
01 SQL-NULL-HDBC   PIC S9(9) BINARY VALUE IS 0.
01 SQL-NULL-HSTMT  PIC S9(9) BINARY VALUE IS 0.

* COLATTRIBUTE OPTION
01 SQL-COLUMN-COUNT PIC S9(4) BINARY VALUE IS 1.
01 SQL-COLUMN-NAME PIC S9(4) BINARY VALUE IS 2.
01 SQL-COLUMN-TYPE PIC S9(4) BINARY VALUE IS 3.
01 SQL-COLUMN-LENGTH    PIC S9(4) BINARY VALUE IS 4.
01 SQL-COLUMN-PRECISION PIC S9(4) BINARY VALUE IS 5.
01 SQL-COLUMN-SCALEPIC S9(4) BINARY VALUE IS 6.
01 SQL-COLUMN-NULLABLE  PIC S9(4) BINARY VALUE IS 7.
01 SQL-COLUMN-UNNAMED   PIC S9(4) BINARY VALUE IS 8.
```

Working Draft

Annex B

Sample C programs

Here are two examples of using CLI.

The first example illustrates creating a table, adding some data to it, and selecting the inserted data. Both methods of SQL specification and execution are shown (that is, Prepare( ), Execute( ) and ExecDirect( )). Embedded SQL has been included in this example for illustrative purposes.

The second example shows interactive ad hoc query processing.

Actual applications include more complete error checking following calls to CLI routines. That material is omitted from this Appendix for the sake of clarity.

B.1 Create Table, Insert, Select

```
 include "sqlcli.h"
 include <string.h>

 ifndef NULL
 define NULL   0
 endif

int print_err(SQLHDBC hdbc, SQLHSTMT hstmt);

int example1(SQLCHAR * server, SQLCHAR * uid)
{
SQLHENV henv;
SQLHDBC hdbc;
SQLHSTMT hstmt;
SQLINTEGER id;
SQLCHAR name[50];
SQLINTEGER namelen;
SQLSMALLINT scale;

scale = 0;
/* EXEC SQL CONNECT TO :server USER :uid; */
SQLAllocEnv(&henv); /* allocate an environment handle */

SQLAllocConnect(henv, &hdbc);  /* allocate a connection handle */

/* connect to database */
if (SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS)
     != SQL_SUCCESS)
    return(print_err(hdbc, SQL_NULL_HSTMT));
```

Working Draft

B.1 Create Table, Insert, Select

```
SQLAllocStmt(hdbc, &hstmt);  /* allocate a statement handle */

/* EXEC SQL CREATE TABLE NAMEID (ID integer, NAME varchar(50));  */
{
     SQLCHAR create[] ="CREATE TABLE NAMEID (ID integer, NAME varchar(50))";

     /* execute the SQL statement */
     if (SQLExecDirect(hstmt, create, SQL_NTS) != SQL_SUCCESS)
    return(print_err(hdbc, hstmt));
}
/* EXEC SQL COMMIT WORK; */
SQLTransact(henv, hdbc, SQL_COMMIT);  /* commit create table */

/* EXEC SQL INSERT INTO NAMEID VALUES ( :id, :name ); */
{
     SQLCHAR insert[] = "INSERT INTO NAMEID VALUES (?, ?)";

/* show the use of SQLPrepare/SQLExecute method */
/* prepare the insert */
     if (SQLPrepare(hstmt, insert, SQL_NTS) != SQL_SUCCESS)
    return(print_err(hdbc, hstmt));
     SQLBindParam(hstmt, 1, SQLBUF_LONG, SQL_INTEGER,
    (SQLINTEGER)sizeof(SQLINTEGER), scale, (SQLPOINTER)&id,
    (SQLINTEGER *)NULL);
     SQLBindParam(hstmt, 2, SQLBUF_CHAR, SQL_VARCHAR,
    (SQLINTEGER)sizeof(name), scale, (SQLPOINTER)name,
    (SQLINTEGER *)NULL);

     /* now assign parameter values and execute the insert */
     id = 500;
     (void)strcpy((char *)name, "Cabbage");
     if (SQLExecute(hstmt) != SQL_SUCCESS)
    return(print_err(hdbc, hstmt));
}
/* EXEC SQL COMMIT WORK; */
SQLTransact(henv, hdbc, SQL_COMMIT);  /* commit insert */

/* EXEC SQL DECLARE c1 CURSOR FOR SELECT ID, NAME FROM NAMEID; */
/* EXEC SQL OPEN c1; */
/* The application doesn't specify "declare c1 cursor for" */
{
     SQLCHAR select[] = "select ID, NAME from NAMEID";
     if (SQLExecDirect(hstmt, select, SQL_NTS) != SQL_SUCCESS)
    return(print_err(hdbc, hstmt));
}
/* EXEC SQL FETCH c1 INTO :id, :name; */
/* use column binding with SQLBindCol */
SQLBindCol(hstmt, 1, SQLBUF_LONG, (SQLPOINTER)&id,
     (SQLINTEGER)sizeof(SQLINTEGER), (SQLINTEGER *)NULL);
SQLBindCol(hstmt, 2, SQLBUF_CHAR, (SQLPOINTER)name,
     (SQLINTEGER)sizeof(name), (SQLINTEGER *)&namelen);

SQLFetch(hstmt);  /* now execute the fetch */

/* finally, we should commit, discard hstmt, disconnect */
/* EXEC SQL COMMIT WORK; */
SQLTransact(henv, hdbc, SQL_COMMIT);    /* commit the transaction  */

/* EXEC SQL CLOSE c1; */
SQLFreeStmt(hstmt, SQL_DROP); /* free the statement handle */

/* EXEC SQL DISCONNECT; */
```

Working Draft

```
SQLDisconnect(hdbc);/* disconnect from the database */

SQLFreeConnect(hdbc);    /* free the connection handle */
SQLFreeEnv(henv);   /* free the environment handle  */

return(0);
}
```

B.2 Interactive Query

```
/*
* sample program to get the general idea for interactively executing
* an ad hoc statement.
*/

 include "sqlcli.h"
 include <string.h>
 include <stdlib.h>

 define  MAXCOLS   100

 define  max(a,b) (a>b?a:b)
int print_err(SQLHDBC hdbc, SQLHSTMT hstmt);
int build_indicator_message(SQLCHAR * errmsg, SQLINTER * data,
        SQLINTEGER collen, SQLINTEGER *outlen,
        SQLSMALLINT colnum);
SQLINTEGER display_length(SQLSMALLINT coltype, SQLINTEGER collen,
      SQLCHAR *colname);

example2(SQLCHAR * server, SQLCHAR * uid, SQLCHAR * sqlstr)
{
int    i;
SQLHENVhenv;
SQLHDBChdbc;
SQLHSTMT    hstmt;
SQLCHARerrmsg[SQL_MAX_MESSAGE_LENGTH+1];
SQLCHARcolname[SQL_MAX_IDENTIFIER_LENGTH+1];
SQLSMALLINT coltype;
SQLSMALLINT colnamelen;
SQLSMALLINT nullable;
SQLINTEGER  collen[MAXCOLS];
SQLSMALLINT scale;
SQLINTEGER  outlen[MAXCOLS];
SQLCHAR *   data[MAXCOLS];
SQLSMALLINT nresultcols;
SQLINTEGER  rowCount;
SQLRETURN   rc;

SQLAllocEnv(&henv);  /* allocate an environment handle */

SQLAllocConnect(henv, &hdbc);  /* allocate a connection handle */

/* connect to database */
if (SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS)
     != SQL_SUCCESS )
  return( print_err(hdbc, SQL_NULL_HSTMT) );

/* allocate a statement handle */
```

Working Draft

```
SQLAllocStmt(hdbc, &hstmt);

/* execute the SQL statement */
if (SQLExecDirect(hstmt, sqlstr, SQL_NTS) != SQL_SUCCESS)
    return(print_err(hdbc,hstmt));

/* see what kind of statement it was */
SQLNumResultCols(hstmt, &nresultcols);
if (nresultcols == 0) {
    /* no result columns, so must be non-select */
    /* check rowcount */
    SQLRowCount(hstmt, &rowcount);
    if (rowcount > 0 ) {
   /* rowcount was affected, so must have been update, insert or delete */
   if (SQLTransact(henv, hdbc, SQL_COMMIT) == SQL_SUCCESS) {
  (void)printf("%ld rows affected0, rowcount);
  (void)printf("Operation successful0);
   }
   else
  (void)printf("Operation failed0);
}
    else {
   /* rowcount == 0, so assume it was not an update/delete and
      therefore a DDL, Grant/Revoke, or Commit/Rollback.
      Of course this isn't necessarily so, it could be that the
      where clause in the update/delete did not match any rows */
  if (SQLTransact(henv, hdbc, SQL_COMMIT) == SQL_SUCCESS)
      (void)printf("Operation successful0);
  else
      (void)printf("Operation failed0);
    }
}
else {  /* must have result rows */
    /* display column names */
    for (i=0; i<nresultcols; i++) {
   SQLDescribeCol(hstmt, i+1, colname, sizeof(colname),
  &colnamelen, &coltype, &collen[i], &scale, &nullable);
   /* assume there is a display_length function that computes
   correct length given the data type  */
   collen[i] = display_length(coltype, collen[i], colname);
   (void)printf("%*.*s ", collen[i], collen[i], colname);
   /* allocate memory to bind column */
   data[i] = (SQLCHAR *) malloc(collen[i]);
   /* bind columns to program vars, converting all types to CHAR */
   SQLBindCol(hstmt, i+1, SQLBUF_CHAR, data[i], collen[i], &outlen[i]);
    }
    /* display result rows */
   while ((rc=SQLFetch(hstmt)) != SQL_ERROR) {
   errmsg[0] = '';
   if (rc == SQL_SUCCESS_WITH_INFO) {
  for (i=0; i<nresultcols; i++) {
      if (outlen[i] == SQL_NULL_DATA || outlen[i] >= collen[i])
     build_indicator_message(errmsg,
     (SQLPOINTER *)&data[i], collen[i], &outlen[i], i);
  }
  (void)printf("%*.*s ", outlen[i], outlen[i], data[i]);
   } /* for all columns in this row  */
   (void)printf("126s", errmsg);  /* print any truncation messages */
    } /* while rows to fetch */
} /* else select statement */

/* free data buffers */
for (i=0; i<nresultcols; i++)  {
```

Working Draft

```
    (void)free(data[i]);
}

SQLFreeStmt(hstmt, SQL_DROP );  /* free statement handle */
SQLDisconnect(hdbc);  /* disconnect from database */
SQLFreeConnect(hdbc);  /* free connection handle */
SQLFreeEnv(henv);    /* free environment handle */

return(0);
}

 /************************************************************
The following functions are given for completeness, but are
not relevant for understanding the database processing nature of CLI
************************************************************/

 define MAX_NUM_PRECISION 15
/* define max length of char string representation of
number as: = max(precision) + leading sign + . + exp sign + max exp length
      =  15  + 1  + 1 + 1   + 2
      =  15 + 5
*/
 define MAX_NUM_STRING_SIZE (MAX_NUM_PRECISION + 5)

SQLINTEGER display_length(SQLSMALLINT coltype, SQLINTEGER collen,
      SQLCHAR *colname)
{
switch (coltype) {

  case SQL_VARCHAR:
  case SQL_CHAR:
 return(max(collen,strlen((char *)colname)));
 break;

  case SQL_FLOAT:
  case SQL_DOUBLE:
  case SQL_NUMERIC:
  case SQL_REAL:
  case SQL_DECIMAL:
 return(max(MAX_NUM_STRING_SIZE,strlen((char *)colname)));
 break;

  case SQL_INTEGER:
 return(max(10,strlen((char *)colname)));
 break;

  case SQL_SMALLINT:
 return(max(5,strlen((char *)colname)));
 break;

  default:
 (void)printf("Unknown datatype, %d0, coltype);
 return(0);
 break;
  }
}

int build_indicator_message(SQLCHAR * errmsg, SQLPOINTER * data,
        SQLINTEGER collen, SQLINTEGER *outlen,
        SQLSMALLINT colnum)
{
```

Working Draft

```
if (*outlen == SQL_NULL_DATA) {
    (void)strcpy((char *)data, "NULL");
    *outlen=4;
}
else {
    sprintf((char *)errmsg+strlen((char *)errmsg),
      "%d chars truncated, col %d0, *outlen-collen+1, colnum);
    *outlen=255;
    }
}
```

Working Draft

Working Draft