Technical Standard

Data Management:
SQL Call Level Interface (CLI)
[This page intentionally left blank]
X/Open CAE Specification

Data Management:
SQL Call Level Interface (CLI)

X/Open Company Ltd.
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Development of CLI</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Relation to Other X/Open Documents</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Conceptual Differences from Embedded SQL</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Feature Extensions over Embedded SQL</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Relation to Standards</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>Compliance Policy</td>
<td>6</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Language Binding</td>
<td>6</td>
</tr>
<tr>
<td>1.4.2</td>
<td>SQL Statement Text</td>
<td>6</td>
</tr>
<tr>
<td>1.4.3</td>
<td>Distributed Transaction Delimitation</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>Compliance Terminology</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Concepts and Conventions</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>Syntactic Characteristics</td>
<td>11</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Function Variants</td>
<td>11</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Short Function Names</td>
<td>11</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Case Sensitivity</td>
<td>14</td>
</tr>
<tr>
<td>2.1.4</td>
<td>COBOL Naming Variations</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Use of Data Types</td>
<td>15</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Generic Data Types for Function Parameters</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Data Types for C</td>
<td>16</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Data Types for COBOL</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Semantic Conventions</td>
<td>18</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Direction of Parameters</td>
<td>18</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Null Pointers</td>
<td>18</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Output Arguments in Non-success Cases</td>
<td>18</td>
</tr>
<tr>
<td>2.3.4</td>
<td>International Character Data</td>
<td>18</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Variable-length Input Parameters</td>
<td>19</td>
</tr>
<tr>
<td>2.3.6</td>
<td>Variable-length Output Parameters</td>
<td>19</td>
</tr>
<tr>
<td>2.3.7</td>
<td>Interpretation of Strings</td>
<td>21</td>
</tr>
<tr>
<td>2.3.8</td>
<td>Identifiers</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Database System</td>
<td>22</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Clients and Servers</td>
<td>22</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Metadata and Data</td>
<td>22</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Three-part Object Naming</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>Data Structures</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>Handle Overview</td>
<td>26</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Attributes</td>
<td>26</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Rationale</td>
<td>26</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Handle Type Identifier</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>Environment</td>
<td>28</td>
</tr>
</tbody>
</table>
### Chapter 4 Interface Overview

4.1 Overview of Control Flow ................................................................. 42
4.1.1 Basic Control Flow ........................................................................ 42
4.1.2 Example Control Flow for SQL Statement Processing ............... 43
4.2 Executing SQL Statements ............................................................... 44
4.2.1 Prepare() and Execute() ................................................................. 44
4.2.2 ExecDirect() .................................................................................. 45
4.3 Specifying Dynamic Arguments ...................................................... 46
4.3.1 Use of Implementation Parameter Descriptor.............................. 47
4.3.2 Specifying Parameter Values at Execute Time ............................. 48
4.4 Analysing a Prepared Statement ...................................................... 50
4.5 Fetching Data..................................................................................... 50
4.5.1 Updating Fetched Data .................................................................. 52
4.6 Metadata Functions .......................................................................... 53
4.6.1 Parameters of Metadata Functions ................................................ 54
4.7 Introspection ...................................................................................... 57
4.8 Data Conversion ................................................................................ 57
4.8.1 Specifying SQL Data Types ......................................................... 57
4.8.2 Specifying Application Buffer Types ............................................ 58
4.8.3 Permitted Combinations ............................................................... 60
4.8.4 Transfers and Conversions ............................................................. 61
4.9 Transaction Management ................................................................. 63
4.9.1 Explicit Transaction Demarcation ................................................ 64
4.9.2 Distributed Transaction Processing ............................................ 64

### Chapter 5 Diagnostics

5.1 Return Value ..................................................................................... 67
5.2 Detailed Diagnostic Information ....................................................... 69
5.2.1 Error Codes from the X/Open SQL Specification ....................... 70
5.2.2 CLI-specific Errors ....................................................................... 71
5.3 Other Information in the Diagnostics Data Structure ........................ 72
5.3.1 Type of SQL Statement ................................................................. 72
5.3.2 Row Count .................................................................................... 72
5.4 General Diagnostics .......................................................................... 73
5.4.1 Connection Errors........................................................................ 73
5.4.2 Change of Connection ................................................................. 73
5.4.3 State Transition Errors ................................................................. 73
## Contents

5.4.4 Transaction Errors ................................................................. 74
5.4.5 Inability to Access Handle ...................................................... 74
5.4.6 Cancellation ........................................................................... 75
5.4.7 String Lengths ....................................................................... 75
5.4.8 Null Pointers ......................................................................... 75
5.4.9 Descriptor Record Consistency ................................................. 75
5.4.10 Data Conversion and Assignment Diagnostics ...................... 76

Chapter 6 Concise CLI Functions ......................................................... 79
6.1 Inventory of Concise Functions ..................................................... 80
6.2 Definition of Concise Functions .................................................... 80
6.3 Overview of Concise Functions ..................................................... 81

Chapter 7 Deprecated CLI Functions ............................................... 83
7.1 Inventory of Deprecated Functions ............................................... 84
7.2 Definition of Deprecated Functions .............................................. 84
7.3 Overview of Deprecated Functions .............................................. 85

Chapter 8 Reference Manual Pages ................................................... 87
AllocConnect( ) ................................................................................. 88
AllocEnv( ) ....................................................................................... 90
AllocHandle( ) .................................................................................. 92
AllocStmt( ) ...................................................................................... 95
BindCol( ) ......................................................................................... 97
BindParam( ) ...................................................................................... 100
Cancel( ) ........................................................................................ 105
CloseCursor( ) .................................................................................. 107
ColAttribute( ) ................................................................................. 108
Columns( ) ...................................................................................... 111
Connect( ) ....................................................................................... 116
CopyDesc( ) .................................................................................... 119
DataSources( ) ................................................................................. 120
DescribeCol( ) ................................................................................ 123
Disconnect( ) .................................................................................. 126
EndTran( ) ...................................................................................... 127
Error( ) ........................................................................................ 129
ExecDirect( ) .................................................................................. 132
Execute( ) ....................................................................................... 135
Fetch( ) ........................................................................................ 137
FetchScroll( ) ................................................................................ 139
FreeConnect( ) ............................................................................... 142
FreeEnv( ) ...................................................................................... 143
FreeHandle( ) .................................................................................. 144
FreeStmt( ) ..................................................................................... 146
GetConnectAttr( ) .......................................................................... 148
GetConnectOption( ) ...................................................................... 150
GetCursorName( ) .......................................................................... 152
GetData( ) ...................................................................................... 154
Contents

Appendix A Diagnostic Cross-reference .............................................................. 233
Appendix B State Tables ........................................................................................ 239
B.1 Environment State Transitions ...................................................................... 240
B.2 Connection State Transitions ........................................................................ 241
B.3 Statement Transitions .................................................................................... 242
B.3.1 Data-at-Execute Dialogue .......................................................................... 244
B.4 Descriptor State Transitions .......................................................................... 245
Appendix C Language-specific Header Files ..................................................... 247
C.1 C-language File <sqlcli.h> ........................................................................... 247
C.2 COBOL File SQLCLI.CBH ............................................................................ 258
Appendix D Sample C Programs ......................................................................... 265
D.1 Create Table, Insert, Select ........................................................................... 265
D.2 Interactive Query .......................................................................................... 268
D.3 Providing Long Dynamic Arguments at Execute Time .............................. 272

GetDescField() ..................................................................................................... 157
GetDescRec() ........................................................................................................... 160
GetDiagField() ....................................................................................................... 163
GetDiagRec() .......................................................................................................... 166
GetEnvAttr() .......................................................................................................... 171
GetFunctions() ...................................................................................................... 173
GetInfo() ................................................................................................................ 176
GetStmtAttr() ........................................................................................................ 183
GetStmtOption() .................................................................................................... 185
GetTypeInfo() ......................................................................................................... 187
NumResultCols() .................................................................................................. 191
ParamData() ............................................................................................................ 192
Prepare() .................................................................................................................. 194
PutData() ................................................................................................................ 196
RowCount() .............................................................................................................. 198
SetConnectAttr() .................................................................................................. 199
SetConnectOption() .............................................................................................. 201
SetCursorName() ................................................................................................... 203
SetDescField() ...................................................................................................... 205
SetDescRec() .......................................................................................................... 208
SetEnvAttr() ........................................................................................................... 211
SetParam() .............................................................................................................. 213
SetStmtAttr() ....................................................................................................... 214
SetStmtOption() ................................................................................................... 216
SpecialColumns() ................................................................................................. 218
Statistics() ............................................................................................................. 222
Tables() .................................................................................................................. 226
Transact() .............................................................................................................. 229

vi x/Open CAE Specification
Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world’s largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open’s strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- **CAE Specifications**

  CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

  CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.
• Preliminary Specifications

These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

Preliminary specifications are analogous to the trial-use standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

• Guides

These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

• Technical Studies

X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open’s Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

• Snapshots

These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

 Versions and Issues of Specifications

As with all live documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

• a new Version indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it replaces the previous publication.
• a new Issue does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains both the previous and new issue as current publications.

**Corrigenda**

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done either by email to the X/Open info-server or by checking the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information by email, send a message to info-server@xopen.co.uk with the following in the Subject line:

request corrigenda; topic index

This will return the index of publications for which Corrigenda exist.

**SQL Access Group moved into X/Open**

Until the fourth quarter of 1994, the SQL Access Group was a separate body operating in the U.S.A., working with X/Open under a joint development and publishing agreement. It developed SQL-related specifications in collaboration with the X/Open Data Management Group.

During the fourth quarter of 1994, the SQL Access Group transferred all its assets and current activities to X/Open. These activities are now continuing in a technical working group within X/Open, called the X/Open SQL Access Group.

Much of the development work on this SQL CLI CAE Specification was achieved before the transfer of the SQL Access Group to X/Open. Under the joint agreement which applied before this transfer, this joint CLI development work resulted in joint publication of the SQL CLI Snapshot (S203) in October 1992, followed by the SQL CLI Preliminary Specification (P303) in October 1993. The Acknowledgements section of this publication expresses recognition of this progression, which has now resulted in publication of this SQL CLI CAE Specification.

**This Document**

This document is a CAE Specification (see above). It is intended for application programmers using the SQL Call Level Interface (CLI), which is an application programming interface (API) for database access.

Readers are expected to be familiar with the X/Open SQL specification to enable them to understand the definition of a database and the intended result of executing SQL statements.

This document is structured as follows:

- Chapter 1 is an introduction to CLI.
- Chapter 2 describes concepts and conventions of CLI.
- Chapter 3 describes the CLI data structures.
- Chapter 4 presents an overview of the interface.
- Chapter 5 discusses diagnostics.
• Chapter 6 presents an overview of the concise functions.
• Chapter 7 presents an overview of the deprecated functions.
• Chapter 8 contains detailed descriptions of each function.
• Appendix A cross-references error codes and functions.
• Appendix B contains state tables.
• Appendix C contains language-specific header files.
• Appendix D contains two sample C programs.
• Appendix E provides mappings to and from X/Open embedded SQL.
• A glossary and index are provided at the end of the book.

Typographical Conventions
The following typographical conventions are used throughout this document:

• **Bold** font is used in text for filenames, keywords, type names, data structures and their members.

• **Italic** font is used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  — variable names, for example, substitutable argument prototypes
  — functions; these are shown as follows: `name()`.  

• Normal font is used for the names of constants and literals.

• The notation `<file.h>` indicates a C-language header file.

• Names surrounded by braces, for example, `{ARG_MAX}`, represent C-language symbolic limits or configuration values, which may be declared in appropriate header files by means of the C `#define` construct.

• The notation `[ABCD]` is used to identify a C-language coded return value.

• Error classes are indicated by 5-character character-string literals; where these occur in text, they are shown in SQL syntax, for example, `'HY001'`.

• Syntax, code examples and user input in interactive examples are shown in fixed width font.

• Variables within syntax statements are shown in italic fixed width font.

• Shading is used, with margin notation, in the following cases:
  
  **DE** — Discussions of deprecated features.  
  
  **EX** — Discussions of features that are extensions to features of the X/Open SQL specification.  
  
  **OP** — Discussions of optional features.  

Refer to Section 1.5 on page 8 for details.
Trade Marks

UNIX® is a registered trade mark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open® is a registered trade mark, and the “X” device is a trade mark, of X/Open Company Limited.
Acknowledgements

X/Open gratefully acknowledges past collaboration with and contributions from members of the SQL Access Group, which transferred its activities and assets to X/Open in the fourth quarter of 1994. Its activities are now continuing in a technical working group within X/Open, called the X/Open SQL Access Group.
The following documents are referenced in this specification:

ANSI CLI
   X3H2-93-082, ISO Working Draft — SQL Call Level Interface (CLI) (Addendum 1 to SQL-92).

DTP

Internationalisation Guide

ISO C

ISO SQL

RDA

SQL

SQL Technical Corrigendum

TX

XA
Chapter 1

Introduction

The SQL Call Level Interface (CLI) is an application programming interface (API) for database access. CLI is an alternative invocation technique to dynamic SQL that provides essentially equivalent operations. CLI is a set of functions that application programs call directly using normal function call facilities, whereas embedded SQL is typically converted by a preprocessor.

The definition of CLI relies heavily on the referenced X/Open SQL specification, which defines a database and the intended result of executing SQL statements.

This chapter traces the development of CLI, explains its relationship to the X/Open SQL specification and the ISO SQL standard, and defines terms used to gauge compliance with this specification.

1.1 Development of CLI

SQL was originally developed as a way to embed, in an application program, static or dynamic operations on a database. Embedded SQL code is typically converted by an implementation-specific preprocessor into code that is compiled and executed.

Dynamic SQL makes SQL more flexible and applies it to cases where the desired database operations are not known at the time the application program is written. For example, in fourth-generation languages, the desired database operations are often based on interaction with the user. Dynamic SQL lets SQL statement text reside in host-language character strings. The application generates them and the SQL implementation interprets them dynamically during the course of the program's execution. Dynamic SQL is still an embedded invocation technique and still typically works through a preprocessor. The X/Open SQL specification specifies both static and dynamic SQL.

CLI gives SQL even more flexibility. It addresses several developments in the database industry:

- **Client/server architecture**
  
  Databases are increasingly structured as clients and servers. Both the ISO SQL standard and the X/Open SQL specification conceptualise SQL in terms of client and server operations. When clients and servers are separated, the application writer may not know what operations are desired or even what the structure of the database is.

  CLI is ideally suited for a client/server environment, in which the target database is not known when the application program is built. CLI is totally symmetric with regard to database operations, providing the same syntax to execute any SQL data definition or data manipulation statement.

- **Preprocessor-independence**
  
  Embedded SQL's assumption of a preprocessor typically requires that portable applications are distributed as source code. Developers are reluctant to disclose proprietary source code.

  CLI permits the production of portable object modules containing SQL database operations. There are no implementation-specific transformations on source code; implementation-specific features and added value reside in the vendor's CLI run-time library.

  A goal of CLI is binary compatibility of object modules. Binary compatibility may be restricted by factors outside the scope of this document, such as choice of the processor,
operating system and, sometimes, memory model.

- **Concurrent processing**

  Applications are increasingly specifying concurrent processing, including concurrent database operations. The existence of global data areas in SQL raises the question of the scope and visibility of each change to such data.

  CLI eliminates global data areas, associating all implementation data that is accessible to the application with a specific handle that the implementation passes to the application.

- **Distributed transaction processing (DTP)**

  DTP distributes work between processors, with the guarantee that either all operations or none are committed (global atomicity). The referenced X/Open DTP, XA and Transaction Demarcation specifications address this topic.

  The X/Open SQL specification delimits transactions using the COMMIT and ROLLBACK statements. A transaction begins implicitly when the application operates on a database. The X/Open SQL specification mentions a technique to permit SQL work to be completed atomically with non-SQL work, and to permit the application more precisely to delimit transactions.

  However, the X/Open SQL specification assumes a model of a single transaction active on a single connection. In more general models, the COMMIT and ROLLBACK statements do not specify to which work they pertain.

  The transaction model in CLI is a basic model in which the application completes all active work in the database environment. An alternative model, in which the application can complete transactions on individual connections, is allowed as an implementation option.
1.2 Relation to Other X/Open Documents

1.2.1 Conceptual Differences from Embedded SQL

CLI introduces a new style of application program binding for SQL that contains elements of X/Open embedded SQL and of direct invocation as defined in the referenced ISO SQL standard. However, CLI is conceptually different from prior SQL implementations in the following ways:

- **Execution model**
  CLI introduces a new model for the execution of any SQL statement that is preparable in dynamic SQL. CLI does not require explicit declaration of cursors, nor does it require a different SQL verb (OPEN as opposed to EXECUTE) depending on the SQL text.

- **Cursor**
  The CLI cursor model is a mixture of the current dynamic and direct invocation binding styles. Executing a cursor-specification\(^1\) can return multiple rows even though the application does not explicitly declare a cursor. The application can also use the normal cursor fetch model on such cursor-specifications; it can also use positioned UPDATE and DELETE statements. This follows from the rule that any preparable SQL statement can be executed using CLI.

- **Statement handles**
  A statement handle is a variable that refers to an implementation-defined data structure used to contain all information related to an SQL statement. The statement handle corresponds roughly to the diagnostics area and SQLSTATE of embedded SQL (see Section 3.1 on page 26).

- **Environment, connection and descriptor handles**
  These other handles take the place of all remaining global variables, of connection-specific state, and of SQL descriptor areas in embedded SQL (see Section 3.1 on page 26).

- **Automatic sizing of data structures**
  For data structures with a variable number of records, such as a diagnostics area or an SQL descriptor, the CLI implementation takes any necessary action to accommodate however many records are written to the data structure. The application does not have to declare the desired number of records when it allocates the data structure.

- **Automatic data conversion**
  In CLI, the application can specify the host-language buffer format for dynamic parameters and column data. If this differs from the format used for communication with the server, the client automatically converts data when it sends dynamic arguments to the database and when it fetches columns from the database.

\(^1\) Throughout this specification, cursor-specification refers to the entire syntax of the cursor-specification (SELECT statement) defined in the X/Open SQL specification. This does not include the SELECT...INTO syntax of the dynamic FETCH statement.
1.2.2 Feature Extensions over Embedded SQL

CLI is an alternative method for application programs to gain access to the features described in the X/Open SQL specification. CLI might be implemented by adapting an existing embedded SQL implementation. For this reason, this edition of CLI generally confines itself to features contained in the X/Open SQL specification.

The exceptions to this rule are listed below. They represent cases where the X/Open SQL specification has not yet embraced features of the ISO SQL standard. This edition specifies the feature based on its definition in the ISO SQL standard:

- Semantics for date/time/timestamp appear in this document (see Section 4.8.3 on page 60). CLI does not yet specify intervals.
- The SQL_CATALOG_NAME characteristic, obtained by calling GetInfo() indicates whether the server uses three-part object naming. This specification lets implementations support a subset of three-part naming (see Naming on page 54); the X/Open SQL specification does not address this possibility.
- The application can determine the type of SQL statement that was most recently executed, by calling GetDiagField(). This corresponds to the DYNAMIC_FUNCTION field of the diagnostics area in the ISO SQL standard.
- An application can obtain information on the properties of any dynamic parameters in a prepared statement. This optional capability corresponds to the <describe input statement> in the ISO SQL standard.
- The criteria by which the implementation selects a diagnostic record to appear first in the diagnostic area are different from those described in the X/Open SQL specification and reflect modifications to the ISO SQL standard documented in the ISO SQL Technical Corrigendum.
- A statement attribute lets the application specify that cursors are sensitive or are insensitive to changes made to a result set by another cursor.
- The FetchScroll() function permits the cursor to be moved to a row of a result set other than the next sequential row, provided the application has successfully set an attribute of a statement handle to enable the “scrollable cursors” feature.

In the following case, CLI includes semantics that do not correspond to anything in either the X/Open SQL specification or the ISO SQL standard:

- The Statistics() function may return information associated with an index but not available by querying the INDEXES system view.

Eventually, equivalent semantics will be added to the X/Open SQL specification and X/Open CLI will reference those descriptions.

Discussions of features that are extensions to features of the X/Open SQL specification are shaded with the EX margin notation, as shown here.
1.3 Relation to Standards

X/Open is aware of the discrepancies between this document and the ISO CLI Draft International Standard in the following list:

- The X/Open SQL specification aligns with Entry Level SQL as defined in the ISO SQL standard. The ISO SQL standard defines other levelling rules and thereby provides features not present in X/Open SQL. The ISO CLI Draft International Standard takes advantage of some of these features, while X/Open CLI does not. Examples are the BIT, BIT VARYING, and INTERVAL data types.

- X/Open CLI contains a set of metadata functions that are not present in the ISO CLI Draft International Standard.

- Certain other minor discrepancies exist, for which X/Open intends to propose corrections to the ISO CLI Draft International Standard.

Where other discrepancies remain between this CAE specification and the eventual International Standard (for instance, in cases of oversights or editorial errors), X/Open intends to issue a statement explicitly deferring to the International Standard, so that it is the authority by which any discrepancies are resolved.

X/Open's goal is that implementations be able to conform both to X/Open CLI and to the eventual International Standard, and that application writers have clear guidelines for writing applications with maximum portability.
1.4 Compliance Policy

1.4.1 Language Binding

This document describes a set of CLI functions that are callable from C and COBOL. Some CLI products support additional languages.

The goal of this X/Open specification is the ability to write portable programs. Conformance to this specification means that the CLI implementation must include bindings to an X/Open-compliant COBOL or C implementation. It is implementation-defined whether bindings to both COBOL and C are provided, and to which other languages an implementation provides bindings. X/Open intends to publish, for each SQL product it brands, the extent to which purchasers can use the product to write portable applications.

1.4.2 SQL Statement Text

This specification gives applications a way to provide SQL statement text for execution. The SQL statement text is typically a statement from the database language specified in the X/Open SQL specification. Compliance with this document is separate from, and does not presume, compliance with the X/Open SQL specification. However, X/Open recommends that implementations comply with both this specification and the X/Open SQL specification.

On an implementation that complies with both this specification and the X/Open SQL specification, valid SQL statement text for execution using CLI is defined as any SQL statement that can be prepared in dynamic SQL, as specified by the X/Open SQL specification: CREATE, cursor-specification, searched DELETE, dynamic positioned DELETE, DROP, GRANT, INSERT, REVOKE, searched UPDATE, dynamic positioned UPDATE and the vendor escape clause. The COMMIT and ROLLBACK statements of dynamic SQL are specifically excluded from execution using CLI, as this specification provides other methods of transaction delimitation (see Section 4.9).

In addition, any dynamic arguments must appear so that their data type can be deduced, and prefixes, terminators, comments and embedded variable names are prohibited. Refer, in the X/Open SQL specification, to the explanation of the '42000' diagnostic for the PREPARE statement.

1.4.3 Distributed Transaction Delimitation

Section 4.9 on page 63 discusses transactions, which are sequences of database operations with certain collective characteristics such as atomicity. Transaction delimitation must exist in order to define the grouping of database operations into transactions.

All transactions must be delimited in exactly one of the following ways, selected based on implementation-defined criteria:

- A transaction begins when an application operates on a database, as defined in Section 4.9 on page 63. The transaction ends when the application calls EndTran().
- A transaction begins when an application executes the tx_begin() function described in the X/Open TX specification; and ends when it executes tx_end().
- An implementation-defined transaction delimitation interface is used.

The application must mark the end of a transaction using the same technique it used to mark the start of a transaction.

X/Open-compliant CLI implementations that also comply with the X/Open XA specification support the second option for all transactions. The tx_begin() and tx_end() functions call a
transaction manager, which coordinates completion of SQL and non-SQL work so as to provide global atomicity. The CLI implementation supports the X/Open XA interface, in the role of a Resource Manager (RM). The XA interface lets the transaction manager inform the CLI implementation of the delimitation and disposition of transactions.
1.5 Compliance Terminology

The following compliance terms convey the same meanings as defined in the X/Open SQL specification.

Optional features
An optional feature serves as guidance to implementors on the preferred syntax for a feature that is not yet widespread. Implementors are currently free not to implement the feature, but if they implement it, they should do so as specified in this document. X/Open does not currently enforce the implementation of optional features, but intends to make them mandatory in future issues of the XPG, in the manner in which they are specified in this edition. At that time, implementations will be required to provide the feature.

X/Open may test implementations to see if they implement optional features as specified in this document. X/Open would make the results available to prospective purchasers.

Application writers may use optional features that are known to be available on the implementation in use, at the risk of reduced portability.

Discussions of optional features are shaded with the OP margin notation, as shown below.

The following features are optional:

- The ability to have the implementation describe dynamic parameters in prepared statements.
- An application’s ability to request scorable cursors.
- An application’s ability to specify that cursors are either sensitive or insensitive (to changes made to a result set by another cursor).

Deprecated features
Deprecated features include syntax that X/Open views as obsolete or non-optimal. Implementors must provide features labelled deprecated, in the interest of backward-compatibility. Application writers using deprecated features are advised that X/Open intends to remove them from future issues of this specification.

Each deprecated feature lists a preferred method of performing the same function. X/Open’s policy on deprecated features is to maintain the deprecated designation for at least one issue of the XPG. This gives application writers adequate notice to change their coding to the recommended method. When X/Open reissues the XPG with a feature omitted, implementations may remove support for the feature.

Currently, the only deprecated features in CLI are the functions described in Chapter 7.

Compliance
A CLI implementation is X/Open-compliant if it supports all the assertions this document makes that are not labelled optional. The implementation may also support the features labelled optional, or other features not specifically identified in this document.

An application program is X/Open-compliant if it uses only the syntax contained in this document that is not labelled optional.

Implementation-defined
Implementation-defined means that the resolution of the issue in question may vary between implementations, and that each X/Open-compliant implementation must publish information on how it resolves that issue.

Undefined
Undefined means that the resolution of the issue in question may vary between
implementations, and that an X/Open-compliant implementation need not publish information on how it resolves that issue.

Footnotes are used as a technique to improve readability of the main text. However, information in footnotes is as much a part of this specification as information in the main text.
Chapter 2

Concepts and Conventions

This chapter describes substantive concepts of CLI that must be understood prior to encountering the CLI functions in Chapter 4.

Concepts presented in this chapter include syntactic characteristics of CLI (the naming and variants of CLI functions), use of host-language data types, and semantic conventions.

2.1 Syntactic Characteristics

2.1.1 Function Variants

This specification defines generic CLI functions. Each function has a by-value variant and a by-reference variant.

When using the by-value variant, the application provides each input argument as an actual value. By-value variants begin with the prefix SQL; for example, SQLAllocHandle(). The C examples in the reference manual pages use by-value variants. They typically simplify coding because constants can be coded as input arguments. If the binding language is a language other than C, then it is implementation-defined whether by-value CLI variants are available.

When using the by-reference variant, the application provides a pointer to each input argument and deferred input argument. By-reference CLI variants begin with the prefix SQLR; for example, SQLRAllocHandle(). The COBOL examples in the reference manual pages use by-reference variants. It is implementation-defined whether by-reference variants are available in C.

For output parameters in both variants, the application provides a pointer to the output argument.

2.1.2 Short Function Names

In any programming environment where the CLI function names published in this document cannot be used (because of their length or because some are made identical by truncation), the published function names must cause the generation of the abbreviated names for the CLI functions listed in Table 2-1 on page 12.

---

2. An exception is made for arguments that supply the value of a deferred descriptor field (see Section 3.5.5 on page 36). For these, pointers are used in both the by-value and by-reference variants.
### Table 2-1 Abbreviations for CLI Function Names

<table>
<thead>
<tr>
<th>Published Generic Name</th>
<th>Abbreviation</th>
<th>by-Value Variant</th>
<th>by-Reference Variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocConnect()</td>
<td>DE</td>
<td>SQLAC</td>
<td>SQRAC</td>
</tr>
<tr>
<td>AllocEnv()</td>
<td>DE</td>
<td>SQLAE</td>
<td>SQRAE</td>
</tr>
<tr>
<td>AllocHandle()</td>
<td></td>
<td>SQLAH</td>
<td>SQRAH</td>
</tr>
<tr>
<td>AllocStmt()</td>
<td>DE</td>
<td>SQLAS</td>
<td>SQRAS</td>
</tr>
<tr>
<td>BindCol()</td>
<td></td>
<td>SQLBP</td>
<td>SQRBP</td>
</tr>
<tr>
<td>BindParam()</td>
<td></td>
<td>SQLCAN</td>
<td>SQRCAN</td>
</tr>
<tr>
<td>Cancel()</td>
<td></td>
<td>SQLCC</td>
<td>SQRCC</td>
</tr>
<tr>
<td>CloseCursor()</td>
<td></td>
<td>SQLCO</td>
<td>SQRCO</td>
</tr>
<tr>
<td>ColAttribute()</td>
<td>DE</td>
<td>SQLCOL</td>
<td>SQRCOL</td>
</tr>
<tr>
<td>Columns()</td>
<td></td>
<td>SQLCON</td>
<td>SQRCON</td>
</tr>
<tr>
<td>CopyDesc()</td>
<td></td>
<td>SQLCD</td>
<td>SQRCD</td>
</tr>
<tr>
<td>DataSource()</td>
<td></td>
<td>SQLDS</td>
<td>SQRDS</td>
</tr>
<tr>
<td>DescribeCol()</td>
<td></td>
<td>SQLDC</td>
<td>SQRDC</td>
</tr>
<tr>
<td>Disconnect()</td>
<td></td>
<td>SQLDIS</td>
<td>SQRDIS</td>
</tr>
<tr>
<td>EndTran()</td>
<td></td>
<td>SQLLET</td>
<td>SQRRET</td>
</tr>
<tr>
<td>Error()</td>
<td>DE</td>
<td>SQLER</td>
<td>SQRER</td>
</tr>
<tr>
<td>ExecDirect()</td>
<td></td>
<td>SQLED</td>
<td>SQRED</td>
</tr>
<tr>
<td>Execute()</td>
<td></td>
<td>SLEX</td>
<td>SREX</td>
</tr>
<tr>
<td>Fetch()</td>
<td></td>
<td>SQLFT</td>
<td>SQRFT</td>
</tr>
<tr>
<td>FetchScroll()</td>
<td></td>
<td>SQLFTS</td>
<td>SQRFTS</td>
</tr>
<tr>
<td>FreeConnect()</td>
<td>DE</td>
<td>SQLFC</td>
<td>SQRFC</td>
</tr>
<tr>
<td>FreeEnv()</td>
<td>DE</td>
<td>SQLFE</td>
<td>SQRFE</td>
</tr>
<tr>
<td>FreeHandle()</td>
<td></td>
<td>SQLFH</td>
<td>SQRFH</td>
</tr>
<tr>
<td>FreeStmt()</td>
<td>DE</td>
<td>SQLFS</td>
<td>SQRFS</td>
</tr>
<tr>
<td>GetConnectAttr()</td>
<td></td>
<td>SQLGCA</td>
<td>SQRGCA</td>
</tr>
<tr>
<td>GetConnectOption()</td>
<td>DE</td>
<td>SQLGCO</td>
<td>SQRGCO</td>
</tr>
<tr>
<td>GetCursorName()</td>
<td></td>
<td>SQLGCN</td>
<td>SQRGCN</td>
</tr>
<tr>
<td>GetData()</td>
<td></td>
<td>SQLGDA</td>
<td>SQRGDA</td>
</tr>
<tr>
<td>GetDescField()</td>
<td></td>
<td>SQLGDF</td>
<td>SQRGDF</td>
</tr>
<tr>
<td>GetDescRec()</td>
<td></td>
<td>SQLGDR</td>
<td>SQRGDR</td>
</tr>
<tr>
<td>GetDiagField()</td>
<td></td>
<td>SQLGXF</td>
<td>SQRGXF</td>
</tr>
<tr>
<td>GetDiagRec()</td>
<td></td>
<td>SQLGXR</td>
<td>SQRGXR</td>
</tr>
<tr>
<td>GetEnvAttr()</td>
<td></td>
<td>SQLGEA</td>
<td>SQRGEA</td>
</tr>
<tr>
<td>GetFunctions()</td>
<td></td>
<td>SQLGFU</td>
<td>SQRGFU</td>
</tr>
<tr>
<td>GetInfo()</td>
<td></td>
<td>SQLGI</td>
<td>SQRGI</td>
</tr>
<tr>
<td>GetStmtAttr()</td>
<td></td>
<td>SQLGSA</td>
<td>SQRGSA</td>
</tr>
<tr>
<td>GetStmtOption()</td>
<td>DE</td>
<td>SQLGSO</td>
<td>SQRGSO</td>
</tr>
<tr>
<td>GetTypeInfo()</td>
<td></td>
<td>SQLGTI</td>
<td>SQRGTI</td>
</tr>
<tr>
<td>NumResultCols()</td>
<td></td>
<td>SQLNRC</td>
<td>SQRNRC</td>
</tr>
<tr>
<td>ParamData()</td>
<td></td>
<td>SQLPRD</td>
<td>SQRPRD</td>
</tr>
<tr>
<td>Prepare()</td>
<td></td>
<td>SQLPR</td>
<td>SQRPR</td>
</tr>
<tr>
<td>PutData()</td>
<td></td>
<td>SQLPTD</td>
<td>SQRPTD</td>
</tr>
<tr>
<td>RowCount()</td>
<td>DE</td>
<td>SQLRC</td>
<td>SQRRC</td>
</tr>
<tr>
<td>SetConnectAttr()</td>
<td></td>
<td>SQLSCA</td>
<td>SQRSCA</td>
</tr>
<tr>
<td>SetConnectOption()</td>
<td>DE</td>
<td>SQLSCO</td>
<td>SQRSCO</td>
</tr>
<tr>
<td>SetCursorName()</td>
<td></td>
<td>SQLSCN</td>
<td>SQRSCN</td>
</tr>
</tbody>
</table>
## Concepts and Conventions

### Syntactic Characteristics

<table>
<thead>
<tr>
<th>Published Generic Name</th>
<th>Abbreviation</th>
<th>by-Value Variant</th>
<th>by-Reference Variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetDescField()</td>
<td>SQLSDF</td>
<td></td>
<td>SQRSDF</td>
</tr>
<tr>
<td>SetDescRec()</td>
<td>SQLSDR</td>
<td></td>
<td>SQRSDR</td>
</tr>
<tr>
<td>SetEnvAttr()</td>
<td>SQLSEA</td>
<td></td>
<td>SQRSEA</td>
</tr>
<tr>
<td>SetParam()</td>
<td>SQLSP</td>
<td>DE</td>
<td>SQRSP</td>
</tr>
<tr>
<td>SetStmtAttr()</td>
<td>SQLSSA</td>
<td>DE</td>
<td>SQRSSA</td>
</tr>
<tr>
<td>SetStmtOption()</td>
<td>SQLSSO</td>
<td>DE</td>
<td>SQRSSO</td>
</tr>
<tr>
<td>SpecialColumns()</td>
<td>SQLSC</td>
<td></td>
<td>SQRSC</td>
</tr>
<tr>
<td>Statistics()</td>
<td>SQLSTA</td>
<td></td>
<td>SQRSTA</td>
</tr>
<tr>
<td>Tables()</td>
<td>SQLTAB</td>
<td></td>
<td>SIRTAB</td>
</tr>
<tr>
<td>Transact()</td>
<td>SQLTR</td>
<td>DE</td>
<td>SQRTR</td>
</tr>
</tbody>
</table>
2.1.3 Case Sensitivity

Names specified in this document use both upper-case and lower-case letters. The capitalisation of names helps indicate the function of the name, as described in the next section. An application must exactly follow this document’s use of capitalisation if the base programming environment is case-sensitive. In programming environments that are not case-sensitive, a CLI name such as:

\texttt{SQLAllocHandle}

may be written:

\texttt{SQLALLOCHANDLE}

In programming environments that do not support lower-case letters, names must be capitalised even though this document shows them containing lower-case letters.

2.1.4 COBOL Naming Variations

This document refers to values and to their symbolic names using the symbols defined for the C language. (These symbols are defined in the header file presented in Section C.1 on page 247.)

In COBOL programs, different symbols are used for the reasons explained in this section. (The symbols for COBOL are defined in the header file presented in Section C.2 on page 258.)

Underscore

This document defines symbolic names using the underscore character, such as SQL\_SUCCESS\_WITH\_INFO. In languages such as COBOL that do not use the underscore character, the hyphen is used instead: SQL\_SUCCESS\_WITH\_INFO.

Multiple Names for Different Data Types

In some cases, a symbolic name is valid in contexts that call for an SQL SMALLINT data type and in other contexts that call for an SQL INTEGER data type. The COBOL language requires different constants for the respective contexts. The entire set of these COBOL symbolic names appears below.

<table>
<thead>
<tr>
<th>C Symbol Used in This Document</th>
<th>COBOL Symbol for SMALLINT Contexts</th>
<th>COBOL Symbol for INTEGER Contexts</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ATTR_suffix</td>
<td>SQL_OPT_suffix</td>
<td>SQL_ATTR_suffix</td>
</tr>
<tr>
<td>SQL_FALSE</td>
<td>SQL_FALSE</td>
<td>SQL_FALSEL</td>
</tr>
<tr>
<td>SQL_NTS</td>
<td>SQL_NTS</td>
<td>SQL_NTSL</td>
</tr>
<tr>
<td>SQL_TRUE</td>
<td>SQL_TRUE</td>
<td>SQL_TRUEL</td>
</tr>
</tbody>
</table>
2.2 Use of Data Types

The following sections discuss the data types used in the reference manual pages in Chapter 8.

2.2.1 Generic Data Types for Function Parameters

The generic parameter types are defined in order to relate to SQL data types. This permits bindings to programming languages other than C and COBOL.

The ANY generic parameter type represents a function parameter whose type is not fixed but may vary between other data types from invocation to invocation.

The following table shows the most nearly equivalent C and COBOL data types to the generic CLI function parameters.

<table>
<thead>
<tr>
<th>Generic Parameter Type</th>
<th>C Data Type</th>
<th>COBOL Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(L)</td>
<td>unsigned char[L+1]</td>
<td>PICTURE X(L)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>short</td>
<td>PICTURE S9(4) BINARY</td>
</tr>
<tr>
<td>INTEGER</td>
<td>long</td>
<td>PICTURE S9(9) BINARY</td>
</tr>
<tr>
<td>ANY</td>
<td>void * (see above)</td>
<td>(see above)</td>
</tr>
</tbody>
</table>

Table 2-2 Generic Parameter Data Types: Correspondences for C and COBOL

Certain situations and COBOL implementations may require the use of different PICTURE specifications. (See the X/Open SQL specification for alternative PICTURE specifications.)
2.2.2 Data Types for C

The C-language synopsis for CLI uses the following symbolic names for all C data types. For binary compatibility of object modules, applications must use symbolic data types defined as follows (see, for example, the `<sqlcli.h>` header file in Appendix C):

<table>
<thead>
<tr>
<th>Symbolic Data Type</th>
<th>C Type</th>
<th>Typical Usage in CLI</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCHAR</td>
<td>unsigned char</td>
<td>Representation of CHAR and VARCHAR column data.</td>
</tr>
<tr>
<td>EX SQLDATE</td>
<td>unsigned char</td>
<td>Representation of DATE column data.</td>
</tr>
<tr>
<td>SQLDECIMAL</td>
<td>unsigned char</td>
<td>Representation of DECIMAL column data.</td>
</tr>
<tr>
<td>EX SQLDOUBLE</td>
<td>double</td>
<td>Representation of FLOAT and DOUBLE PRECISION column data.</td>
</tr>
<tr>
<td>SQLHDBC</td>
<td>long</td>
<td>Handle referencing connection information.</td>
</tr>
<tr>
<td>SQLHDESC</td>
<td>long</td>
<td>Handle referencing descriptor information.</td>
</tr>
<tr>
<td>SQLHENV</td>
<td>long</td>
<td>Handle referencing environment information.</td>
</tr>
<tr>
<td>SQLHSTMT</td>
<td>long</td>
<td>Handle referencing statement information.</td>
</tr>
<tr>
<td>SQLINTEGER</td>
<td>long</td>
<td>Length of buffers for column data and representation of INTEGER column data.</td>
</tr>
<tr>
<td>SQLNUMERIC</td>
<td>unsigned char</td>
<td>Representation of NUMERIC column data.</td>
</tr>
<tr>
<td>SQLPOINTER</td>
<td>void *</td>
<td>Representation of data types determined at run time.</td>
</tr>
<tr>
<td>SQLREAL</td>
<td>float</td>
<td>Representation of REAL column data.</td>
</tr>
<tr>
<td>SQLRETURN</td>
<td>short</td>
<td>Return code from CLI functions.</td>
</tr>
<tr>
<td>SQLSMALLINT</td>
<td>short</td>
<td>Column numbers, dynamic parameter numbers, and representation of SMALLINT column data.</td>
</tr>
<tr>
<td>EX SQLTIME</td>
<td>unsigned char</td>
<td>Representation of TIME column data.</td>
</tr>
<tr>
<td>EX SQLTIMESTAMP</td>
<td>unsigned char</td>
<td>Representation of TIMESTAMP column data.</td>
</tr>
</tbody>
</table>

Table 2-3 Uses of C Data Types
2.2.3 Data Types for COBOL

Programs in COBOL use the following PICTURE specifications to support CLI. Although the symbolic names in the left column are not used in COBOL, they are included in the following table for comparison with the C data types:

<table>
<thead>
<tr>
<th>Symbolic Data Type (C)</th>
<th>COBOL PICTURE</th>
<th>Typical Usage in CLI</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCHAR</td>
<td>PIC X(n)</td>
<td>Representation of CHAR and VARCHAR column data.</td>
</tr>
<tr>
<td>SQLDATE</td>
<td>PIC X(10)</td>
<td>Representation of DATE column data.</td>
</tr>
<tr>
<td>SQLDECIMAL</td>
<td>PIC S9(p-s)V9(s)</td>
<td>Representation of DECIMAL column data. (p = precision; s = scale)</td>
</tr>
<tr>
<td>SQLDOUBLE</td>
<td>PIC X(n)</td>
<td>Representation of DOUBLE column data.</td>
</tr>
<tr>
<td>SQLHDBC</td>
<td>PIC S9(9) BINARY</td>
<td>Handle referencing connection information.</td>
</tr>
<tr>
<td>SQLHDESC</td>
<td>PIC S9(9) BINARY</td>
<td>Handle referencing descriptor information.</td>
</tr>
<tr>
<td>SQLHENV</td>
<td>PIC S9(9) BINARY</td>
<td>Handle referencing environment information.</td>
</tr>
<tr>
<td>SQLHSTMT</td>
<td>PIC S9(9) BINARY</td>
<td>Handle referencing statement information.</td>
</tr>
<tr>
<td>SQLINTEGER</td>
<td>PIC S9(9) BINARY</td>
<td>Length of buffers for column data and representation of INTEGER column data.</td>
</tr>
<tr>
<td>SQLNUMERIC</td>
<td>PIC S9(p-s)V9(s)</td>
<td>Representation of NUMERIC column data. (p = precision; s = scale)</td>
</tr>
<tr>
<td>SQLPOINTER</td>
<td>not used in COBOL</td>
<td>Representation of REAL column data.</td>
</tr>
<tr>
<td>SQLREAL</td>
<td>PIC X(n)</td>
<td>Return code from CLI functions.</td>
</tr>
<tr>
<td>SQLRETURN</td>
<td>PIC S9(4) BINARY</td>
<td>Column numbers, dynamic parameter numbers, and representation of SMALLINT column data.</td>
</tr>
<tr>
<td>SQLSMALLINT</td>
<td>PIC S9(4) BINARY</td>
<td></td>
</tr>
<tr>
<td>SQLTIME</td>
<td>PIC X(n)</td>
<td>Representation of TIME column data.*</td>
</tr>
<tr>
<td>SQLTIMESTAMP</td>
<td>PIC X(n)</td>
<td>Representation of TIMESTAMP column data.*</td>
</tr>
</tbody>
</table>

* The length n is 8 for SQLTIME and 19 for SQLTIMESTAMP if there is no fractional seconds component. See Section 4.8.3 on page 60 for the exact format and length of these data types.
2.3 Semantic Conventions

2.3.1 Direction of Parameters

Each parameter is either an input or an output parameter.

- Input parameters, denoted by the legend (input) in the parameter list in the reference manual pages, let the application pass data to the CLI implementation. In cases where a specific function call does not require a given input parameter, the implementation ignores the argument that the application passes.

- Output parameters, denoted by (output) in the parameter list, let the implementation return values to the application. In cases where a specific function call does not use an output parameter, the value returned in that parameter is undefined.

Parameters that supply a value of a deferred descriptor field (see Section 3.5.5 on page 36) are a special case, since the input or output argument is a pointer to the actual data.

There are no combined input/output parameters; no CLI function modifies its input arguments. This lets the by-value and by-reference variants use the same parameter list.

2.3.2 Null Pointers

In a programming language that lets the application use pointers, passing a null pointer (see Null Pointer on page 21) as an argument to a CLI function has the following effects:

- In some metadata functions (see Section 4.6 on page 53), input string parameters specify a basis for qualifying the query. Passing a null pointer or a zero-length string in these cases inhibits qualification of the query on that basis.

- In certain other cases specified in Chapter 8, passing a null pointer as an argument for an output parameter inhibits the implementation from returning a value to the application. This effect is restricted to output length parameters (see Null Pointer on page 21) and to other cases explicitly specified in Chapter 8.

For diagnostics pertaining to invalid uses of null pointers, see Section 5.4.8 on page 75.

2.3.3 Output Arguments in Non-success Cases

Except where otherwise noted in the reference manual pages, whenever a CLI function produces a return value of [SQL_ERROR], [SQL_INVALID_HANDLE] or [SQL_NO_DATA], all output arguments are undefined.

2.3.4 International Character Data

The characters used in database data, in SQL statement text, and in CLI string arguments, are taken from one or more character sets.

Rationale

Historically, character sets have been defined such that each character requires one byte of storage. By contrast, this specification also permits character sets in which each character requires multiple bytes of storage, and permits character sets in which various characters require different amounts of storage. (For additional background, see the X/Open Internationalisation Guide.)

This specification avoids the use of the term byte. It uses the term octet as a fixed measure of storage space, and the term character to denote a character independent of the required storage...
The embedded SQL database language deals in units of characters and generally conceals storage requirements from the application programmer. CLI always specifies in octets the length of character data passed between the application and the implementation. Some additional CLI features deal in units of characters. For example, CLI lets the application measure the length or maximum length of table columns in characters. Additional operations that measure strings in terms of characters (such as the CHAR_LENGTH and POSITION scalar functions) are available from CLI using SQL statement text.

**Character Set**

It is undefined how an application or CLI implementation determines the character set and null character for a string.

### 2.3.5 Variable-length Input Parameters

Input parameters that reference variable-length character data (such as column names, dynamic parameters and string attribute values) have an associated length parameter. If the application terminates strings with the null character, as is typical in C, then it provides as an argument either the length in octets of the string (not including the null terminator) or SQL_NTS (Null-terminated String).

Thus, a non-negative length argument specifies the actual length of the associated string. The length argument may be 0 to specify a zero-length string, which is distinct from a null value. The negative value SQL_NTS directs the implementation to determine the length of the string by locating the null terminator.

The application’s ability to specify SQL_NTS for input arguments is independent of whether it has specified null termination for output arguments (see **String Termination**).

### 2.3.6 Variable-length Output Parameters

Output parameters that reference variable-length character data have two associated length parameters:

- The application uses the associated input parameter to describe the length in octets of the buffer the application allocated.
- The implementation uses a separate output parameter to describe the length in octets of the data it generated (whether or not it fits in the allocated buffer).

(For database fetches, CLI returns separately the other information for which embedded SQL uses indicator variables; this information is not combined with length information.)

**String Termination**

The application has two alternatives for termination of output character strings:

- By default, the implementation terminates with a null character every string that it writes to a variable-length output parameter. The null character is defined by the character set that specifies the interpretation of the string. In this case, the application must allocate enough space to hold the maximum number of characters it expects, plus the null terminator. For example, assuming a single-byte character set, if the application expects a 10-character string from the database, it allocates and specifies a buffer size of 11 (one for the null terminator).

A buffer size equal to the length of the null terminator obtains the null terminator and nothing else. A buffer shorter than the length of the null terminator is invalid; such a buffer
cannot even transfer a zero-length string. However, in languages that support pointers, the application can prevent the setting of the output string argument by supplying a null pointer (see Section 2.3.2 on page 18).

- As an environment attribute (see Section 3.2 on page 28), the application can disable null termination of variable-length output data. In this case, the application allocates a buffer exactly as long as the longest string it expects, and a one-character buffer is meaningful. COBOL applications typically use this alternative.

A buffer size of 0 is valid in this case and prevents the implementation from setting the associated output string argument.

The examples in this specification assume that C programs enable null termination and COBOL programs disable it. For this reason, in some examples, the lengths of buffers vary depending on the programming language.

String Truncation

If the data the implementation generates does not fit in the application’s buffer, the CLI implementation returns in the associated output parameter the length of the data generated, not of the data returned, indicating truncation to the application.

That is, the size of the string the implementation can fit in the buffer is the argument value for the input length parameter (minus the size of any null terminator, as described above). If the implementation returns a greater number in the output length parameter, it means the implementation has truncated one or more characters from the string it placed in the buffer.

For example, assume a single-byte character set and suppose the buffer can hold 10 characters. (If null termination is in effect, this requires an 11-character buffer.) If the database value actually contains 20 characters, the implementation writes the first 10 characters into the buffer and returns an output length of 20. The returned length of 20 informs the application that it must allocate a 20-character buffer (21 characters if null termination is in effect) to obtain the entire data value without truncation.

Truncation is also indicated by the fact that the CLI function returns [SQL_SUCCESS_WITH_INFO] and by additional diagnostic information obtainable by calling GetDiagField() or GetDiagRec().

If a fetch operation indicates truncation, the application compares the output length and input length value of each column to see which columns were truncated.

If, on the other hand, the value (including any null terminator) fits within the output buffer, then the output length argument accurately describes the string returned to the application (not including any null terminator). The CLI implementation leaves any remaining space in the buffer unchanged.

3. This paragraph does not apply to the two functions it names. They do not return [SQL_SUCCESS_WITH_INFO] even if they truncate an output argument.
Null Pointer

If the application passes a null pointer for the output length parameter, the implementation does not return an output length. This is typical when the application’s buffers are known to be large enough to accommodate any data the implementation returns. The application locates the null terminator to determine the length of the output. In languages that do not support pointers, this feature is not available. This feature should not be used when the function might truncate string data, because the function does not then inform the application of the length of the string before truncation.

If the application uses a null pointer, but the implementation needed a non-null pointer in order to return the value SQL_NULL_DATA, the attempt to fetch the column fails (see Fetch() on page 137 and BindCol() on page 97).

2.3.7 Interpretation of Strings

Normally, the CLI implementation interprets string arguments in a case-sensitive manner and does not trim any spaces from the values. Any exceptions to this appear in the appropriate reference manual pages.

2.3.8 Identifiers

SQL identifiers such as table names, column names and cursor names are arguments of some CLI functions. The X/Open SQL specification discusses these identifiers in the broader category of user-defined names. The X/Open SQL specification defines restrictions on user-defined names. CLI applications are also bound by these restrictions when they generate identifiers.

Length

The X/Open SQL specification imposes a portability requirement limiting the length of user-defined names to 18 characters. (C programs should use the symbolic constant [SQL_MAX_ID_LENGTH].) However, the referenced ISO SQL standard permits identifiers to have lengths to and including 128 characters. Portable applications should allocate enough space to accommodate identifiers of 128 characters returned by the implementation.

Delimited Identifiers

The X/Open SQL specification permits identifiers to be enclosed in double quotes ("'). The body of these delimited user-defined names can contain characters that the syntax of SQL would not permit or would interpret specially, including blanks and reserved keywords, and is treated in a case-sensitive manner.

The CLI implementation interprets an input argument as a delimited identifier if, after trimming leading and trailing spaces, the first character is a double quote and the last character is a double quote.

The CLI implementation never encloses in double quotes any SQL identifier that it returns as an output argument.
2.4 Database System

2.4.1 Clients and Servers
The operation of a database system effectively involves two processors, a client and a server. The terms client and server in this document always mean SQL client and SQL server. The Connect() function manages the associations between a client and one or more servers.

When a program is active, it is bound in an implementation-defined manner to a single client that processes the first implicit or explicit Connect() call. The client communicates with one or more servers, manages connections to servers, maintains the diagnostics area, and allocates data structures and handles. The server processes other CLI functions, including all operations on the database. Following these operations, diagnostic information is passed (in an undefined way) into the diagnostics area of the client.

Number and Location of Servers
It is implementation-defined whether there can be more than one server.

X/Open CLI provides location transparency: the application has no notion of whether the server is local or remote.

2.4.2 Metadata and Data
Each server provides a database, which consists of metadata and data.

- Metadata is the definitions of all active base tables, viewed tables, indexes, privileges and user names. (An item of metadata is active if it has been defined and has not subsequently been dropped.)
- Data comprises every value in every active table.

Schemata
A schema is a collection of related objects. A schema may contain base tables, and contains any indexes that are defined on the base tables. Namespace issues and ownership of schemata are discussed in the X/Open SQL specification.

2.4.3 Three-part Object Naming
A server may support catalogs, in which case every schema resides in a catalog. An application can uniquely identify a table or index by qualifying the table or index identifier (preceding it with its catalog and schema name). The use of catalog, schema, and object name is called three-part naming. Periods separate the catalog, schema, and object name.

It is implementation-defined which of the following object naming systems is supported\(^4\) as valid syntax for the qualifier:

- catalog-name.schema-name

---

\(^4\) An application can determine whether the server supports catalog names by obtaining the SQL_CATALOG_NAME characteristic from a call to GetInfo().
On servers that support catalog names, there may be a catalog that does not have a name.

Certain CLI metadata functions return result sets whose TABLE_CAT column indicates the catalog name of an object. For objects that do not have a catalog name, this column contains a zero-length string, except that if the implementation does not support catalog names, the column may instead be null.

The catalog-name and schema-name are each syntactically a user-defined-name. Other implementation-defined object naming systems may also be supported. In all cases, use of qualification is optional for the application program.

(Some implementations impose other restrictions on qualification; see the X/Open SQL specification.)
This chapter describes the data structures of CLI. Handles are described generally in Section 3.1 on page 26. Subsequent sections describe each type of data structure for which a handle exists. Section 3.6 on page 38 describes cursors and cursor names.
3.1 Handle Overview

A handle is a variable that refers to a data structure within the CLI implementation. CLI defines the following types of handle:

- SQLHENV Reference to an environment (see Section 3.2 on page 28).
- SQLHDBC Reference to a connection (see Section 3.3 on page 29).
- SQLHSTMT Reference to an SQL statement (see Section 3.4 on page 30).
- SQLHDESC Reference to a descriptor (see Section 3.5 on page 32).

Calling AllocHandle() allocates a handle of a specified type. In most cases the application specifies a parent handle of a different type. AllocHandle() allocates memory for, and defines, whatever data structures the implementation needs to manage the corresponding functions, and assigns a value to the handle that refers to these data structures. Once allocated, the application can pass a handle to subsequent CLI functions to indicate the environment, connection, statement or descriptor on which the function should act. The application should not perform any other operation on a handle, such as analysing or modifying its contents.

Null values are defined for each type of handle. AllocHandle() returns the null handle as one type of error indication.

The application calls FreeHandle() to deallocate a handle that it has allocated.

This specification uses handle parameter names colloquially to refer to the data structure referred to by the handle. For example, “preparing StatementHandle” technically means preparing the statement associated with StatementHandle.

3.1.1 Attributes

Attributes are defined for environments, connections, and statements. Some attributes are intrinsic properties of the implementation; others can be used to affect the subsequent behaviour of the implementation. The application can obtain the value of any attribute by providing the corresponding handle. The application can change the value of an attribute if it is described as a settable attribute.

A list of attributes appears in this chapter in the sections that discuss environments, connections, and statements. Attributes are described using the symbolic name for the SQL INTEGER value that C programs use to specify them. Within each list, each attribute is marked (read-only) or (settable).

Settable attributes may carry restrictions on the application’s ability to set the attribute’s value; see Timing of Attempts to Set Attributes on page 74.

3.1.2 Rationale

Handles obviate global variables. There is no global information in CLI; the application gains access to all information through a handle. This avoids problems with global variables that arise when asynchronous processing occurs within an application, as is the case for asynchronous statement processing and in multi-threaded environments. The application need not protect this global memory nor synchronise with other threads or processes to prevent unsolicited updates to global information.

Any implementation-defined limits on the maximum number of a given type of data structure are enforced by returning an error on any call to AllocHandle() that would exceed the limit.

All cases where only one data structure can be used at a time (such as for the current connection to a server) can be handled by the implementation without the application’s knowledge, by performing implicit switching based on the handles the application provides as CLI function
Handles make error reporting more logical. Certain diagnostics apply only to a given connection or only to the execution of a single SQL statement. Because there are handles for these concepts, diagnostic information in CLI is bound to the relevant data structure. The `GetDiagField()` and `GetDiagRec()` functions return any associated diagnostic information. When an implementation produces diagnostic information, it overwrites any previous information associated with the same handle, but it does not destroy unrelated diagnostic information, as is the case when diagnostic variables are global.

The only lexical restrictions on the use of handle variables are the scope rules of the relevant programming language. The domain in which the value of a handle can be used to refer to the associated data structure is implementation-defined.

### 3.1.3 Handle Type Identifier

Handle values do not necessarily self-describe their handle type, and the values of handles of different types are not necessarily allocated without duplication. For example, the value of both an environment handle and a statement handle might be 1.

In cases where a function's parameter accepts an argument of multiple handle types, a separate input argument specifies the handle type. The manual pages for those functions specify a diagnostic for the case where the application passes an invalid handle type identifier.

For each handle type, each handle value is unique across the CLI implementation, not just in its context. For example, two statement handles cannot have the same value even if allocated on different connections or environments. Thus, a handle, along with its handle type identifier, uniquely identifies that handle throughout the CLI implementation.
3.2 Environment

An environment is a global context for database access. Associated with the environment handle is any data that must be global to the application. This includes attributes (see below), the defined connections (see Section 3.3 on page 29), and information on which connection is current.

It is implementation-defined whether an application may allocate more than one environment. In cases where multiple environments are allowed, the effects on transaction semantics are discussed in Section 4.9 on page 63.

Environment Attribute

The following attribute is associated with each environment handle, and affects the behaviour of CLI functions in that environment:

SQL_ATTR_OUTPUT_NTS (type: INTEGER) (settable)

This attribute controls the implementation’s use of null termination in output arguments. (See also String Termination on page 19.)

If this attribute has the value SQL_TRUE, then the implementation uses null termination to indicate the length of output character strings. (This setting is appropriate for programs written in C.) If this attribute has the value SQL_FALSE, then the CLI implementation does not use null-termination. (COBOL applications typically must specify this behaviour.)

The initial value is SQL_TRUE.

The CLI functions affected by this attribute are all functions called for the environment (and for any connection allocated under the environment) that have character-string parameters.

An implementation may define additional environment attributes.

The application can specify the value of any settable environment attribute by calling SetEnvAttr() and can obtain the current attribute value by calling GetEnvAttr().
3.3 Connection

A connection is a method of gaining access to a particular database (metadata plus data). In client/server implementations, a connection represents the application’s access to a particular server. Associated with the connection handle is any data that must be global to the server; namely, general status information, transaction state and certain diagnostic information.

Any connection allocated is within the context of a single environment.

An application can be connected to several servers at the same time, and can establish several distinct connections to the same server. Only one connection is current at any time. All other connections are dormant.

A connection is active if transaction work is being performed over it. This is defined in Section 4.9 on page 63.

Transaction semantics in the case where work is simultaneously pending over several connections are discussed in Section 4.9 on page 63.

Connection Attributes

The application allocates a connection handle before specifying a server to which to connect. Each connection attribute has one of the following characteristics:

(CA1)
(Reserved for future use.) The application can manipulate the attribute at any time that the connection handle is allocated. For example, this characteristic will apply to future attributes that influence the selection of servers or the process of connecting to a server.

(CA2)
The application can manipulate the attribute only during the time that it is actually connected to a server, since the value of the attribute varies among servers. When not connected (that is, before the application has specified the next server to which to connect), this attribute is unavailable.

(CA3)
(Reserved for future use.) The settable attribute not only varies among servers, as in (CA2) above, but changing its value incompatibly with the connected server causes the connection to immediately fail.

The following attributes are associated with each connection handle:

SQL_ATTR_AUTO_IPD (type: INTEGER) (read-only) (CA2)
If this attribute has the value SQL_TRUE, then the CLI implementation is capable of populating an implementation parameter descriptor as the result of a call to \texttt{Prepare}().

This attribute is initially set during \texttt{Connect()} and its initial value is implementation-defined.

SQL_ATTR_METADATA_ID (type: INTEGER) (settable) (CA2)
This value is used as the default for the attribute by the same name at the statement level (see Statement Attributes on page 30).

This attribute is initially set during \texttt{Connect()} and its initial value is SQL_FALSE.

An implementation may define additional connection attributes.

The application can specify the value of any settable connection attribute by calling \texttt{SetConnectAttr()} and can obtain the current attribute value by calling \texttt{GetConnectAttr()}. 
3.4 Statement

A statement handle refers to a data structure that tracks the execution of a single SQL statement. Associated with the statement handle is information on the statement’s execution: dynamic arguments, bindings for dynamic parameters and columns, result values and status information.

When a statement is executed repeatedly, the information associated with the handle reflects the most recent execution of the statement.

The application can reuse a statement handle to prepare a different SQL statement. However, if the application has specified dynamic arguments or bindings for dynamic parameters, they persist and may be inappropriate in number or type for the new SQL statement (see Section 4.3 on page 46).

Any statement handle allocated is within the context of a single connection (and thus within a single environment).

Limitations

There may be an implementation-defined limit on the number of concurrent activities. This is the number of statement handles that may simultaneously be interacting with the server. This limit can be obtained as described in `GetInfo()` on page 176.

Statement Attributes

The following attributes are associated with each statement handle, and affect the behaviour of CLI functions on that statement:

SQL_ATTR_APP_ROW_DESC (type: INTEGER) (settable)

The value of this attribute is a descriptor handle (see Section 3.5 on page 32) that serves as the application row descriptor for subsequent fetches on the statement handle. If the attribute’s value is SQL_NULL_HDESC, there is no application row descriptor associated with the statement handle.

The initial value of this attribute is a descriptor implicitly allocated when the statement is allocated.

SQL_ATTR_APP_PARAM_DESC (type: INTEGER) (settable)

The value of this attribute is a descriptor handle (see Section 3.5 on page 32) that serves as the application parameter descriptor for subsequent calls to `Execute()` and `ExecDirect()` on the statement handle. If the attribute’s value is SQL_NULL_HDESC, there is no application parameter descriptor associated with the statement handle.

The initial value of this attribute is a descriptor implicitly allocated when the statement is allocated.

SQL_ATTR_CURSOR_SCROLLABLE (type: INTEGER) (settable)

The application sets this attribute to specify what level of support for scrollable cursors it requires. Setting this attribute affects subsequent calls to `ExecDirect()` and `Execute().` The attribute’s value is one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_FALSE</td>
<td>Scorable cursors are not required on the statement handle; if the application calls <code>FetchScroll()</code> on this handle, the only mode it will specify is SQL_FETCH_NEXT (which gives the same effect as <code>Fetch()</code>).</td>
</tr>
<tr>
<td>SQL_TRUE</td>
<td>Scorable cursors are required on the statement handle; when calling <code>FetchScroll()</code>, the application may specify any valid value of <code>FetchOrientation</code>, achieving cursor...</td>
</tr>
</tbody>
</table>
positioning in modes other than the sequential mode.

The initial value of the attribute is SQL_FALSE.

SQL_ATTR_CURSOR_SENSITIVITY (type: INTEGER)
This attribute's value is one of the following:

SQL_UNSPECIFIED
It is unspecified whether cursors on the statement handle make visible the changes made to a result set by another cursor. Cursors on the statement handle may make visible none, some, or all of such changes.

SQL_INSENSITIVE
All cursors on the statement handle show the result set without reflecting any changes made to it by any other cursor.

SQLSENSITIVE
All cursors on the statement handle make visible all changes made to a result set by another cursor.

Setting this attribute affects subsequent calls to ExecDirect() and Execute(). An application can read back the value of this attribute to obtain its initial state or its state as most recently set by the application.

The initial value of this attribute is SQL_UNSPECIFIED.

SQL_ATTR_IMP_ROW_DESC (type: INTEGER) (read-only)
The value of this attribute is a descriptor handles (see Section 3.5 on page 32) that serves as the implementation row descriptor.

The initial value of this attribute is a descriptor implicitly allocated when the statement is allocated.

SQL_ATTR_IMP_PARAM_DESC (type: INTEGER) (read-only)
The value of this attribute is a descriptor handles (see Section 3.5 on page 32) that serves as the implementation parameter descriptor.

The initial value of this attribute is a descriptor implicitly allocated when the statement is allocated.

SQL_ATTR_METADATA_ID (type: INTEGER) (settable)
This attribute has one of the following values:

SQL_TRUE
This specifies that the metadata functions (see Section 4.6 on page 53) treat certain arguments as identifiers. This means that the values are folded to upper case unless they are delimited (by enclosing the value in double quotes).

SQL_FALSE
This specifies that the metadata functions do not treat arguments as identifiers and enables treatment as pattern-values in the cases specified in Qualification by Pattern-values on page 55.

This attribute is initially set when the statement is allocated and its initial value is that of the connection attribute by the same name (see Connection Attributes on page 29).

An implementation may define additional statement attributes.

The application can specify the value of any settable statement attribute by calling SetStmtAttr() and can obtain the current attribute value by calling GetStmtAttr().
3.5 Descriptor

A descriptor handle refers to a data structure that holds information about either columns or dynamic parameters. A CLI descriptor is analogous to the SQL descriptor area in embedded SQL. “Descriptor” in this document means the CLI data structure, not the data structure from embedded SQL.

3.5.1 Types of Descriptor

A descriptor is used to describe one of the following:

1. A set of zero or more dynamic parameters. A parameter descriptor can be used for:
   — the application parameter buffer, which contains the dynamic arguments as set by the application
   — the implementation parameter buffer, which contains the same dynamic arguments after any data conversion the application may specify.*

   The application must operate on an application parameter descriptor before executing any SQL statement that contains dynamic parameter markers. The application may specify different data types from those in the implementation parameter descriptor to achieve data conversion of dynamic arguments.

2. A single row of database data. A row descriptor can be used for:
   — the implementation row buffer, which contains the row from the database*
   — the application row buffer, which contains the row, following any data conversion the application may specify, in which form the data is presented to the application.

   The application operates on the application row descriptor in any case where column data from the database must appear in application variables. The application may specify different data types from those in the implementation row descriptor to achieve data conversion of column data.

The following table summarises the descriptor types:

<table>
<thead>
<tr>
<th></th>
<th>Rows</th>
<th>Dynamic Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Buffer</td>
<td>Application Row</td>
<td>Application Parameter</td>
</tr>
<tr>
<td></td>
<td>Descriptor</td>
<td>Descriptor</td>
</tr>
<tr>
<td>Implementation Buffer</td>
<td>Implementation Row</td>
<td>Implementation Parameter</td>
</tr>
<tr>
<td></td>
<td>Descriptor</td>
<td>Descriptor</td>
</tr>
</tbody>
</table>

Table 3-1 The Four Types of Descriptor

For either the parameter or row buffers, if the application specifies different data types in corresponding records of the implementation and application descriptors, the CLI implementation performs data conversion when it uses the descriptors. For example, it may convert numeric and date/time values to character-string format. For valid combinations and

* The implementation buffers are conceptually the data as written to, or read from, the database. However, X/Open does not specify the stored form of database data, and an implementation (server) could perform additional conversion on the data from its form in the implementation buffer.
their effects, see Section 4.8.3 on page 60.

A descriptor may perform different roles. Different statements can share any descriptor that the application explicitly allocates. A row descriptor in one statement can serve as a parameter descriptor in another statement.5

It is always known whether a given descriptor is an application descriptor or an implementation descriptor, even if the descriptor has not yet been used in a database operation. For the descriptors that the implementation automatically allocates, the implementation records the predefined role relative to the statement handle. Any descriptor the application allocates using AllocHandle() is an application descriptor.

3.5.2 Header Fields of the Descriptor

A descriptor contains a single copy of the fields listed below, as well as zero or more descriptor records,6 as described in the following section.

**COUNT**

The COUNT field specifies how many records contain data. In general, whichever component (application or CLI implementation) sets the data structure must also set the COUNT field to show how many records are significant.

Unlike embedded SQL, an application does not have to specify, when it allocates an instance of this data structure, how many records to reserve room for. As the application specifies the contents of records, the CLI implementation takes any required action to ensure that the descriptor handle refers to a data structure of adequate size.

**ALLOC_TYPE**

The ALLOC_TYPE field specifies whether the descriptor was allocated automatically by the CLI implementation or explicitly by the application. The application can obtain but not modify this field.

3.5.3 Fields of the Descriptor Records

Each record contains the following fields, which are defined in the X/Open SQL specification:

- **TYPE**
- **LENGTH**
- **PRECISION**
- **SCALE**
- **NULLABLE**
- **NAME**
- **UNNAMED**

Since dynamic parameters are always nullable and do not have names, in an implementation parameter descriptor, the NULLABLE field always has the value SQL_NULLABLE and the UNNAMED field always has the value SQL_UNNAMED.

5. By reusing a row descriptor that contains a fetched row of a table as a parameter descriptor of an INSERT statement, an application could copy rows between tables without specifying copying of the data at the application level. However, an application can copy rows between different databases in this way only if the implementation supports simultaneous access to multiple connections, because the descriptor is valid only while connected.

6. These records correspond to the item descriptor areas in the SQL descriptor area of embedded SQL.
Each record also has the following fields:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME_INTERVAL_CODE</td>
<td>An integer date/time subcode whose value distinguishes the types DATE, TIME and TIMESTAMP.</td>
</tr>
<tr>
<td>OCTET_LENGTH</td>
<td>The length in octets of a character-string data type. Depending on the data type, this is the length or maximum length of the string. This value always excludes any null terminator.</td>
</tr>
</tbody>
</table>

For values whose type is TIME or TIMESTAMP, the PRECISION field is the precision of the fractional seconds component, in decimal digits. For values whose type is DATE, TIME or TIMESTAMP, the LENGTH field is the length in characters of the character-string representation of the date/time value (see Transfers with Conversion to/from String on page 62) and OCTET_LENGTH is the length in octets of that character string representation.

Each record also contains the following fields, which are deferred fields (see Section 3.5.5 on page 36) and correspond to certain fields defined in the X/Open SQL specification:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDICATOR_PTR</td>
<td>Corresponds to the INDICATOR field in the item descriptor area of embedded SQL.</td>
</tr>
<tr>
<td>DATA_PTR</td>
<td>Corresponds to the DATA field in the item descriptor area of embedded SQL.</td>
</tr>
<tr>
<td>OCTET_LENGTH_PTR</td>
<td>Indicates the length in octets of a dynamic argument (see Section 4.3 on page 46) or of a bound column value (see Section 4.5 on page 50).</td>
</tr>
</tbody>
</table>

The data type of each field is specified in SetDescField() on page 205.

### 3.5.4 Operations on Descriptors

#### Automatic Allocation/Freeing

When an application allocates a statement handle, the implementation automatically allocates one set of four descriptors. The application can obtain the handles of these automatically-generated descriptors as attributes of the statement handle. When the application frees the statement handle, the implementation frees all automatically-generated descriptors on that handle.

---

7. The deferred fields, which an application associates with its own variables, are present but not used in implementation descriptors.

8. The implementation has the option of deferring allocation of any descriptor until the point at which it is actually used.
Explicit Allocation/Freeing

The application can explicitly allocate an application descriptor on a connection at any time it is actually connected to a database. By specifying that descriptor handle as an attribute of a statement handle using `SetStmtAttr()`, the application directs the implementation to use that descriptor in place of the respective automatically-generated application descriptor. (The application cannot specify alternative implementation descriptors.)

The application can associate an explicitly-allocated descriptor with more than one statement. The application can free such a descriptor explicitly, or implicitly by freeing its connection.

Obtaining a Descriptor Handle

The application obtains the handle of any explicitly-allocated descriptor as an output argument of the call to `AllocHandle()`. The handle of an automatically-generated descriptor is available by calling `GetStmtAttr()`.

Initialisation of Fields

When an application row descriptor record is first used, the initial value of its TYPE field is SQL_DEFAULT. This provides for a standard treatment of database data for presentation to the application (see Default Transfers on page 60). The application may specify different treatment of the data by modifying fields of the descriptor record.

Access to Fields

The application can call `GetDescField()` to obtain a single field of a descriptor record. `GetDescField()` gives the application access to all the descriptor fields defined in the X/Open SQL specification. `GetDescField()` returns one field per call. The function is extensible, using additional argument values, to return future or implementation-defined information.

To modify fields of a descriptor, the application can call `SetDescField()`, an extensible function that sets a single descriptor field per call.\(^9\)

When setting fields individually, the application should follow a sequence as defined in `SetDescField()` on page 205. Setting some fields causes the CLI implementation to set other fields. These cases, directly analogous to cases defined in the X/Open SQL specification, ensure that a descriptor is always ready to use once the application has specified a data type. When the application sets the DATA_PTR field, the implementation checks that other fields that specify the type are valid and consistent (see the General Diagnostic in Section 5.4.9 on page 75).

The `CopyDesc()` function copies all information except ALLOC_TYPE from one descriptor to another.

More information on operations on parameter descriptors, in context, appears in Section 4.3 on page 46. More information on row descriptors appears in Section 4.5 on page 50.

---

\(^9\) The application cannot set the NAME, NULLABLE and UNNAMED fields. They are not used for parameter descriptors; in row descriptors, they contain column properties that applications cannot override.

The `GetData()` function also modifies the TYPE field in an application row descriptor.
3.5.5 Deferred Fields

The DATA_PTR, INDICATOR_PTR and OCTET_LENGTH_PTR fields of a descriptor record are deferred. When the application sets these fields by calling SetDescField(), the implementation does not use the current value of the application variables, but saves the addresses of the variables in the descriptor for a deferred effect as follows:

- For an application parameter descriptor, the implementation uses the contents of the variables at the time of the call to ExecDirect() or Execute().
- For an application row descriptor, the implementation assigns values to the variables at the time of the fetch.

When a descriptor is allocated, the deferred fields of each descriptor record initially have a null value. The meaning of the null value is as follows:

- If DATA_PTR has the null value, the record is unbound (see Section 3.5.6 on page 37).
- If INDICATOR_PTR has the null value:
  - For an application parameter descriptor, there is no indicator information for the descriptor record. This means the application cannot use the buffer to specify null dynamic arguments.
  - For an application row descriptor, a null INDICATOR_PTR prevents the implementation from returning indicator information for that column. (As in embedded SQL, the implementation needs an indicator to report the fetch of a null value; in this case, failure to bind INDICATOR_PTR is an error.)
- If OCTET_LENGTH_PTR has the null value:
  - For an application parameter descriptor, OCTET_LENGTH_PTR indicates the length in octets of character-string dynamic arguments. A null value directs the implementation to assume the string is null-terminated. (Setting and changing OCTET_LENGTH_PTR for character-string dynamic arguments is discussed in Section 4.3 on page 46. If TYPE does not indicate a character-string dynamic argument, OCTET_LENGTH_PTR is ignored.)
  - For an application row descriptor, the implementation does not return length information for that column.

In a language that supports pointers, the application can obtain the value of a deferred field by calling GetDescField(). Such a call returns not the actual data but a pointer to the associated application variable (or the null pointer if the field is not associated with an application variable). A routine that takes as an argument a descriptor handle could call GetDescField() to obtain pointers to the data, indicator or length of any descriptor record.

Once the application has associated a deferred field with an application pointer, it can specify a different application pointer, or specify the null pointer to return the deferred field to the initial, unbound state. To reuse the same application descriptor with a different number and position of bound records, an application can free the descriptor and allocate a new one, or overwrite the previous bindings and change the COUNT field.

The application must not deallocate or discard variables used for deferred fields between the time it associates them with the fields and the time the CLI implementation reads or writes them.
3.5.6 Bound Descriptor Records

When the application sets the DATA_PTR field of a descriptor record, so that it no longer contains the null value, the record is said to be bound.

If the descriptor is an application parameter descriptor, which describes dynamic arguments to an SQL statement, then each bound record constitutes a bound parameter. The application must bind a parameter for each dynamic parameter marker in the SQL statement before executing the statement (see Section 4.3 on page 46).

If the descriptor is an application row descriptor, which describes a row of database data, then each bound record constitutes a bound column. The application retrieves data from bound and unbound columns using different methods (see Section 4.5 on page 50).
3.6 Cursors

A cursor is a movable pointer into a derived table by which the application can retrieve, update and delete rows. (This process is described in Section 4.5 on page 50.) Cursors have names by which the application can reference them. Each cursor name within a given connection is unique. The scope of a cursor name is the connection in which it is defined.

In embedded SQL, a cursor is declared and named by the non-executable DECLARE CURSOR statement. In CLI, the application executes a cursor-specification (specifying a result set to which the cursor applies) without declaring a cursor. If the application has not already provided a cursor name for the statement handle, then the implementation generates a cursor name, starting with 'SQLCUR' or 'SQL_CUR' and guaranteed to be unique within the relevant connection, and executes an implicit DECLARE CURSOR statement.

The application can specify a cursor name and can retrieve any cursor name the implementation has generated. This lets the application generate text for positioned DELETE and UPDATE statements using that cursor (see Section 4.5.1 on page 52).
This chapter introduces the CLI functions and gives an overview of their use.

Table 4-1 on page 40 lists the functions available and gives section references for further information; for detailed information, see the appropriate reference manual page in Chapter 8.

Section 4.1 on page 42 provides flowcharts showing the basic flow of control for the use of CLI functions. Subsequent sections describe the usage of CLI functions, grouped by functional category. There are sections devoted to executing SQL statements, specifying dynamic arguments, analysing a prepared statement, fetching data or metadata, data conversion and transaction management.

The functions that return diagnostic information are discussed in Chapter 5.

**Types of CLI Function**

In some cases CLI provides multiple functions that achieve comparable effects. Some functions are designed for ease of use by application writers in common situations; others are designed for complete access to the underlying data structure and for extensibility to additional forms defined by X/Open in the future or by implementations. Functions that are part of such a pair are marked by (concise) or (extensible) as appropriate. For an overview of the concise CLI functions, see Chapter 6.

In addition, this document defines additional functions for compatibility with APIs commonly used in existing applications. These functions are deprecated. For an overview of the deprecated functions, see Chapter 7.

The interface overview in this chapter describes use of the extensible functions. Citation of a specific extensible function in this chapter does not imply that an application could not instead use a concise or deprecated function to achieve the same effect.
## Table 4-1 CLI Functions

<table>
<thead>
<tr>
<th>CLI Function</th>
<th>DE</th>
<th>Description</th>
<th>Overview Section</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Allocate and Deallocate</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AllocConnect()</td>
<td>DE</td>
<td>Allocate a connection handle.</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>AllocEnv()</td>
<td>DE</td>
<td>Allocate an environment handle.</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>AllocHandle()</td>
<td></td>
<td>Allocate a handle.</td>
<td>Section 3.1 on page 26</td>
</tr>
<tr>
<td>AllocStmt()</td>
<td>DE</td>
<td>Allocate a statement handle.</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>FreeConnect()</td>
<td>DE</td>
<td>Free a connection handle.</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>FreeEnv()</td>
<td>DE</td>
<td>Free an environment handle.</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>FreeHandle()</td>
<td></td>
<td>Free a handle.</td>
<td>Section 3.1 on page 26</td>
</tr>
<tr>
<td>FreeStmt()</td>
<td>DE</td>
<td>Free a statement handle.</td>
<td>Chapter 7</td>
</tr>
<tr>
<td><strong>Get and Set Attributes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetConnectAttr()</td>
<td></td>
<td>Get the value of a connection attribute.</td>
<td>Section 3.3 on page 29</td>
</tr>
<tr>
<td>GetConnectOption()</td>
<td>DE</td>
<td>Get the value of a connection attribute.</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>GetEnvAttr()</td>
<td></td>
<td>Get the value of an environment attribute.</td>
<td>Section 3.2 on page 28</td>
</tr>
<tr>
<td>GetStmtAttr()</td>
<td></td>
<td>Get the value of a statement attribute.</td>
<td>Section 3.4 on page 30</td>
</tr>
<tr>
<td>GetStmtOption()</td>
<td>DE</td>
<td>Get the value of a statement attribute.</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>SetConnectAttr()</td>
<td></td>
<td>Set a connection attribute.</td>
<td>Section 3.3 on page 29</td>
</tr>
<tr>
<td>SetConnectOption()</td>
<td>DE</td>
<td>Set a connection attribute.</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>SetEnvAttr()</td>
<td></td>
<td>Set an environment attribute.</td>
<td>Section 3.2 on page 28</td>
</tr>
<tr>
<td>SetStmtAttr()</td>
<td></td>
<td>Set a statement attribute.</td>
<td>Section 3.4 on page 30</td>
</tr>
<tr>
<td>SetStmtOption()</td>
<td>DE</td>
<td>Set a statement attribute.</td>
<td>Chapter 7</td>
</tr>
<tr>
<td><strong>Connection</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connect()</td>
<td></td>
<td>Open a connection to a server.</td>
<td></td>
</tr>
<tr>
<td>Disconnect()</td>
<td></td>
<td>Close a connection to a server.</td>
<td></td>
</tr>
<tr>
<td><strong>Descriptor Access</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CopyDesc()</td>
<td></td>
<td>Copy a descriptor.</td>
<td>Section 3.5 on page 32</td>
</tr>
<tr>
<td>GetDescField()</td>
<td></td>
<td>Get a descriptor field (extensible).</td>
<td>Section 3.5 on page 32</td>
</tr>
<tr>
<td>GetDescRec()</td>
<td></td>
<td>Get a descriptor record (concise).</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>SetDescField()</td>
<td></td>
<td>Set a descriptor field (extensible).</td>
<td>Section 3.5 on page 32</td>
</tr>
<tr>
<td>SetDescRec()</td>
<td></td>
<td>Set a descriptor record (concise).</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>ColAttribute()</td>
<td>DE</td>
<td>Get an implementation row descriptor field.</td>
<td>Chapter 7</td>
</tr>
<tr>
<td><strong>Executing SQL Statements</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BindParam()</td>
<td></td>
<td>Bind a dynamic parameter (concise).</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>ExecDirect()</td>
<td></td>
<td>Execute an SQL statement directly.</td>
<td>Section 4.2.2 on page 45</td>
</tr>
<tr>
<td>Execute()</td>
<td></td>
<td>Execute a prepared SQL statement.</td>
<td>Section 4.2.1 on page 44</td>
</tr>
<tr>
<td>GetCursorName()</td>
<td></td>
<td>Get the name of a cursor.</td>
<td>Section 3.6 on page 38</td>
</tr>
<tr>
<td>ParamData()</td>
<td></td>
<td>Address the next dynamic parameter requiring data.</td>
<td>Section 4.3.2 on page 48</td>
</tr>
<tr>
<td>Prepare()</td>
<td></td>
<td>Prepare an SQL statement for execution.</td>
<td>Section 4.2.1 on page 44</td>
</tr>
<tr>
<td>PutData()</td>
<td></td>
<td>Send part or all of a dynamic argument at execute time.</td>
<td>Section 4.3.2 on page 48</td>
</tr>
<tr>
<td>SetCursorName()</td>
<td></td>
<td>Set the name of a cursor.</td>
<td>Section 4.2.1 on page 44</td>
</tr>
<tr>
<td>SetParam()</td>
<td>DE</td>
<td>Synonym for BindParam()</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>CLI Function</td>
<td>Description</td>
<td>Overview Section</td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------------</td>
<td></td>
</tr>
<tr>
<td><strong>Receiving Results</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BindCol()</td>
<td>Bind a column in a result set (concise).</td>
<td>Chapter 6</td>
<td></td>
</tr>
<tr>
<td>CloseCursor()</td>
<td>Close a cursor.</td>
<td>Chapter 6</td>
<td></td>
</tr>
<tr>
<td>DescribeCol()</td>
<td>Describe a column of a result set (concise).</td>
<td>Chapter 6</td>
<td></td>
</tr>
<tr>
<td>Fetch()</td>
<td>Fetch the next row of a result set.</td>
<td>Section 4.5 on page 50</td>
<td></td>
</tr>
<tr>
<td>FetchScroll()</td>
<td>Move the cursor to, and fetch, a specified row of a result set.</td>
<td>Section 4.5 on page 50</td>
<td></td>
</tr>
<tr>
<td>GetData()</td>
<td>Retrieve one column of a row of the result set.</td>
<td>Section 4.5 on page 50</td>
<td></td>
</tr>
<tr>
<td>NumResultCols()</td>
<td>Get the number of columns in a result set (concise).</td>
<td>Chapter 6</td>
<td></td>
</tr>
<tr>
<td>RowCount() DE</td>
<td>Get the number of rows affected by a statement.</td>
<td>Chapter 7</td>
<td></td>
</tr>
<tr>
<td><strong>Metadata Functions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Columns()</td>
<td>Get column information.</td>
<td>Section 4.6 on page 53</td>
<td></td>
</tr>
<tr>
<td>SpecialColumns()</td>
<td>Get unique row identifier for a table.</td>
<td>Section 4.6 on page 53</td>
<td></td>
</tr>
<tr>
<td>Statistics()</td>
<td>Get index information for a base table.</td>
<td>Section 4.6 on page 53</td>
<td></td>
</tr>
<tr>
<td>Tables()</td>
<td>Get table information.</td>
<td>Section 4.6 on page 53</td>
<td></td>
</tr>
<tr>
<td><strong>Introspection</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataSource()</td>
<td>Get a list of server names.</td>
<td>Section 4.7 on page 57</td>
<td></td>
</tr>
<tr>
<td>GetFunctions()</td>
<td>Determine whether a function is supported.</td>
<td>Section 4.7 on page 57</td>
<td></td>
</tr>
<tr>
<td>GetInfo()</td>
<td>Get information about the CLI implementation and the currently-connected</td>
<td>Section 4.7 on page 57</td>
<td></td>
</tr>
<tr>
<td></td>
<td>server.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetTypeInfo()</td>
<td>Get information on server data types.</td>
<td>Section 4.7 on page 57</td>
<td></td>
</tr>
<tr>
<td><strong>Transaction Control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndTran()</td>
<td>Commit or roll back a transaction.</td>
<td>Section 4.9 on page 63</td>
<td></td>
</tr>
<tr>
<td>Transact() DE</td>
<td>Commit or roll back a transaction.</td>
<td>Section 4.9 on page 63</td>
<td></td>
</tr>
<tr>
<td><strong>Diagnostic Information</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetDiagField()</td>
<td>Return diagnostic information (extensible).</td>
<td>Section 5.2 on page 69</td>
<td></td>
</tr>
<tr>
<td>GetDiagRec()</td>
<td>Return diagnostic information (concise).</td>
<td>Section 5.2 on page 69</td>
<td></td>
</tr>
<tr>
<td>Error() DE</td>
<td>Return diagnostic information.</td>
<td>Chapter 7</td>
<td></td>
</tr>
<tr>
<td><strong>Function Cancellation</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cancel()</td>
<td>Attempt to cancel a CLI operation.</td>
<td>Section 4.3.2 on page 48</td>
<td></td>
</tr>
</tbody>
</table>
4.1  **Overview of Control Flow**

The following diagrams show the basic flow of control for the use of CLI functions. These diagrams assume (and do not illustrate) that the application performs error checking where appropriate using the `GetDiagField()` or `GetDiagRec()` functions.

For more detailed information concerning control flow and function sequencing rules, refer to the state transition tables in Appendix B.

4.1.1  **Basic Control Flow**

The following figure shows the initiation sequence, the termination sequence and an overview of transaction completion.

![Control Flow Diagram](image)

**Figure 4-1** Initiation, Termination and Transaction Completion
4.1.2 Example Control Flow for SQL Statement Processing

Figure 4-2 shows typical control flow for processing SQL statements, including the allocation and release of a statement handle.

While this is the basic control flow for SQL statements executed interactively, there are other valid sequences, such as modifying the buffer descriptor between successive fetches.

![Flowchart](image)

**Figure 4-2** Example Control Flow for Statement Processing
4.2 Executing SQL Statements

There are two different ways in CLI to specify and execute SQL statements:

1. Call Prepare() and then call Execute().
2. Call ExecDirect().

These methods roughly correspond to PREPARE/EXECUTE and EXECUTE IMMEDIATE respectively in embedded SQL. Method 1 allows the prepared statement to be executed multiple times without duplicating the work of preparing the statement.

The application can use either method to execute any statement that the X/Open SQL specification allows in the PREPARE statement. An important addition is that CLI also allows a cursor-specification to be input to either method. Embedded SQL requires the use of the OPEN statement to execute a cursor-specification. Moreover, both CLI methods allow the use of dynamic parameters; in embedded SQL, dynamic parameters are not permitted in EXECUTE IMMEDIATE.

4.2.1 Prepare() and Execute()

The application prepares an SQL statement for execution and possible repetition by taking the following steps:

1. It prepares the SQL statement by calling Prepare().
2. It defines dynamic arguments (see Section 4.3 on page 46).
3. If the SQL statement is a cursor-specification, the application can declare a cursor name by calling SetCursorName(). Otherwise, the implementation generates a cursor name. CLI uses cursor names only in positioned UPDATE and DELETE statements.
4. It executes the prepared statement by calling Execute(). After execution, the application can assign new dynamic arguments and execute the statement again.

This method separates the specification of the SQL text from its execution. It is advisable when any of the following is true:

- The application must gain access to row descriptors (that is, column names, types and lengths) after preparing and before executing the statement.
- The application must gain access to parameter descriptors after preparing and before executing the statement.\(^{10}\)
- The application wants highest performance in repeated executions of the same SQL statement text. Preparing the text typically compiles the SQL statement; the faster, compiled form is used by all calls to Execute().

---

\(^{10}\) Portable applications that require a particular data type conversion for dynamic parameters should use Prepare() and Execute(). On some implementations, Prepare() populates the implementation parameter descriptor (see Section 4.3.1 on page 47). The application can specify different field values after calling Prepare().
4.2.2 ExecDirect()

Calling ExecDirect() is preferable for a single execution of an SQL statement. Use this method when all of the following are true:

- The application does not need to gain access to row descriptors until after execution of a cursor-specification.
- The SQL statement is executed only once.

COMMIT and ROLLBACK are not preparable statements, and thus cannot be input to either Prepare() or ExecDirect(). Transaction completion in CLI is discussed in Section 4.9 on page 63.
4.3 Specifying Dynamic Arguments

An SQL statement executed under CLI may contain question mark (?) characters. Each of these is a *dynamic parameter marker*. A dynamic parameter marker represents a position in the SQL statement (a *dynamic parameter*) where the application is to provide a value (a *dynamic argument*).

The application must provide a dynamic argument for each dynamic parameter in the SQL statement before it executes the statement. Information for each parameter remains in effect until overridden, or until the application drops the statement handle or application parameter descriptor handle. To prepare a statement with the same statement handle that has a different number of dynamic parameter markers, the application must revise the affected descriptors or allocate new ones in order to specify the correct number and type of dynamic arguments.

The application provides dynamic arguments by using an application parameter descriptor (see Section 3.5 on page 32). The application can use the automatically-generated application parameter descriptor (and obtain its handle by calling `GetStmtAttr()`), or can allocate a separate descriptor by calling `AllocHandle()` and declare it the application parameter descriptor of the statement by calling `SetStmtAttr()`.

Each record of the application parameter descriptor describes one dynamic argument. Within each record, the application sets various fields, binding some of them to application variables:

- It sets the DATA_PTR field to a variable that, at execute time, will contain the value of the dynamic argument.
- It sets other fields to define the data type, type attributes and indicator information.

If the data type is character string, the application sets the LENGTH field to the maximum number of characters of the parameter and sets the OCTET_LENGTH_PTR field to a variable that, at execute time, will describe the length in octets of the dynamic argument.\(^{11}\) (The OCTET_LENGTH field of parameter descriptors is not used.)

The application can call `BindParam()` or `SetDescRec()` to make a complete specification for a dynamic parameter, or can call `SetDescField()` to set individual descriptor fields. Using `BindParam()`, the application can specify a number of characters for the dynamic argument below the maximum allowed by the data type.

---

\(^{11}\) If OCTET_LENGTH_PTR is not bound to an application variable, the dynamic argument must be null-terminated. If OCTET_LENGTH_PTR is bound and the variable contains the value SQL_NTS[L], the dynamic argument must be null-terminated. Otherwise, the application variable, at execute time, must contain the length in octets of the dynamic argument, excluding any null terminator.

This rule allows applications using null-terminated strings as dynamic arguments to avoid using the OCTET_LENGTH_PTR field.
If the application executes the SQL statement repeatedly without changing the application parameter descriptor, then the implementation uses the same pointer to storage on each execution. However, the application may change the information at that storage location from one execution to another, resulting in different dynamic arguments. (If the length of a dynamic argument changes from one execution to another, the application must ensure that, at each execution, the respective descriptor record’s OCTET_LENGTH_PTR field, and any application variable it is bound to, continues to define correctly the length of the dynamic argument, as described above.)

4.3.1 Use of Implementation Parameter Descriptor

If the application does not require type conversion of the dynamic arguments in its variables, it need not reference the implementation parameter descriptor.

On some implementations, preparing a statement sets the fields of an implementation parameter descriptor to indicate the data type and type attributes of the dynamic parameters. (This corresponds to support for the DESCRIBE INPUT statement of embedded SQL.) The application can configure dynamic arguments based on this information by following this sequence:

- Prepare the statement by calling Prepare().
- Call CopyDesc() to copy the fields of the automatically-generated implementation parameter descriptor to the application parameter descriptor. (The application obtains the handles of any descriptor it did not allocate by calling GetStmtAttr()).
- Bind the deferred fields of the application parameter descriptor to application variables as required. If desired, change data type and type attribute fields (from the values copied from the implementation parameter descriptor) to specify any permitted type conversion of dynamic arguments.
- Execute the statement by calling Execute().

The application must explicitly set fields of the implementation parameter descriptor if either of the following is true:

- The application uses ExecDirect() instead of Prepare() and Execute() but requires the implementation to perform data conversion of any dynamic parameters. In this case, the application must set the implementation parameter descriptor explicitly before calling ExecDirect(). The implementation parameter descriptor is unaffected by a call to ExecDirect().
- The implementation does not populate the implementation parameter descriptor during a call to Prepare(). (The application can test for this feature by obtaining the SQL_ATTR_AUTO_IPD connection attribute.) In this case, the implementation does not modify the implementation parameter descriptor.
The application may also set fields in the implementation parameter descriptor in other cases, in order to achieve conversions of dynamic arguments from the type of data in the application parameter buffer.

### 4.3.2 Specifying Parameter Values at Execute Time

An application may specify values for specific bound parameters at execute time rather than providing values in memory for all bound parameters before calling `Execute()` or `ExecDirect()`. This is especially useful for passing sizable dynamic arguments in pieces.

To indicate that a bound parameter will be a data-at-execute parameter, the application does the following:

- Sets the OCTET_LENGTH_PTR field in the corresponding record of the application parameter descriptor to a variable that, at execute time, will contain the value `SQL_DATA_AT_EXEC`.
- If there is more than one such field, it sets each DATA_PTR field to some value that it will recognise as uniquely identifying the field in question.

(The application can make these settings directly by calling `SetDescField()` or `SetDescRec()`, or by providing suitable `StrLen_or_Ind` and `ParameterValue` arguments in a call to `BindParam()`)

When the application calls `ExecDirect()` or `Execute()`, if there are any data-at-execute parameters, the call returns `[SQL_NEED_DATA]`. The application responds as follows:

1. It calls `ParamData()` to advance to the first such parameter. This function returns `[SQL_NEED_DATA]` and provides the contents of the DATA_PTR field of the application parameter descriptor to identify the information required.
2. It calls `PutData()` to pass the actual data for the parameter. Long dynamic arguments can be sent in pieces by calling `PutData()` repeatedly.
3. It calls `ParamData()` again after it has provided the complete dynamic argument.

If more data-at-execute parameters exist, `ParamData()` returns `[SQL_NEED_DATA]` and the application repeats steps 2 and 3 above.

When no more data-at-execute parameters exist, `ParamData()` completes execution of the SQL statement. The `ParamData()` function produces a return value and diagnostics as the original `ExecDirect()` or `Execute()` statement would have produced.

The following flowchart illustrates this technique:
An example of this technique in the C language is provided in Section D.3 on page 272. While the data-at-execute dialogue is in progress, the only CLI functions the application can call are:

- The `ParamData()` and `PutData()` functions, as discussed above
- The `Cancel()` function, to cancel the data-at-execute dialogue and force an orderly exit from the loop shown above without executing the SQL statement
- The diagnostic functions.

Moreover, the application cannot end the transaction (see Section 4.9 on page 63) nor set any connection attribute that would have an impact on the treatment of the statement handle.
4.4 Analysing a Prepared Statement

After preparing a statement by calling `Prepare()`, or executing it by calling `ExecDirect()`, an application can call `GetDiagField()` to examine the string SQL_DIAG_DYNAMIC_FUNCTION field or integer SQL_DIAG_DYNAMIC_FUNCTION_CODE field, and thus determine the type of statement that was prepared. If the statement was a cursor-specification, the application can examine the implementation row descriptor by calling `GetDescField()` to determine the number of columns that will appear in the result set. The application can then retrieve information on any such column by further analysing the descriptor with `GetDescField()`. This analysis is especially useful if the statement text is generated at execute time (for example, typed by the user).

To determine the type of an executed SQL statement, see Section 5.3.1 on page 72. To obtain the count of rows that were affected by a searched DELETE, INSERT or searched UPDATE, see Section 5.3.2 on page 72.

4.5 Fetching Data

Executing a cursor-specification defines a result set. The application fetches individual rows of the result set by calling `Fetch()` or `FetchScroll()`. The `Fetch()` function always moves the cursor to the next row in sequence and fetches that row. The `FetchScroll()` function can move the cursor to any specified row of the result set and fetches that row. To use the features of `FetchScroll()` that provide modes of cursor movement other than the sequential mode, the application must use a statement attribute to enable scrollable cursors before preparing or executing the cursor-specification.

The application obtains data from a fetched row of the result set in one of the following ways:

- If it has used an application row descriptor to bind columns (see Section 3.5 on page 32), then column data is present in the application variables to which the respective columns are bound.
- If the columns are unbound, it can read database values by calling `GetData()`.
- It may use both the above techniques if some columns are bound and some are unbound.

Bound Columns

To bind columns to application variables, the application sets the DATA_PTR field of the corresponding records of the application row descriptor. The application sets other descriptor fields to define the data type, type attributes and variables that will hold indicator and length information.

The application can call `BindCol()` or `SetDescRec()` to make a complete specification for a column of the result set, or can call `SetDescField()` to set individual descriptor fields.

If the application row descriptor specifies different data types or type attributes from the implementation row descriptor, the CLI implementation performs type conversion on affected columns when it fetches the row (see Section 4.8.3 on page 60).
Unbound Columns

If the application elects not to bind any columns, it need not reference the row descriptor. It can obtain column data by simply calling `GetData()`.

Bound and Unbound Columns

If binding some but not all columns of a row, the application must reference the row descriptor, but need not specify any field of a record that pertains to an unbound column.

During a fetch, the implementation implicitly copies database information for bound columns to the application variables to which the columns are bound, and may perform type conversion of the bound column data, as described above.

For unbound columns following the highest-numbered bound column, portable applications obtain the column data by calling `GetData()` in ascending order of column number (from left to right). It is implementation-defined whether an application can obtain column data in a different sequence. It is implementation-defined whether column data for lower-numbered, unbound columns is available.12

The application can achieve type conversion of the column data by specifying in the call to `GetData()` either the desired target type or the value SQL_ARDB_TYPE, which means that the application row descriptor indicates the desired target type even though the column is unbound.

Length of Application Buffers

For bound columns, the application uses its knowledge of the column (perhaps discerned as described in Section 4.4 on page 50) to allocate the maximum memory the column value could occupy, in order to avoid truncation of the value. For unbound columns, the column can be arbitrarily long. If the length of the column value exceeds the length of the application’s buffer, a feature of `GetData()` lets the application use repeated calls to obtain the value of a single column of CHAR or VARCHAR type in pieces of manageable size.

---

12. The SQL_GETDATA_EXTENSIONS characteristic, available by calling `GetInfo()`, indicates whether the implementation supports these features. Implementations may omit these features so that the client does not have to buffer an entire row of data.
4.5.1 Updating Fetched Data

To update a row that has been fetched, the application uses two statement handles, one for the fetch and one for the update. The application calls `GetCursorName()` to obtain the cursor name used for the fetch. (The implementation generates this name for handles that pertain to `cursor-specifications`, but the application can specify a cursor name by calling `SetCursorName()` before preparing the `cursor-specification`.) The application generates text of a positioned UPDATE (or DELETE), including the same cursor name used by the fetch, and executes that SQL statement using the second statement handle. (The application cannot reuse the fetch statement handle to execute a positioned UPDATE; this would discard the result set fetched.)

Update during Fetch

A portable application should not try to update a row until it has fetched all necessary information from unbound columns of the row. If a fetched row is updated (by the same application or by some other process) and the application then calls `GetData()` to obtain additional unbound columns, the call to `GetData()` does one of the following:

- Obtains data including any intervening updates.
- Returns the value of the specified unbound columns as they existed at the time of the fetch.
- Returns an implementation-defined error.

This implementation variability does not exist for bound columns. At the time of the fetch, the implementation copies information for all bound columns to the application variables to which the columns are bound.

Cursor Name

A cursor name exists for a statement handle when the application successfully calls `SetCursorName()`. If the application has not called `SetCursorName()` before calling `Prepare()`, then `Prepare()` generates the cursor name.
4.6 Metadata Functions

An application may query a database’s metadata (for example, the names of tables or columns) as well as the data itself with the CLI metadata functions. They operate by returning to the application a result set through a statement handle. After calling these functions, the application can fetch individual rows of the result set, and can process the metadata as it would process column data from an ordinary fetch.

The X/Open SQL specification defines a number of system views, read-only views from which the application can retrieve metadata. If the implementation complies with both this specification and the X/Open SQL specification, another way to get metadata is to perform queries on the system views.

The metadata functions are conceptually the same (so, for instance, they have the same effect on the state tables) as the execution, using ExecDirect(), of a query on the X/Open system views. However, the result set of a metadata function does not necessarily have the same columns in the same sequence as the system views of the X/Open SQL specification. It is undefined how the CLI implementation actually obtains the metadata.

<table>
<thead>
<tr>
<th>Metadata Function</th>
<th>Conceptual Equivalent Embedded SQL Query on System Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>Columns()</td>
<td>SELECT...FROM COLUMNS</td>
</tr>
<tr>
<td>SpecialColumns()</td>
<td>no conceptual equivalent; see below</td>
</tr>
<tr>
<td>Statistics()</td>
<td>SELECT...FROM INDEXES (and see below)</td>
</tr>
<tr>
<td>Tables()</td>
<td>SELECT...FROM TABLES</td>
</tr>
</tbody>
</table>

Table 4-2 Metadata Functions: Embedded SQL Equivalent Queries

The Columns() and SpecialColumns() functions both return information about the columns of a specified table.

- The Columns() function returns a set of columns that would be returned by a query on the COLUMNS system view of embedded SQL. The columns are limited to those specified when an application created or altered the table. The query may be qualified by name.

- The SpecialColumns() function returns a set of columns that uniquely identifies a row from the specified table. The SpecialColumns() function can obtain information on pseudo-columns — that is, columns that an implementation may automatically create in addition to those the application specifies, such as a row identifier.

  For each row queried, SpecialColumns() returns in its result set a COLUMN_NAME column. This name is intended to be the name that the application would use in a cursor-specification or WHERE clause to specify most efficiently the row in question.

The Statistics() function returns information available from the INDEXES system view and may also return other information on indexes unavailable through INDEXES.

Result Set Definition

The manual pages for the metadata functions define the result set that the functions return. The X/Open columns of the result set always occur first and in the specified order. There may be additional, implementation-defined columns in the result set. Any such columns occur following the X/Open columns.
Naming

As described in Section 2.4.3 on page 22, the application can qualify the names of tables and other objects by prepending a schema name and, on some implementations, a catalog name. An implementation that does not support the use of catalog names rejects a metadata function argument that would restrict the result set based on catalog name.

Constraints

Different users may be entitled to view different amounts of the metadata. The metadata functions may thus return different result sets to different CLI users. These constraints on metadata are identical to those defined in the X/Open SQL specification for the corresponding system view.

4.6.1 Parameters of Metadata Functions

The following table summarises the treatment of certain string input parameters of the metadata functions. The OA, ID, VL and PV legends are explained in the following sections.

<table>
<thead>
<tr>
<th>Metadata Function</th>
<th>Parameter Name</th>
<th>Catalog-Name</th>
<th>Schema-Name</th>
<th>Table-Name</th>
<th>Table-Type</th>
<th>Column-Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Columns()</td>
<td>OA/ID</td>
<td>PV/ID</td>
<td>PV/ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpecialColumns()</td>
<td>OA/ID</td>
<td>OA/ID</td>
<td>OA/ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statistics()</td>
<td>OA/ID</td>
<td>OA/ID</td>
<td>OA/ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tables()</td>
<td>PV/ID</td>
<td>PV/ID</td>
<td>PV/ID</td>
<td>VL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4-3 Treatment of Parameters of Metadata Functions

Ordinary Arguments

Parameters with the OA legend in Table 4-3 are treated as ordinary arguments if the SQL_ATTR_METADATA_ID attribute of the relevant statement handle is SQL_FALSE. (Otherwise, see Treatment as Identifier.)

The corresponding argument is taken literally and the case of letters is significant.

The argument does not qualify a query but rather specifies the metadata desired. The application cannot pass a null pointer or otherwise avoid making this specification except when specifying CatalogName in a case where the implementation does not support catalog names.

Treatment as Identifier

Parameters with the ID legend in Table 4-3 are treated as identifiers if the SQL_ATTR_METADATA_ID attribute of the relevant statement handle is SQL_TRUE.

Identifiers, including delimited identifiers, are discussed in Section 2.3.8 on page 21. A metadata function interprets an argument as an identifier by conceptually performing the following operations:

- It removes any leading and trailing spaces.

- Then, if the first character is a double quote and the last character is a double quote, the argument is a delimited identifier and the characters between the first and the last character constitute the value of the argument. Otherwise, the entire string, with any lower-case letters
converted to the corresponding upper-case letter, is the value of the argument.

The argument does not qualify a query but rather specifies the metadata desired. The application cannot pass a null pointer or otherwise avoid making this specification except when specifying \textit{CatalogName} in a case where the implementation does not support catalog names.

\textbf{Qualification — General Rules}

When an argument qualifies a metadata query, it excludes non-matching rows from the result set, as though the equivalent embedded SQL queries in the above table were qualified by a \texttt{WHERE} clause.

If the application passes a null pointer as a string input argument, it inhibits qualification of the query and does not restrict the result set.

When an application uses more than one qualification of a metadata function, the qualifications apply conjunctively, as though the \texttt{WHERE} clauses in the equivalent embedded SQL query were joined by \texttt{AND}. A row appears in the result set only if it meets all the qualifications.

\textbf{Qualification by Value List}

For parameters with the \textbf{VL} legend in Table 4-3 on page 54, the corresponding argument can contain a value list. This is a list of one or more acceptable values (server attributes or table types). Acceptable values are defined on the respective reference manual page.

Values must be specified in upper case. Multiple values are separated by commas. The value \% matches any result and thus keeps that argument from qualifying the result set.* Multiple values in a value list are applied disjunctively; a row is qualified to appear in the result set if it matches any of the values in the list.

Redundant value lists (lists in which a value is repeated, or in which \% is present along with one or more discrete values) are valid but do not produce multiple copies of rows in the result set. The implementation ignores values that it does not recognise and processes the value list as though any such values were not present.

Quotes are not used in value lists, either to enclose the value list or to enclose a single list item.

\textbf{Qualification by Pattern-values}

Parameters with the \textbf{PV} legend in Table 4-3 on page 54 are treated as pattern-values if the SQL\_ATTR\_METADATA\_ID attribute of the relevant statement handle is SQL\_FALSE. (Otherwise, see \textit{Treatment as Identifier}).

The corresponding arguments function as a \textit{pattern-value} does in the X/Open SQL specification. That is:

\begin{itemize}
  \item The underscore character (\_) stands for any single character.
  \item The percent character (\%) stands for any sequence of zero or more characters.*
  \item All other characters stand for themselves.
\end{itemize}

The case of a letter is significant.

\begin{itemize}
  \item Passing a null pointer or a zero-length string as the argument has the same effect.
\end{itemize}
In embedded SQL, an application would achieve comparable qualification by using a LIKE predicate as follows, where the question mark is a dynamic parameter marker:

\textit{column} \text{ LIKE } ?

Some implementations support an \textit{escape character}. The escape character is not treated as part of the pattern, and causes the following character in \textit{pattern-value} to be taken literally (without the special effects for _ and % described above). The escape character itself can be specified in a pattern by including it twice in succession. An application can extract the SQL\_SEARCH\_PATTERN\_ESCAPE characteristic by calling \textit{GetInfo()}. This indicates whether an escape character is supported and, if so, what the escape character is.
4.7 Introspection

An application may use the introspection functions described below to find out about the characteristics and capabilities of both a CLI implementation and servers enabled through that implementation. These functions are of particular use to generic applications that are written to customise themselves to adapt to and to take advantage of the facilities provided by a range of CLI implementations and DBMS (Database Management System) servers.

The following introspection functions are defined:

- `DataSource()`
  - Returns a list of server names to which the application may be able to connect, and may return a description of each such server.

- `GetFunctions()`
  - Returns a true/false result that indicates whether the implementation supports a specified CLI function.

- `GetInfo()`
  - Obtains implementation-defined limit values. The application specifies one limit, for which it seeks the implementation-defined value, on each call to `GetInfo()`. Certain limits are enforced by the client or server. The information available from `GetInfo()` is in addition to the information obtained using the analogous SERVER_INFO system view of embedded SQL.

- `TypeInfo()`
  - Obtains information about data types supported by a server, in the form of a result set.

4.8 Data Conversion

Data in CLI moves between application buffers and implementation buffers (which the CLI implementation exchanges with the database). Application descriptors and implementation descriptors describe the data in the respective buffer. (Section 4.3 on page 46 describes this process when the application uses dynamic arguments to move information into the database. Section 4.5 on page 50 describes this process when the application uses row descriptors to extract information from the database.)

Fields of the respective descriptors specify data type and type attributes. Section 4.8.1 describes the use of the TYPE field in implementation descriptors. Section 4.8.2 on page 58 describes the use of the TYPE field in application descriptors.

When the implementation and application descriptors specify different data types or type attributes, transferring data results in data conversion. Section 4.8.3 on page 60 specifies the permitted combinations and Section 4.8.4 on page 61 specifies their effects.

4.8.1 Specifying SQL Data Types

Implementation descriptors have fields that describe data from the standpoint of the database. The integer values that are valid for the TYPE field denote one of the SQL data types. (They use the same values defined in the X/Open SQL specification and the ISO SQL standard.) For some data types, other descriptor fields specify attributes such as length, precision and scale. For date/time values, CLI uses an integer date/time subcode based on the ISO SQL standard. To specify these integer values, applications should use the symbols shown in the centre and right columns in the following table. The application can also use implementation-defined values.
### Table 4-4 Integers Specifying SQL Data Types

<table>
<thead>
<tr>
<th>SQL Data Type</th>
<th>Integer Data Type Identifier</th>
<th>Date/Time Type Subcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER(n)</td>
<td>SQL_CHAR</td>
<td>SQL_CODE_DATE</td>
</tr>
<tr>
<td>DATE</td>
<td>SQL_DATETIME</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>SQL_DECIMAL</td>
<td></td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>SQL_DOUBLE</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>SQL_FLOAT</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>SQL_INTEGER</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>SQL_NUMERIC</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>SQL_REAL</td>
<td></td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SQL_SMALLINT</td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>SQL_DATETIME</td>
<td>SQL_CODE_TIME</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>SQL_DATETIME</td>
<td>SQL_CODE_TIMESTAMP</td>
</tr>
<tr>
<td>CHARACTER VARYING(n)</td>
<td>SQL_VARCHAR</td>
<td></td>
</tr>
</tbody>
</table>

### Table 4-5 Integers Specifying Date/Time Subtypes

<table>
<thead>
<tr>
<th>Special Integer Data Type Identifier</th>
<th>Resulting Integer Date Type Identifier</th>
<th>Resulting Date/Time Type Subcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_TYPE_DATE</td>
<td>SQL_DATETIME</td>
<td>SQL_CODE_DATE</td>
</tr>
<tr>
<td>SQL_TYPE_TIME</td>
<td>SQL_DATETIME</td>
<td>SQL_CODE_TIME</td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td>SQL_DATETIME</td>
<td>SQL_CODE_TIMESTAMP</td>
</tr>
</tbody>
</table>

### 4.8.2 Specifying Application Buffer Types

Application descriptors have fields that describe data from the standpoint of the application. The tables below give the values that specify particular application data types, and recapitulate (from Section 2.2.2 on page 16 and Section 2.2.3 on page 17) the specific data types the application uses.

The application can also use implementation-defined values.

#### C Language

In C programs, the application descriptor’s TYPE field denotes one of the generic C data types defined in Section 2.2.2 on page 16. Applications should set the descriptor’s TYPE field to one of the values from the centre column of the following table:
Interface Overview

<table>
<thead>
<tr>
<th>Symbolic Data Type</th>
<th>Integer Data Type Identifier</th>
<th>C Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCHAR</td>
<td>SQL_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>SQLDATE</td>
<td>SQL_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>SQLDECIMAL</td>
<td>SQL_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>SQLDOUBLE</td>
<td>SQL_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>SQLFLOAT</td>
<td>SQL_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>SQLINTEGER</td>
<td>SQL_INTEGER</td>
<td>long</td>
</tr>
<tr>
<td>SQLNUMERIC</td>
<td>SQL_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>SQLREAL</td>
<td>SQL_REAL</td>
<td>float</td>
</tr>
<tr>
<td>SQLSMALLINT</td>
<td>SQL_SMALLINT</td>
<td>short</td>
</tr>
<tr>
<td>SQLTIME</td>
<td>SQL_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>SQLTIMESTAMP</td>
<td>SQL_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>SQLVARCHAR</td>
<td>SQL_CHAR</td>
<td>unsigned char</td>
</tr>
</tbody>
</table>

Table 4-6  Integers Specifying Application Buffers in C

COBOL

In COBOL programs, the application descriptor’s TYPE field denotes one of the generic COBOL data types defined in Section 2.2.3 on page 17. Applications should set the descriptor’s TYPE field to one of the values from the centre column of the following table:

<table>
<thead>
<tr>
<th>Symbolic Data Type</th>
<th>Integer Data Type Identifier</th>
<th>COBOL Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCHAR</td>
<td>SQL_CHAR</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>SQLDATE</td>
<td>SQL_CHAR</td>
<td>PIC X(10)</td>
</tr>
<tr>
<td>SQLDECIMAL</td>
<td>SQL_DECIMAL</td>
<td>PIC S9(p-s)V9(s) PACKED-DECIMAL</td>
</tr>
<tr>
<td>SQLDOUBLE</td>
<td>SQL_CHAR</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>SQLFLOAT</td>
<td>SQL_CHAR</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>SQLINTEGER</td>
<td>SQL_INTEGER</td>
<td>PIC S9(9)</td>
</tr>
<tr>
<td>SQLNUMERIC</td>
<td>SQL_NUMERIC</td>
<td>PIC S9(p-s)V9(s) DISPLAY SIGN</td>
</tr>
<tr>
<td>SQLREAL</td>
<td>SQL_CHAR</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>SQLSMALLINT</td>
<td>SQL_SMALLINT</td>
<td>PIC S9(4)</td>
</tr>
<tr>
<td>SQLTIME</td>
<td>SQL_CHAR</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>SQLTIMESTAMP</td>
<td>SQL_CHAR</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>SQLVARCHAR</td>
<td>SQL_CHAR</td>
<td>PIC X(n)</td>
</tr>
</tbody>
</table>

Table 4-7  Integers Specifying Application Buffers in COBOL
4.8.3 Permitted Combinations

For a given dynamic argument or database column value, the implementation and application data types must be compatible, as defined below. (Compatibility is only relevant when there are separate implementation and application descriptors and they specify different data types.)

Implementations’ use of data types may vary. No correspondence is guaranteed to be exact; any data value may change precision and scale when moving between the application and implementation. Limits on precision and scale, as well as truncation and rounding rules for type conversions, follow the rules of the X/Open SQL specification. Truncation of values to the right of the decimal point for numeric values returns a truncation warning, whereas truncation to the left of the decimal point returns an error.

Direct Transfers

The following combinations are valid and do not result in data conversion:

<table>
<thead>
<tr>
<th>Integer Describing SQL Data Type</th>
<th>Integer Describing Application Buffer Type (C)</th>
<th>COBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR</td>
<td>SQL_CHAR</td>
<td>SQL-CHAR</td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>no direct transfer in C</td>
<td>SQL-DECIMAL</td>
</tr>
<tr>
<td>SQL_INTEGER</td>
<td>SQL_DOUBLE</td>
<td>no direct transfer in COBOL</td>
</tr>
<tr>
<td>SQL_NUMERIC</td>
<td>no direct transfer in C</td>
<td>SQL-NUMERIC</td>
</tr>
<tr>
<td>SQL_REAL</td>
<td>SQL_REAL</td>
<td>no direct transfer in COBOL</td>
</tr>
<tr>
<td>SQL_SMALLINT</td>
<td>SQL_SMALLINT</td>
<td>SQL-SMALLINT</td>
</tr>
<tr>
<td>SQL_VARCHAR</td>
<td>SQL_CHAR</td>
<td>SQL-CHAR</td>
</tr>
</tbody>
</table>

Table 4-8 Buffer Combinations not Resulting in Data Conversion

No direct transfer for SQL_FLOAT is shown in the table above because neither C nor COBOL has a floating-point data type whose precision can be stated specifically, as in SQL. Except as noted below, the C programmer must assume that the FLOAT value of interest can reasonably fit a C `double` variable; but any transfer in either direction between an FLOAT column and a C `double` variable may result in loss of precision, depending on the precision of the FLOAT column.

DATE, TIME and TIMESTAMP have no standard host-language support in either C or COBOL, and values for such columns must therefore be both supplied and retrieved as character strings (see Transfers with Conversion to/from String on page 62).

Default Transfers

The application can specify SQL_DEFAULT as a data type. This tells the implementation to assume a data type for the application buffer based on the data type in the implementation descriptor.

- If the application provides SQL_DEFAULT in an application parameter descriptor or in `BindParam()`, then the dynamic argument is interpreted based on the data type of the corresponding dynamic parameter, as specified in the following table, and data conversion may occur.

- If the application provides SQL_DEFAULT in an application row descriptor or in `BindCol()`, then columns fetched from the database are placed in the application buffer in a data type that depends on their data type in the database according to the following table.
SQL_DEFAULT is the initial value when a record of an application row descriptor is created.

<table>
<thead>
<tr>
<th>Database Value</th>
<th>Implied Data Type of Application Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR</td>
<td>SQL_CHAR (no conversion)</td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>SQL_CHAR</td>
</tr>
<tr>
<td>SQL_DOUBLE</td>
<td>SQL_DOUBLE (no conversion)</td>
</tr>
<tr>
<td>SQL_INTEGER</td>
<td>SQL_INTEGER (no conversion)</td>
</tr>
<tr>
<td>SQL_NUMERIC</td>
<td>SQL_CHAR</td>
</tr>
<tr>
<td>SQL_REAL</td>
<td>SQL_REAL (no conversion)</td>
</tr>
<tr>
<td>SQL_SMALLINT</td>
<td>SQL_SMALLINT (no conversion)</td>
</tr>
<tr>
<td>SQL_VARCHAR</td>
<td>SQL_CHAR (no conversion)</td>
</tr>
</tbody>
</table>

Table 4-9 Implied Application Data Types under SQL_DEFAULT

Cautions on Use of SQL_DEFAULT

At the time of the transfer, the implementation replaces SQL_DEFAULT in the TYPE field of the descriptor with an integer that describes the result of the transfer. Thus, to specify SQL_DEFAULT semantics again for the next transfer, the application must reset the TYPE field of the descriptor.

Specifying this type of default semantics from COBOL when the database type is REAL or DOUBLE inhibits data conversion, as shown in the table. Since REAL and DOUBLE are not COBOL data types, transferring such data without conversion typically results in misinterpretation by the receiver.

To retrieve column data successfully, the application must determine correctly the length and starting point of the data in the application buffer. When the application specifies an explicit buffer type, application misconceptions are readily detected. However, when the application specifies SQL_DEFAULT, the transfer can be applied to a column of a different data type from the one intended by the application, either from changes to the metadata or by applying the code to a different column. In this case, the application may fail to determine the start or length of the fetched column data. This can lead to unreported data errors or memory violations.

Other Transfers

It is implementation-defined what other combinations of SQL data types and application buffer types are allowed. For example, an implementation might support a transfer from the INTEGER SQL data type into an application buffer type of SQL_DOUBLE. In these cases, the criteria under which truncation warnings, truncation errors and overflow errors occur are implementation-defined.

4.8.4 Transfers and Conversions

String-to-String Transfers

When assigning a character-string value to a character-string table column or application buffer, the length of the source may differ from the length of the destination:

- If the source is shorter than the destination, and the destination is of type CHARACTER, remaining trailing characters of the destination are filled with blanks. (If the destination is of type CHARACTER VARYING, and the source is shorter than the maximum length of the destination, assignment occurs, determined by the length of the source, and there is no
• If the source is longer than the destination, trailing characters of the source are truncated. This may produce diagnostics; see Section 5.4.10 on page 76.

These same rules apply when a non-character value is converted to string because the application specifies assignment to a character-string target.

Transfers with Conversion to/from String

The application can transfer a value of any SQL data type by specifying SQL_CHAR as the application buffer type. For numeric or date/time SQL data types, this implies a conversion. That is, when retrieving a numeric or date/time value, the implementation moves the value's character-string representation to the application's buffer. When supplying a numeric or date/time value, the application's buffer must contain a valid character-string representation of that value. Transfers in both directions follow the rules for literals of the respective type:

• The rules for numeric literals are specified in the X/Open SQL specification. Conversion between numeric and string formats depends on two attributes, precision and scale, discussed below.

EX

• The rules for date/time literals are as follows:

| — Valid dates (SQLDATE) have a length of 10 characters (SQL_DATE_LEN), follow the sequence year, month, day, and have the following format: yyyy-mm-dd. |
| — Valid times (SQLTIME) have a length of 8 characters (SQL_TIME_LEN) or greater (see below), follow the sequence hour, minute, second, and have the following format: hh:mm:ss. |
| — Valid timestamps (SQLTIMESTAMP) have a format that concatenates the format of SQLDATE, one space as a separator, and the format of SQLTIME. Thus a timestamp's length is 19 characters (SQL_TIMESTAMP_LEN) or greater (see below). |

EX

For SQLTIME and SQLTIMESTAMP, the precision attribute refers to the fractional part of the seconds component. If the precision is 0, the seconds component does not have a fractional part and the length and format of the data are as specified above. If the precision is greater than 0, then it specifies a number of decimal digits for the fractional part of the seconds component. The data type's length in characters is that specified above, plus 1 character for a decimal point, plus the precision. An example timestamp with a precision of 2 characters is: 1993-12-31 23:59:59.99

EX

C programs must specify SQL_CHAR, forcing string conversion, for SQL data types DATE, TIME, TIMESTAMP, DECIMAL and NUMERIC, because there is no direct equivalent for DECIMAL and NUMERIC in C. COBOL programs must likewise specify SQL-CHAR for SQL data types DATE, TIME, TIMESTAMP, DOUBLE, FLOAT and REAL. The application must also compute a buffer size in octets sufficient to hold strings of the length in characters given above.

For conversion of numeric data, the application uses additional descriptor fields to specify precision and scale. (For conversion of times and timestamps, the application specifies precision only. This refers to the seconds component, as described above.) For example, to provide a dynamic argument defined as the SQL type DECIMAL(5,3), a C application might specify:

```c
SetDescField(ImpParamDesc, RecNumber, SQL_DESC_TYPE, SQL_DECIMAL, 0);
SetDescField(ImpParamDesc, RecNumber, SQL_DESC_PRECISION, 5, 0);
SetDescField(ImpParamDesc, RecNumber, SQL_DESC_SCALE, 3, 0);
```
Concise alternatives using SetDescRec() and BindCol() are discussed in Section 6.3 on page 81.

4.9 Transaction Management

A transaction is a recoverable unit of work, which may include SQL database operations, that is to be completed atomically (to have an all-or-nothing effect). In embedded SQL, a transaction begins implicitly and ends when the application executes either the COMMIT or ROLLBACK statement. In CLI, a transaction also begins implicitly when an application that does not already have an active transaction operates on a database.\(^\text{13}\) The connection over which the application executes the statement is now active.

The transaction completes when the application calls EndTran(). (The transaction may complete in other ways discussed below.) The application must not specify the COMMIT and ROLLBACK statements in Prepare() or ExecDirect(). When the transaction is completed, the relevant connections are no longer active.

Effects within a Connection

Completing a transaction has the following effects on CLI data structures:

- It is implementation-defined whether prepared SQL statements survive transactions. A portable application should prepare statements again in order to execute them as part of a new transaction.
- Cursor names, bound dynamic parameters and column bindings survive transactions. However, completing a transaction closes any open cursors.

Effects across Connections

Within an environment, it is implementation-defined whether multiple connections can be active. On implementations that allow this, it is implementation-defined which of the following is true:

- A transaction is local to a single connection; completing a transaction on an environment commits or rolls back work done in each connection as a unit, but independent of the success of commitment on any other connection.
- A transaction spans all affected connections in the environment (those on which the application has operated on a database). Completing a transaction on an environment either enacts all work done in the environment since the start of the transaction, or undoes it all.

\(^\text{13}\) The application operates on a database in any of the following ways:

- by preparing an SQL statement that operates on a database, using the CLI functions Prepare() or ExecDirect() (the X/Open SQL specification defines which SQL statements operate on a database)
- by executing a metadata function (see Section 4.6 on page 53)
- it is implementation-defined whether SQL data definition statements (operations on metadata) obey the same transaction semantics — the variety of current implementations is discussed in the X/Open SQL specification.

For some implementations, a transaction may have begun even if one of the above calls fails. After such a failure, portable applications should call EndTran() to ensure that a transaction is not active.

CLI calls concerning connection, allocation and specification of dynamic arguments do not operate on a database, nor do the Cancel(), GetDiagField(), GetDiagRec() and EndTran() calls.
To maximise portability, an application should complete a transaction before switching to another connection.

**Effects across Environments**

In implementations that allow the application to use more than one environment, the application can complete transactions on each environment separately; for example, by calling `EndTran()`. In the case where the application connects to the same database through two or more environments, it is implementation-defined which of the following is true:

- Work performed in several environments is within a single transaction context and is committed atomically.
- Work performed in each environment constitutes a separate transaction and the implementation provides no implicit coordination of commitment between the transactions.

A portable application should allocate only one environment.

**Errors**

Any attempt to start or to complete a transaction may fail for implementation-defined reasons. A transaction may fail for implementation-defined reasons any time during a connection, and it is possible that the application can detect the failure only after its next operation on the database. In addition, a transaction can be rolled back even if the call to `EndTran()` specified commitment. These topics are discussed in Section 5.4.4 on page 74.

### 4.9.1 Explicit Transaction Demarcation

CLI can be used in conjunction with transaction demarcation APIs, provided that the application ends each transaction using the same API that it used to begin that transaction. If an application calls a function from such an API explicitly to start a transaction, it must do so before operating on a database. If an application begins a transaction implicitly (by operating on a database as described above), it must call `EndTran()` to complete the transaction. It is implementation-defined which, if any, explicit transaction demarcation APIs are supported.

For example, the X/Open TX specification specifies a non-SQL-specific API with functions that start, commit and roll back a global transaction. These global transaction control functions allow SQL work to be coordinated with non-SQL work.

### 4.9.2 Distributed Transaction Processing

When the application starts a transaction, it is either global or local according to implementation-defined rules.

A local transaction pertains only to the connection through which the application operated on the database.

In a global transaction, the work can be coordinated with other work, so that the work is either all committed or all rolled back; this means the success of an application's work may depend on other work that the application does not control.

CLI does not define a way for the application to specify local versus global, nor to refer to a particular transaction. An application using CLI refers to global transaction work (with the `EndTran()` call) only by specifying the environment. If the application uses an explicit transaction demarcation API, that API may let the application participate further in the definition of transactions.
For example, the X/Open TX specification defines a global transaction identifier and specifies that the basis on which work is considered part of a global transaction is the fact that the work is performed by the same application thread of control. This and other X/Open Distributed Transaction Processing (DTP) documents describe methods of coordinating databases so that changes to any recoverable resources are committed or rolled back atomically.
This chapter explains how CLI reports diagnostic information to the application.

Each CLI function returns a basic diagnostic as the function return value, as described in Section 5.1. The GetDiagField() function provides more detailed diagnostic information, as described in Section 5.2 on page 69. This arrangement lets applications handle the basic flow of control simply, but lets an application, at its option, determine specific causes of failure.

Other information that is present in the diagnostics area for historical reasons is presented in Section 5.3 on page 72. General diagnostics in Section 5.4 on page 73 cover warning and failure situations, common to many CLI functions, which are not repeated on the individual reference manual pages.

Diagnostics whose value or significance is specific to a CLI function appear on the respective manual page in Chapter 8.

5.1 Return Value

Syntactic Form of Return Value

The definition of all CLI functions includes a return value of the SMALLINT generic type. In the reference manual pages in Chapter 8, the generic definition of all functions includes:

```c
RETURNS (SMALLINT)
```

For example, the CLI function `Execute()` is called in C as follows:

```c
ReturnCode = SQLExecute(StatementHandle);
```

In programming languages such as COBOL where called subroutines do not return a value, the status code is an additional explicit output parameter of the CLI function. For example, when called in COBOL, `Execute()` is instead defined as follows:

```cobol
CALL "SQLRExecute" USING STATEMENTHANDLE, RETURNCODE.
```

All references in this specification to the result of a CLI function should be interpreted in COBOL environments as referring to the output value of this last parameter.

Defined Return Values

The header file (see Appendix C) defines symbolic names for result values of a CLI function that a portable application should use:

- `SQL_SUCCESS`
  
  The function completed successfully. (This includes the case where null column values are fetched.)

- `SQL_SUCCESS_WITH_INFO`
  
  The function completed successfully; warning or additional information is available by calling `GetDiagField()`. For example, truncation of column values (except in `GetDiagField()`) produces this warning outcome.
[SQL_NO_DATA]

**DataSources()**
There are no more server names to retrieve.

**Error()**
No further diagnostic records exist for the specified handle.

**Execute(), ExecDirect() and ParamData()**
The executed SQL statement produced the **No data** outcome. These cases are enumerated in the X/Open SQL specification. Additional implementation-defined diagnostic information may be available, which the application can obtain by calling **GetDiagField()**.

**Fetch()**
There are no rows in the result set, or previous calls have fetched all rows from the result set.

**FetchScroll()**
The specified row of the result set does not exist.

**GetData()**
The preceding fetch operation returned [SQL_NO_DATA].

**GetDescField(), GetDescRec(), GetDiagField(), GetDiagRec()**
The caller asked for information from a record number that exceeds the number of records available.

[SQL_ERROR]
The function failed; call **GetDiagField()** for more specific failure information.

[SQL_INVALID_HANDLE]
The function failed due to an invalid handle passed as an input argument. This indicates a programming error. No further information is available from **GetDiagField()**.

[SQL_NEED_DATA]
The application tried to execute a SQL statement but the implementation lacks parameter data that the application had indicated would be passed at execute time (see Section 4.3.2 on page 48).

In error cases where **AllocHandle()** cannot allocate or gain access to the relevant handle, it uses other semantics to report errors; see **AllocHandle()** on page 92.
5.2 Detailed Diagnostic Information

Most CLI calls produce diagnostic information describing the outcome of the call. The diagnostic information includes a header with general information pertaining to the CLI function's execution, and zero or more status records\(^\text{14}\) each describing one condition or event related to the call's execution. Each status record contains the following:

- A five-character, standardised SQLSTATE diagnostic code. Values of SQLSTATE are shown in this manual in the format ‘CCSSS’, where CC is the class code and SSS is the subclass. The format, values, and usage are the same as in the X/Open SQL specification, except that additional, CLI-specific codes are defined as class ‘HY’. (See Appendix A for a list of CLI SQLSTATE values with cross-references.)

  Implementation-defined conditions or events are reported with class codes or subclass codes in the implementation-defined range listed in the X/Open SQL specification.

- An implementation-defined numeric code.

- An implementation-defined message.

- Strings that indicate the document that defines the class and the subclass portion of the SQLSTATE value in this record.

- Strings that specify the connection name and the server name that the diagnostic record pertains to.

Status records may also contain other implementation-defined data.

All types of handle accommodate a diagnostics data structure containing such records. Any CLI function that writes diagnostic information to a handle's data structure (even a success indication) writes the entire data structure without regard for its previous contents.

If Cancel() was called to cancel the execution of a CLI function that is running in another process or thread, and it succeeded in cancelling the execution, a status record with ‘HY008’ appears in the diagnostic data structure of the associated statement handle to indicate this.

**Number of Status Records**

If a function returns [SQL_SUCCESS], there are no status records; the corresponding SQLSTATE (‘00000’) from embedded SQL is not used in CLI.

If a function returns [SQL_INVALID_HANDLE], there is no valid handle with which to associate diagnostic information.

If a function returns [SQL_ERROR] or [SQL_SUCCESS_WITH_INFO], then there is at least one status record.

If a function returns [SQL_NO_DATA], there is no status record corresponding to the SQLSTATE (‘02000’) from embedded SQL, but there may be one or more implementation-defined status records in class ‘02’.

\(^{14}\) The corresponding term in the X/Open SQL specification is exception information item.
Sequence of Status Records

If there are two or more status records, they occur in an undefined order, except that if one of the records contains the value that X/Open embedded SQL would place in the SQLSTATE variable, then that record appears first in the diagnostic data structure.

Consistent with the ISO SQL standard, this initial record can be defined for CLI as the highest-ranking condition using the following ranking:

1. **Errors.** If two or more status records describe the same condition, then the standardised or X/Open-defined SQLSTATE (classes from ‘03’ to and including ‘HZ’) outranks the implementation-defined SQLSTATE.\(^{15}\)

2. **Implementation-defined No Data values** in class ‘02’.

3. **Warnings** — values in class ‘01’. If two or more status records describe the same condition, then the standardised or X/Open-defined SQLSTATE outranks the implementation-defined SQLSTATE.

If the implementation generates two or more status records that all have the highest rank of any status record generated, then it is undefined which of these is the first record.

In the remainder of the diagnostics data structure, an application cannot assume that errors precede warnings. Applications should scan the entire diagnostic data structure to obtain complete information on the non-success outcome of a CLI function.

Obtaining Diagnostic Information

The application obtains diagnostic information by specifying the handle and the number of the desired status record.

A call to `GetDiagField()` obtains a single field of the specified diagnostic record. `GetDiagField()` is extensible and can return diagnostic information defined in the future or by implementations.

`GetDiagField()` returns `SQL_NO_DATA` if the application specifies a record number in excess of the available diagnostic records. Therefore, the application can scan the entire set of status records by calling `GetDiagField()` repeatedly, incrementing the desired record number until the function returns `SQL_NO_DATA`.

5.2.1 Error Codes from the X/Open SQL Specification

The X/Open SQL specification (SQLSTATE Status Variable) reserves all class and subclass codes starting with 0-4 or A-H for definition by an international standard. RDA-specific errors are in class ‘HZ’.

The X/Open SQL specification (SQLSTATE Values) is the authoritative reference for a description of each SQLSTATE code. Using CLI to execute SQL text whose syntax or usage violates the X/Open SQL specification produces the error code specified by the X/Open SQL specification.

\(^{15}\) In addition, records that indicate a failure or possible failure of the transaction outrank records that indicate statement failure.
5.2.2 CLI-specific Errors

Error class 'HY' covers all CLI-specific errors. Implementation-defined errors pertaining to the application's use of CLI specify class 'HY' and subclasses '500' to '9ZZ' inclusive and 'I00' to 'ZZZ' inclusive. (X/Open reserves subclass codes 'S00' to 'SZZ' inclusive.)
5.3 Other Information in the Diagnostics Data Structure

Certain non-diagnostic information is present in the diagnostic data structure for the historical reason that it appears in the diagnostic area in embedded SQL.

5.3.1 Type of SQL Statement

EX The implementation stores, in the diagnostic data structure, the identity of the SQL statement executed by a call to `ExecDirect()` or `Execute()`. The application can obtain this information, as a string or as an integer, by calling `GetDiagField()` with `DiagIdentifier` set to `SQL_DIAG_DYNAMIC_FUNCTION` or `SQL_DIAG_DYNAMIC_FUNCTION_CODE` respectively.

5.3.2 Row Count

After executing a statement by calling `Execute()` or `ExecDirect()`, the diagnostics data structure associated with the statement handle contains a row count that specifies the number of rows that the statement affected. To obtain the row count, an application calls `GetDiagField()` with `DiagIdentifier` set to `SQL_DIAG_ROW_COUNT`.

If the statement was searched DELETE, INSERT or searched UPDATE, the row count is the same statistic that is available in embedded SQL by executing:

```
GET DIAGNOSTICS variable = ROW_COUNT
```

- For the searched DELETE statement, this is the number of rows deleted from the table.
- For the INSERT statement, this is the number of rows inserted into the table.
- For the searched UPDATE statement, this is the number of rows updated in the table.

If the statement was not one of the above types, the resulting row count is undefined.

The count does not include any rows in other tables that the statement may have affected.

OP X/Open encourages implementations to define the row count for additional SQL statements as follows:

a. For any positioned UPDATE and positioned DELETE statement, the row count is 1.
b. For any other SQL statement, the row count is 0.

A portable application cannot assume these semantics, and should not use the row count unless the SQL statement executed was INSERT, searched UPDATE or searched DELETE.
5.4 General Diagnostics

Each CLI function description includes a DIAGNOSTICS section that describes error and warning outcomes. The following information describes errors that pertain to many CLI functions. This information is not repeated in function descriptions.

5.4.1 Connection Errors

Disconnection can occur independently of the application because of communication errors, administrative action or other events. Disconnect(), plus an implementation-defined set of CLI functions,\(^{16}\) can report the following:

- There is no current connection (‘08003’).
- There is no current connection because of an event that occurred independently of the application (‘08006’).\(^{17}\)

Any disconnection of the current connection rolls back any active transaction. The diagnostics associated with the connection handle report both the rollback and the loss of the connection.

5.4.2 Change of Connection

When the application passes the handle of a dormant connection to a CLI function, it implicitly makes that connection current and all other connections dormant. When the application passes a statement handle to a CLI function, the same applies if the connection on which the statement handle was allocated is dormant. Connection and statement context (that is, information associated with the SQLHDBC and SQLHSTMT defined in Section 3.1 on page 26) persist across a period of dormancy.

It is implementation-defined whether a single transaction may encompass more than one connection. If an implementation disallows this, then specifying an SQLHDBC that implies a change of connection, when a transaction is active, fails (‘0A001’). The CLI function also fails if the new connection has been disconnected, as described in the previous section.

5.4.3 State Transition Errors

The application must call CLI functions in a strict order (‘HY010’). Although many cases of invalid calling sequences are listed on the respective reference manual pages, the state tables in Appendix B authoritatively define this type of error.

Valid sequences for CLI function calls when the data-at-execute dialogue is in progress are specified in Section 4.3.2 on page 48 and Table B-3 on page 244.

---

\(^{16}\) Generally, the set of functions that take a connection handle as an argument.

\(^{17}\) The implementation may instead return the SQLSTATE code ‘40003’ (Statement completion unknown) or ‘08007’ (Transaction state unknown). See the X/Open SQL specification (general diagnostics for connection statements) for details.

If Disconnect() reports this condition, it reports a warning (‘01002’) instead of an error because the disconnection succeeds.
**Timing of Attempts to Set Attributes**

Certain attributes can be set only at certain times (`HY011`). This diagnostic covers the following cases:

- The application cannot set any environment attribute while a connection handle is allocated on that environment.
- Certain statement attributes cannot be set when there is a prepared statement or an open cursor on the statement handle (see the **DIAGNOSTICS** for `SetStmtAttr()`).
- Implementation-defined attributes may carry implementation-defined restrictions on the application’s ability to set the attribute’s value. Any such restrictions reflect cases in which processing on the handle has progressed to the point that changing the specified attribute’s value would be meaningless or erroneous.

**5.4.4 Transaction Errors**

**Start of Transaction**

When a CLI function call starts a transaction on a database (see Section 4.9 on page 63), and there is already work in progress over a different connection, the function may fail (`0A001`) for the following reasons:

- There are insufficient resources to add another connection to the current transaction.
- The implementation does not support global transactions.

**Spontaneous Rollback**

Any CLI function, including `EndTran()`, that contacts a server to access the database (that is, any function that could acquire a lock in the database) can report that a rollback has been executed against the current transaction. (The SQLSTATE class is ‘40’ and the subclass is one of those specified in Appendix A.) A program cannot assume that any changes are made to the database until a call to `EndTran()` that specifies commitment returns successfully. The transaction may fail for various reasons, including communication errors, action taken by the server to prevent deadlock, and server failure. In all of these cases, it is possible that the transaction is rolled back.

**5.4.5 Inability to Access Handle**

In some small-system environments, processing of handles can fail under extreme low memory conditions. Any CLI function may thus report that it cannot gain access to the handle (`HY013`).* Any CLI function may also report a memory allocation failure (`HY001`).*

* In a failure where the CLI function cannot gain access to the handle, there is no way to return the specified SQLSTATE value. There are additional error semantics in the case of `AllocHandle()`; see **RETURN VALUE** in `AllocHandle()` on page 92.
5.4.6 Cancellation

A CLI function may fail (‘HY008’) because it tried to execute an SQL statement that was cancelled by use of the Cancel() function during its operation. See Cancel() on page 105 for details.

5.4.7 String Lengths

Except as noted in the reference manual pages, an argument denoting the length of an input string must be non-negative or SQL_NTS[L]. Moreover, a length of 0 is not valid (‘HY090’) except in the following cases:

- to specify a zero-length dynamic argument
- as the server name, user identifier and authentication string in Connect(), where a zero length indicates an implementation-defined default value
- as a qualification parameter of a metadata function (see Section 4.6.1 on page 54), where a zero-length string inhibits qualification of the query on that basis.

If null termination of output strings is in effect in the environment, the input argument describing the length in octets of an output string must be at least the number of octets of the null terminator (see Section 2.3.6 on page 19) (‘HY090’).

5.4.8 Null Pointers

In a programming language that allows the application to use pointers, the application must not supply a null pointer (‘HY009’) except in the following cases:

- to specify a zero-length dynamic argument
- as the server name, user identifier and authentication string in Connect(), where a null pointer indicates an implementation-defined default value
- as a qualification parameter of a metadata function (see Section 4.6.1 on page 54), where a null pointer inhibits qualification of the query on that basis
- any other case explicitly permitted in the reference manual pages in Chapter 8.

5.4.9 Descriptor Record Consistency

Application Descriptors

Whenever the application sets the DATA_PTR field of a descriptor, the implementation checks that the value of the TYPE field and the values of fields applicable to that TYPE are valid and consistent (‘HY021’):18

- The TYPE field must be one of the values listed in Section 4.8.2 on page 58 or an implementation-defined value.

* In a failure where the CLI function cannot gain access to the handle, there is no way to return the specified SQLSTATE value. There are additional error semantics in the case of AllocHandle(); see RETURN VALUE in AllocHandle() on page 92.
18. The comparable SQLSTATE values in embedded SQL are ‘07001’ for parameters and ‘07002’ for rows. However, in CLI, the eventual use of the descriptor record is not known at the time this check is made.
• All other fields that apply to the data type that the descriptor specifies must be valid and consistent.

Implementation Descriptors

The application does not normally set the DATA_PTR field of an implementation descriptor. The application can do so, however, to force the following checking to occur ('HY021'):

• The TYPE field must be one of the values listed in Table 4-4 on page 58 or an implementation-defined value.

• If the TYPE field is SQL_DATETIME, then the DATETIME_INTERVAL_CODE field must be one of the integer Date/Time Type Subcode values listed in Table 4-4 on page 58 or an implementation-defined value.

• All other fields that apply to the data type that the descriptor specifies must be valid (for example, the precision may not be negative) and consistent.

5.4.10 Data Conversion and Assignment Diagnostics

Any CLI function that assigns values to application row buffers or to database columns can produce the diagnostics described below.

String Truncation

When assigning a character string (or the string representation of non-character data), and the target length does not completely hold the string, trailing characters are truncated.

• When assigning to a table column, the string value must be such that no non-blank character is truncated ('22001'). Truncation of blank characters does not produce a diagnostic.

• When assigning to an application buffer, the Success with warning outcome occurs if any characters (even blanks) are truncated ('01004').

Character Representation of Non-Character Data

When assigning a non-character value to a character-string target, there must exist a representation of a literal with the value of the source in the character set defined by the implementation ('22018').

Numeric Assignments

When assigning a non-null numeric value to an exact numeric target, there must be a representation of the numeric value in the target data type that does not cause the whole part of the number (that is, the leading significant digits) to be truncated ('22003'). The fractional part of the number (that is, the trailing significant digits) may be truncated and it is implementation-defined whether warning flags or indicator variables are set.

When assigning a non-null numeric value to an approximate numeric table column or application buffer, the numeric value must be within the range of magnitude of the destination field ('22003'). The result is an approximation of the source numeric value that has the precision of the destination field.
Date/Time Assignments

When the target data type is a date/time type, the source must contain only valid characters ('22007'). (Currently, X/Open defines date/time assignments only for transfers from application parameter buffers to the database.)

Unspecified Transfers

For any combination of source and target data types not specified in this document, it is implementation-defined which of the following occurs:

- An implementation-defined conversion is performed.
- The implementation generates the '07006' error to indicate that the combination is invalid.
- The implementation generates the 'HYC00' error to indicate that the combination is valid but unsupported.
This chapter describes a group of concise CLI functions.

CLI functions that get and set fields have been defined with extensibility in mind. These functions deal with a single field per call. In each function, one argument specifies the desired field. This technique allows future editions of this X/Open specification to define new fields while maintaining backward compatibility with existing applications. It also allows implementations to define proprietary fields whose values can be obtained using the standard interface.

The concise functions are designed to simplify application coding and decrease the required number of CLI functions in many typical cases. They provide a standard higher-level interface to other CLI functions. In many cases, use of the concise functions can keep the application from having to allocate, and obtain the value of, a descriptor.

Most of the concise functions return a set of commonly-used fields of a CLI record, such as a descriptor record. These functions are not designed to be extensible either to yet-to-be-defined fields or to implementation-defined fields. Moreover, the concise functions provide access to an arbitrary selection of fields. An application that needs access to other fields may use the extensible functions, or may supplement use of the concise functions with one or more calls to the extensible functions.
6.1 Inventory of Concise Functions

The concise functions provided, and the underlying functions in whose terms the concise functions are defined, are as follows:

<table>
<thead>
<tr>
<th>Concise Function</th>
<th>Underlying Function(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BindCol()</td>
<td>SetDescField()</td>
</tr>
<tr>
<td>BindParam()</td>
<td>SetDescField()</td>
</tr>
<tr>
<td>DescribeCol()</td>
<td>GetDescField()</td>
</tr>
<tr>
<td>GetDescRec()</td>
<td>GetDescField()</td>
</tr>
<tr>
<td>GetDiagRec()</td>
<td>GetDiagField()</td>
</tr>
<tr>
<td>NumResultCols()</td>
<td>GetDescField()</td>
</tr>
<tr>
<td>SetDescRec()</td>
<td>SetDescField()</td>
</tr>
</tbody>
</table>

Table 6-1 Basis of Concise CLI Functions

6.2 Definition of Concise Functions

Although complete reference manual pages are provided in Chapter 8, the essential conceptual definition of a concise CLI function is in terms of the underlying function(s) shown above. Implementations are not required to implement a concise function in terms of these underlying functions, but are required to produce results as though that were the case.

This conceptual definition minimises ambiguity about side-effects of functions. It explicitly permits applications to mix calls to concise and extensible functions as they see fit.
6.3 Overview of Concise Functions

**BindCol()**

Section 3.5 on page 32 explains descriptors and Section 4.5 on page 50 explains binding columns and fetching data from the database.

*BindCol()* allows the application to bind columns of a result set to application variables without having to use descriptors. A single call to *BindCol()* sets most of the common fields of the application row descriptor associated with the specified statement handle.

*BindCol()* facilitates and standardises application coding in most common cases. It does not give the application access to future or implementation-defined descriptor fields.

**BindParam()**

Section 3.5 on page 32 explains descriptors and Section 4.3 on page 46 explains how to use them to bind dynamic parameters to application variables.

*BindParam()* allows the application to bind dynamic parameters to application variables without having to use descriptors. *BindParam()* specifies the type and attributes of a dynamic parameter and associates the deferred fields of the application parameter descriptor with specified application variables. *BindParam()* sets most of the common fields of the descriptor record in a single call.

*BindParam()* facilitates and standardises application coding in most common cases. It does not give the application access to future or implementation-defined descriptor fields.

**DescribeCol()**

Section 3.5 on page 32 explains descriptors and Section 4.5 on page 50 explains fetching data from the database.

*DescribeCol()* lets the application obtain the type and attributes of a column of a result set without having to use descriptors. *DescribeCol()* retrieves several fields of the implementation row descriptor in a single call.

*DescribeCol()* facilitates and standardises application coding in most common cases. It does not accommodate the case where the application needs to obtain additional fields from the implementation row descriptor. It does not give the application access to future or implementation-defined descriptor fields. It does not return column data from the result set.

**GetDescRec()**

Section 3.5 on page 32 explains descriptors. Chapter 4 gives several cases where descriptors are used but the description and examples assume the application obtains a single descriptor field per CLI call.

*GetDescRec()* obtains all the commonly used fields of a descriptor in a single call. This includes the name (used only when reading the implementation row descriptor), data type and subtype, length, precision, scale and nullable attributes.

*GetDescRec()* facilitates and standardises application coding in most common cases. It does not give the application access to future or implementation-defined descriptor fields.
GetDiagRec()

Section 5.2 on page 69 explains CLI diagnostic information. The description assumes the application obtains a single piece of diagnostic information at a time by calling GetDiagField().

GetDiagRec() obtains all the commonly used fields of a diagnostic record in a single call. GetDiagRec() gives the application access to the SQLSTATE code, the implementation-defined native error, and the implementation-defined message text.

GetDiagRec() facilitates and standardises application coding in most common cases. It does not give the application access to future or implementation-defined diagnostic information fields. It does not give the application access to some currently-defined fields of the diagnostic data structure that are less frequently used, such as the row count or the identity of the SQL statement executed.

NumResultCols()

Section 3.5 on page 32 explains descriptors. Chapter 4 gives several cases where descriptors are used but the description and examples assume the application obtains a single descriptor field per CLI call.

NumResultCols() lets the application obtain the number of columns of a result set without having to use descriptors.

SetDescRec()

SetDescRec() is the counterpart to GetDescRec(). It sets commonly used fields of a descriptor, including the deferred fields for data, string length and indicator. See also the discussion of GetDescRec() above.
This chapter, which describes a group of deprecated CLI functions, should be regarded as completely shaded and subject to the DE margin legend.

The functions described in this chapter do not provide additional capabilities, nor do they reflect the concise/extensible distinction as do the functions described in Chapter 6. However, the functions have entered widespread use based on proprietary CLI implementations that predated the publication of X/Open CLI. X/Open includes those functions in this document, and thereby requires that all X/Open-compliant CLI implementations provide those functions, for the convenience of the CLI applications that have already been written.

However, by marking these functions as deprecated, X/Open recommends that applications move toward use of the CLI functions described in earlier chapters. The reference manual pages for the deprecated functions advise application programmers on the equivalent non-deprecated call that should be used instead.

X/Open intends to remove the deprecated functions from a future issue of this document.
7.1 Inventory of Deprecated Functions

The deprecated functions provided, and the functions to be used in preference to them, are as follows:

<table>
<thead>
<tr>
<th>Deprecated Function</th>
<th>Preferred Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocConnect()</td>
<td>AllocHandle(SQL_HANDLE_DBC)</td>
</tr>
<tr>
<td>AllocEnv()</td>
<td>AllocHandle(SQL_HANDLE_ENV)</td>
</tr>
<tr>
<td>AllocStmt()</td>
<td>AllocHandle(SQL_HANDLE_STMT)</td>
</tr>
<tr>
<td>ColAttribute()</td>
<td>GetDescField()</td>
</tr>
<tr>
<td>Error()</td>
<td>GetDiagRec()</td>
</tr>
<tr>
<td>FreeConnect()</td>
<td>FreeHandle(SQL_HANDLE_DBC)</td>
</tr>
<tr>
<td>FreeEnv()</td>
<td>FreeHandle(SQL_HANDLE_ENV)</td>
</tr>
<tr>
<td>FreeStmt()</td>
<td>FreeHandle(SQL_HANDLE_STMT)</td>
</tr>
<tr>
<td>GetConnectOption()</td>
<td>GetConnectAttr()</td>
</tr>
<tr>
<td>GetStmtOption()</td>
<td>GetStmtAttr()</td>
</tr>
<tr>
<td>RowCount()</td>
<td>GetDiagField(SQL_DIAG_ROW_COUNT)</td>
</tr>
<tr>
<td>SetConnectOption()</td>
<td>SetConnectAttr()</td>
</tr>
<tr>
<td>SetParam()</td>
<td>BindParam()</td>
</tr>
<tr>
<td>SetStmtOption()</td>
<td>SetStmtAttr()</td>
</tr>
<tr>
<td>Transact()</td>
<td>EndTran(SQL_HANDLE_STMT)</td>
</tr>
</tbody>
</table>

Table 7-1 Table of Deprecated Functions

7.2 Definition of Deprecated Functions

Although complete reference manual pages are provided in Chapter 8, the essential conceptual definition of a deprecated CLI function is in terms of the preferred function(s) shown above. Implementations are not required to implement a deprecated function in terms of its preferred function, but are required to produce results as though that were the case.

This conceptual definition minimises ambiguity about side-effects of functions. It explicitly permits applications to mix calls to deprecated and non-deprecated functions as they see fit.

X/Open encourages implementations to implement deprecated functions as calls to, or macros of, the corresponding preferred functions.
### 7.3 Overview of Deprecated Functions

#### Explicit Allocate and Free Functions

The preferred `AllocHandle()` and `FreeHandle()` functions take an argument of the SMALLINT data type that indicates the handle type to be allocated or freed. These functions are thus extensible to future or implementation-defined handle types.

For three of the handle types, the following deprecated CLI functions to allocate and free that specific handle type are in widespread use:

- `AllocConnect()`
- `AllocEnv()`
- `AllocStmt()`
- `FreeConnect()`
- `FreeEnv()`
- `FreeStmt()` (see below)

Five of these cases correspond directly to a call to `AllocHandle()` or `FreeHandle()` with appropriate arguments. Applications should replace calls to the functions listed above with calls to `AllocHandle()` or `FreeHandle()`.

The `FreeStmt()` function is an exception. The `Option` argument lets the application specify four different behaviours, only one of which corresponds directly to a call to `FreeHandle()`. Two of the behaviours return descriptor records to the unbound state. The only way to achieve these effects with non-deprecated functions is to free and reallocate the descriptor.

#### Handle Option Functions

The following handle option functions are comparable to the CLI attribute functions, such as `GetConnectAttr()`, but align with CLI implementations in which attributes are known as `options`:

- `GetConnectOption()`
- `GetStmtOption()`
- `SetConnectOption()`
- `SetStmtOption()`

These handle option functions do not have parameters that indicate the length of the attribute value. When retrieving character-string attributes using these functions, the application must either know the exact length in octets of the attribute or allocate the maximum size for the output buffer and initialise it before retrieving the attribute.

Functions `GetEnvOption()` and `SetEnvOption()` are not provided because CLI implementations that support options do not define any environment options.

#### Descriptor Function

The `ColAttribute()` function is included in this document because it is in widespread use. The function is equivalent to `GetDescField()`.
Overview of Deprecated Functions

Deprecated CLI Functions

Diagnostic Functions

The following deprecated CLI functions that obtain diagnostic information are included because they are in widespread use:

- Error()
-RowCount()

A sequence of calls to Error() corresponds to a sequence of calls to GetDiagRec(). Whereas GetDiagRec() can obtain information from the diagnostics area in random order, each call to Error() returns one record and removes that record from the set of records subsequently available through Error().

In X/Open-compliant systems, applications are allowed to intermix calls to Error() and GetDiagRec(). Thus, a compliant implementation of Error() must not modify the diagnostic area. One model of a compliant implementation is to use a counter as RecNumber, which is reset on any statement that writes the diagnostics area and is incremented on each call to Error().

The RowCount() function corresponds to a call to GetDiagField() that specifies the SQL_DIAG_ROW_COUNT diagnostic field. This is the row count (see Section 5.3.2 on page 72) resulting from the most recent call to ExecDirect() or Execute().

Transaction Management Function

The deprecated Transact() function is included because it is in widespread use. It is comparable to the preferred EndTran() function. The Transact() function uses a connection handle and takes an environment handle and connection handle as arguments. The application can specify null handle values to indicate the scope of transaction completion. By contrast, the preferred EndTran() function uses an argument of the SQLSMALLINT data type to indicate the handle type and thus the scope of work to be completed.

Synonym Function

The deprecated function SetParam() is similar to the preferred function BindParam().

Implementations must support both functions, but may do so by implementing one as a macro of the other.
This chapter contains reference manual pages for the functions in the CLI interface in alphabetical order.

The synopsis for the generic function is provided at the top of each manual page. At the end of the manual page there are prototype function calls, including allocation of variables, for:

- the by-value variant as called from C
- the by-reference variant as called from COBOL.

The **DESCRIPTION** section of each function includes a list of all function parameters. The legend (input) means the parameter passes information from the application to the implementation. The legend (output) means the parameter returns information from the implementation to the application. General semantic information for CLI appears in Section 2.3 on page 18.

The **RETURN VALUE** section lists all return values the function may return. Section 5.1 on page 67 describes the meaning of each return value and explains the syntax of the return value parameter in languages where called subroutines do not return a value.

A **DIAGNOSTICS** section lists any specific errors and warnings that a call to the function can produce. The application can obtain the five-character SQLSTATE values listed (in the format 'CCSSS') by calling `GetDiagField()` or `GetDiagRec()`. All **DIAGNOSTICS** sections should be read in conjunction with the [General Diagnostics](#) discussed in Section 5.4 on page 73.

Where applicable, the manual pages include comments concerning application usage, and include usage examples in C and COBOL. In several cases of variable-length output parameters, the appropriate COBOL PICTURE depends on the context; this is illustrated with a placeholder in italics, as in the following example:

```
 01 VALUE PICTURE type.
```

Each manual page also includes a list of any closely-related functions.
NAME

AllocConnect — Allocate a connection handle (deprecated)

SYNOPSIS

```c
FUNCTION AllocConnect
    (EnvironmentHandle INTEGER,
     ConnectionHandle INTEGER)
RETURNS (SMALLINT)
```

DESCRIPTION

The `AllocConnect()` function allocates a connection within the environment specified by `EnvironmentHandle` and returns a handle to the connection in `ConnectionHandle`. The function has the following parameters:

- `EnvironmentHandle` (input)
  The environment handle with which `ConnectionHandle` is to be associated.

- `ConnectionHandle` (output)
  The handle for the newly-allocated connection.

The connection handle provides for management of information related to the connection, including general status information, transaction state and error information. The application can pass `ConnectionHandle` to `Connect()` to connect the application to a server, and then pass it to `AllocHandle()` or `AllocStmt()` to allocate statement handles associated with that connection.

Using a handle that was already allocated as `ConnectionHandle` is an application programming error. Implementations are not required to detect this error, and typically overwrite the handle value without regard to its previous contents. Specifically, implementations are not required to dispose of resources that the old handle referenced.

The `AllocConnect()` function sets all attributes for the allocated handle to their initial values. Initial values of connection attributes are specified in Connection Attributes on page 29.

RETURN VALUE


  If `AllocConnect()` returns `[SQL_ERROR]`, `ConnectionHandle` contains SQL_NULL_HDBC and additional diagnostic information is associated with `EnvironmentHandle`.

DIAGNOSTICS

The application must not already have allocated the implementation-defined maximum number of connections (`HY014`).

APPLICATION USAGE

If `AllocConnect()` returns `[SQL_ERROR]`, applications should not use `ConnectionHandle` further. Passing the null handle to a subsequent CLI function makes that function fail with `[SQL_INVALID_HANDLE]`.

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHENV EnvironmentHandle;
SQLHDBC ConnectionHandle;
...
ReturnCode = SQLAllocConnect(EnvironmentHandle, &ConnectionHandle);
```

EXAMPLE USAGE (COBOL)

```cobol
01 ENVIRONMENTHANDLE PICTURE S9(9) BINARY.
01 CONNECTIONHANDLE PICTURE S9(9) BINARY.
```
01 RETURNCODE  PICTURE S9(4) BINARY.

...  

CALL "SQLRAlocConnect" USING ENVIRONMENTHANDLE, CONNECTIONHANDLE, 
RETURNCODE.

SEE ALSO

AllocHandle(), FreeConnect(), FreeHandle()
NAME
AllocEnv — Allocate an environment handle (deprecated)

SYNOPSIS

FUNCTION AllocEnv

(EnvironmentHandle INTEGER)

RETURNS (SMALLINT)

DESCRIPTION
The AllocEnv() function creates an environment and returns a handle to it in EnvironmentHandle. The function has the following parameter:

EnvironmentHandle (output)
The handle for the newly-allocated environment.

The environment handle provides for management of global information such as valid connections and the current connection. The application can pass EnvironmentHandle to a subsequent call to AllocHandle() or AllocConnect() to allocate a connection.

Using a handle that was already allocated as EnvironmentHandle is an application programming error. Implementations are not required to detect this error, and typically overwrite the handle value without regard to its previous contents. Specifically, implementations are not required to dispose of resources that the old handle referenced.

The AllocEnv() function sets all attributes for the allocated handle to their initial values. Initial values of environment attributes are specified in Section 3.2 on page 28.

RETURN VALUE

[SQL_SUCCESS] or [SQL_ERROR].

If AllocEnv() cannot allocate memory for an environment, it returns [SQL_ERROR], EnvironmentHandle contains SQL_NULL_HENV, and there is no handle with which to associate additional diagnostic information. If AllocEnv() fails to allocate an environment handle for any other reason, it returns [SQL_ERROR] and EnvironmentHandle contains a restricted handle to a skeleton environment. The application can use this handle only on CLI calls that obtain diagnostic information or that free the handle.

DIAGNOSTICS
The application must not already have allocated the implementation-defined maximum number of environments (’HY014’). In this case, EnvironmentHandle is a restricted handle (see RETURN VALUE).

APPLICATION USAGE
If AllocEnv() returns [SQL_ERROR], applications should not use EnvironmentHandle further, except as described in RETURN VALUE above. Passing the null handle to a subsequent CLI function or passing the restricted handle except as described above, makes that function fail with [SQL_INVALID_HANDLE].

Many implementations support only one environment. Portable applications should only allocate one environment handle at a time.

EXAMPLE USAGE (C)

#include <sqlcli.h>

SQLRETURN ReturnCode;
SQLHENV EnvironmentHandle;
...
ReturnCode = SQLAllocEnv(&EnvironmentHandle);
EXAMPLE USAGE (COBOL)

01 ENVIRONMENTHANDLE PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.

...

CALL "SQLRAlocEnv" USING ENVIRONMENTHANDLE, RETURNCODE.

SEE ALSO

AllocHandle(), FreeEnv(), FreeHandle()
NAME
AllocHandle — Allocate a handle

SYNOPSIS
FUNCTION AllocHandle
(HandleType SMALLINT,
  InputHandle INTEGER,
  OutputHandle INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The AllocHandle() function has the following parameters:

HandleType (input)
A handle type identifier that describes the handle type of OutputHandle. It must contain one of the following values:

- SQL_HANDLE_ENV
  AllocHandle() creates an environment and returns a handle to it in OutputHandle. InputHandle should be SQL_NULL_HANDLE since an environment handle does not exist in the context of any other handle.

- SQL_HANDLE_DBC
  AllocHandle() allocates a connection within the environment specified by the environment handle InputHandle and returns a handle to the connection in OutputHandle.

- SQL_HANDLE_STMT
  AllocHandle() allocates a statement within the connection specified by the connection handle InputHandle and returns a handle to the statement in OutputHandle.

- SQL_HANDLE_DESC
  AllocHandle() allocates a descriptor, associates it with the connection handle InputHandle and returns a handle to the descriptor in OutputHandle.

InputHandle (input)
The handle that describes the data structure in whose context the new data structure is to be allocated; except that, if HandleType is SQL_HANDLE_ENV, this is SQL_NULL_HANDLE.

OutputHandle (output)
The handle for the newly allocated data structure.

The application can pass OutputHandle to subsequent CLI functions to indicate a data structure on which they should operate. In the case of environment handles and connection handles, the application can pass the value of OutputHandle to a subsequent call to AllocHandle() as the InputHandle argument, to allocate additional, subsidiary data structures.

Using a handle that was already allocated as OutputHandle is an application programming error. Implementations are not required to detect this error, and typically overwrite the handle value without regard to its previous contents. Specifically, implementations are not required to dispose of resources that the old handle referenced.
Handle Attributes
If attributes are defined for the handle type of OutputHandle, calling AllocHandle() sets all attributes to their initial values specified in Chapter 3.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR] or [SQL_INVALID_HANDLE].
When allocating a handle other than an environment handle, if AllocHandle() returns [SQL_ERROR], OutputHandle contains SQL_NULL_HDBC, SQL_NULL_STMT or SQL_NULL_DESC (depending on HandleType), and additional diagnostic information is associated with InputHandle. This would occur, for example, if AllocHandle fails before it allocates OutputHandle (which is typical for some memory allocation errors and produces 'HY001').
When allocating an environment handle, if the application provides a null pointer for OutputHandle or if AllocHandle() cannot allocate memory for OutputHandle, AllocHandle() returns [SQL_ERROR], OutputHandle contains SQL_NULL_HENV (unless the application provided a null pointer), and there is no handle with which to associate additional diagnostic information. If AllocHandle() fails to allocate an environment handle for any other reason, it returns [SQL_ERROR] and OutputHandle contains a restricted handle to a skeleton environment. The application can use this handle only on CLI calls that obtain diagnostic information or that free the handle.

DIAGNOSTICS
Depending on the timing and nature of a diagnostic event, the diagnostic may be returned on InputHandle or on OutputHandle; see RETURN VALUE above.
HandleType must be one of the values defined above ('HY092', returned on InputHandle).
The application must not already have allocated the implementation-defined maximum number of data structures corresponding to type HandleType ('HY014'). If this occurs and HandleType is SQL_HANDLE_ENV, then OutputHandle is a restricted handle (see RETURN VALUE).
When allocating a statement or descriptor handle on a connection, the connection must be open ('08003').

APPLICATION USAGE
If AllocHandle() returns [SQL_ERROR], applications should not use OutputHandle further, except as described in RETURN VALUE above. Passing the null handle to a subsequent CLI function, or passing the restricted handle except as described above, makes that function fail with [SQL_INVALID_HANDLE].
Many implementations support only one environment. Portable applications should only allocate one environment handle at a time.

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCodeEnv; SQLRETURN ReturnCodeDbc; SQLRETURN ReturnCodeStmt; SQLRETURN ReturnCodeDesc; SQLHENV EnvironmentHandle; SQLHDBC ConnectionHandle; SQLH_STMT StatementHandle; SQLH_DESC DescriptorHandle; ...
ReturnCodeEnv = SQLAllocHandle(SQL_HANDLE_ENV,
AllocHandle() SQL_NULL_HANDLE, &EnvironmentHandle);
...
ReturnCodeDbc = SQLAllocHandle(SQL_HANDLE_DBC,
  EnvironmentHandle, &ConnectionHandle);
...
ReturnCodeStmt = SQLAllocHandle(SQL_HANDLE_STMT,
  ConnectionHandle, &StatementHandle);
...
ReturnCodeDesc = SQLAllocHandle(SQL_HANDLE_DESC,
  ConnectionHandle, &DescriptorHandle);

EXAMPLE USAGE (COBOL)
  01 ENVIRONMENTHANDLE PICTURE S9(9) BINARY.
  01 CONNECTIONHANDLE PICTURE S9(9) BINARY.
  01 STATEMENTHANDLE PICTURE S9(9) BINARY.
  01 DESCRIPTORHANDLE PICTURE S9(9) BINARY.
  01 RETURNCODEENV PICTURE S9(4) BINARY.
  01 RETURNCODEDBC PICTURE S9(4) BINARY.
  01 RETURNCODESTM T PICTURE S9(4) BINARY.
  01 RETURNCODEDESC PICTURE S9(4) BINARY.
  ...
  CALL "SQLRAlocHandle" USING SQL-HANDLE-ENV,
       SQL-NULL-HANDLE, ENVIRONMENTHANDLE, RETURNCODEENV.
  ...
  CALL "SQLRAlocHandle" USING SQL-HANDLE-DBC,
       ENVIRONMENTHANDLE, CONNECTIONHANDLE, RETURNCODEDBC.
  ...
  CALL "SQLRAlocHandle" USING SQL-HANDLE-STM T,
       CONNECTIONHANDLE, STATEMENTHANDLE, RETURNCODESTM T.
  ...
  CALL "SQLRAlocHandle" USING SQL-HANDLE-DESC,
       CONNECTIONHANDLE, DESCRIPTORHANDLE, RETURNCODEDESC.

SEE ALSO

  FreeHandle()
NAME

AllocStmt — Allocate a statement handle (deprecated)

SYNOPSIS

FUNCTION AllocStmt
  (ConnectionHandle INTEGER,
   StatementHandle INTEGER)
RETURNS (SMALLINT)

DESCRIPTION

The AllocStmt() function allocates a statement within the connection specified by
ConnectionHandle and returns a handle to the statement in StatementHandle. The function has the
following parameters:

ConnectionHandle (input)

  The connection handle with which StatementHandle is to be associated.

StatementHandle (output)

  The handle for the newly-allocated statement.

The statement handle provides for management of information related to the processing of SQL
statements, including descriptors, result values, and status information. The application can
pass StatementHandle to subsequent CLI functions to indicate a data structure on which they
should operate.

Using a handle that was already allocated as StatementHandle is an application programming
error. Implementations are not required to detect this error, and typically overwrite the handle
value without regard to its previous contents. Specifically, implementations are not required to
dispose of resources that the old handle referenced.

The AllocStmt() function sets all attributes for the allocated handle to their initial values. Initial
values of statement attributes are specified in Statement Attributes on page 30.

RETURN VALUE

[SQL_SUCCESS], [SQL_ERROR] or [SQL_INVALID_HANDLE].

If AllocStmt() returns [SQL_ERROR], StatementHandle contains SQL_NULL_STMT and
additional diagnostic information is associated with ConnectionHandle.

DIAGNOSTICS

The application must not already have allocated the implementation-defined maximum number
of statements (’HY014’).

The connection specified by ConnectionHandle must be open (’08003’).

APPLICATION USAGE

If AllocStmt returns [SQL_ERROR], applications should not use StatementHandle further. Passing
the null handle to a subsequent CLI function makes that function fail with [SQL_INVALID_HANDLE].

EXAMPLE USAGE (C)

#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDBC ConnectionHandle;
SQLHSTMT StatementHandle;
...

ReturnCode = SQLAllocStmt(ConnectionHandle, &StatementHandle);
EXAMPLE USAGE (COBOL)

01 CONNECTIONHANDLE PICTURE S9(9) BINARY.
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...

CALL "SQLRAlocStmt" USING CONNECTIONHANDLE, STATEMENTHANDLE, RETURNCODE.

SEE ALSO

AllocHandle(), FreeHandle(), FreeStmt()
NAME

BindCol — Bind a column in a result set (concise)

SYNOPSIS

FUNCTION BindCol
(StateHandle INTEGER,
ColumnNumber SMALLINT,
TargetType SMALLINT,
TargetValue ANY,
BufferLength INTEGER,
StrLen_or_Ind INTEGER)
RETURNS (SMALLINT)

DESCRIPTION

The BindCol() function binds a column of a result set to an application variable, enabling implicit data transfer and optional data conversion when a row of the result set is fetched. The function has the following parameters:

StatementHandle (input)
Statement handle.

ColumnNumber (input)
Column number. Columns are numbered sequentially from left to right and the leftmost is number 1.

TargetType (input)
The application data type of TargetValue. Specifying SQL_DEFAULT provides that a buffer type from Table 4-9 on page 61 will be selected at fetch time based on the SQL data type of the column being bound. Specifying SQL_CHAR, if the database value is a date/time or numeric value, results in string data conversion as described in Section 4.8.3 on page 60.

TargetValue (deferred output)
A variable that the implementation loads with column data when the fetch occurs.

BufferLength (input)
If TargetType specifies a character-string data type (or is SQL_DEFAULT and this implies a character-string data type), then this is the maximum number of octets to store in TargetValue. Otherwise, it is ignored.

StrLen_or_Ind (deferred output)
A variable the implementation sets, when the fetch occurs, to one of the following:

• the value SQL_NULL_DATA if the data value for the column is null
• the length in octets of the data value for the column (even if it does not entirely fit in TargetValue), excluding any null terminator, if TargetType is SQL_CHAR (or TargetType is SQL_DEFAULT and the database column is of type CHAR or VARCHAR)
• an undefined value (but not SQL_NULL_DATA) in all other cases.

The implementation assigns values to TargetValue and StrLen_or_Ind whenever a fetch occurs on StatementHandle. The implementation performs any data conversion specified in the call to BindCol().

RETURN VALUE

[SQL_SUCCESS], [SQL_ERROR], [SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.
DIAGNOSTICS

ColumnNumber must be 1 or greater (’07009’).

TargetType must be one of the Integer Data Type Identifier values listed in Section 4.8.2 on page 58, SQL_DEFAULT or an implementation-defined value (’HY003’).

Any descriptor implicitly referenced (see the conceptual description under APPLICATION USAGE) must be valid (’HY023’).

The subsequent Fetch() call may produce the data conversion diagnostics specified in Section 5.4.10 on page 76.

APPLICATION USAGE

The application can call BindCol() to set in a single call all the descriptor fields commonly used when retrieving a value from the database. Conceptually, BindCol() performs the following steps in sequence:

- Calls GetStmtAttr() to obtain the application row descriptor handle.
- Calls GetDescField() to get this descriptor’s COUNT field, and if ColumnNumber exceeds the value of COUNT, calls SetDescField() to increase the value of COUNT to ColumnNumber.
- Calls SetDescField() multiple times to assign values to the following fields of the application row descriptor:
  - sets TYPE to the value of TargetType
  - sets OCTET_LENGTH to the value of BufferLength
  - sets DATA_PTR to the value of TargetValue
  - sets INDICATOR_PTR to the value of StrLen_or_Ind (see below)
  - sets OCTET_LENGTH_PTR to the value of StrLen_or_Ind (see below).

The variable that StrLen_or_Ind references is used for both indicator and length information. If a fetch encounters a null value for the column, it stores SQL_NULL_DATA in this variable; otherwise it stores the data length in this variable. Passing a null pointer as StrLen_or_Ind keeps the fetch operation from returning the data length, but makes the fetch fail if it encounters a null value and has no way to return SQL_NULL_DATA.

The application must call BindCol() before starting a fetch operation for any columns to be bound on the first fetch. The application may call BindCol() thereafter to change the binding of those columns for subsequent fetches, or to bind additional columns for subsequent fetches. In languages that allow the use of pointers, the application may unbind a column by providing a null pointer for TargetValue to set the DATA_PTR field of the record in the row descriptor to a null pointer. To unbind all columns, the application calls FreeStmt() with the SQL_UNBIND option, or sets the COUNT field of the application row descriptor to zero.

The application must allocate enough storage for TargetValue to accommodate the longest data value the implementation might store there, taking into account any specified data conversion, or truncation will occur at the time of the fetch, causing the fetch to return [SQL_SUCCESS_WITH_INFO]. The application can detect that a fetch truncated the value in a bound column by comparing the value the implementation stored in that column’s StrLen_or_Ind to the buffer length the application declared in the call to BindCol(), minus the length of the null terminator if applicable (see Section 2.3.6 on page 19).

Cautions Regarding SQL_DEFAULT

To retrieve column data successfully, the application must determine correctly the length and starting point of the data in the application buffer. When the application specifies an explicit
**TargetType**, application misconceptions are readily detected. However, when the application specifies a **TargetType** of SQL_DEFAULT, **BindCol()** can be applied to a column of a different data type from the one intended by the application, either from changes to the metadata or by applying the code to a different column. In this case, the application may fail to determine the start or length of the fetched column data. This can lead to unreported data errors or memory violations.

**EXAMPLE USAGE (C)**

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLSMALLINT ColumnNumber;
SQLSMALLINT TargetType;
SQLPOINTER TargetValue;
SQLINTEGER BufferLength;
SQLINTEGER StrLen_or_Ind;
...
ReturnCode = SQLBindCol(StatementHandle, ColumnNumber, 
    TargetType, TargetValue, BufferLength, &StrLen_or_Ind);
```

**EXAMPLE USAGE (COBOL)**

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 COLUMNNUMBER PICTURE S9(4) BINARY.
01 TARGETTYPE PICTURE S9(4) BINARY.
01 TARGETVALUE PICTUREtype .
01 BUFFERLENGTH PICTURE S9(9) BINARY.
01 STRLEN-OR-IND PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRBindCol" USING STATEMENTHANDLE, COLUMNNUMBER, 
    TARGETTYPE, TARGETVALUE, BUFFERLENGTH, STRLEN-OR-IND, RETURNCODE.
```

**SEE ALSO**

*Fetch(), FetchScroll(), SetDescField(), SetDescRec()*
NAME

BindParam, SetParam — Bind a dynamic parameter (concise)

SYNOPSIS

FUNCTION BindParam
(StatementHandle INTEGER,
 ParameterNumber SMALLINT,
 ValueType SMALLINT,
 ParameterType SMALLINT,
 LengthPrecision INTEGER,
 ParameterScale SMALLINT,
 ParameterValue ANY,
 StrLen_or_Ind INTEGER)
RETURNS (SMALLINT)

FUNCTION SetParam
(StatementHandle INTEGER,
 ParameterNumber SMALLINT,
 ValueType SMALLINT,
 ParameterType SMALLINT,
 LengthPrecision INTEGER,
 ParameterScale SMALLINT,
 ParameterValue ANY,
 StrLen_or_Ind INTEGER)
RETURNS (SMALLINT)

DESCRIPTION

The `BindParam()` function binds a parameter in an SQL statement to an application variable, enabling implicit data transfer and optional data conversion when the statement is executed.

The `SetParam()` function is similar to `BindParam()`. (Differences appear with shading within this entry.)

The function has the following parameters:

* **StatementHandle (input)**
  Statement handle.

* **ParameterNumber (input)**
  Parameter number. Parameters are numbered sequentially from left to right and the leftmost is number 1.

* **ValueType (input)**
  The application data type of `ParameterValue`. Specifying SQL_DEFAULT provides that a buffer type from Table 4-9 on page 61 will be selected at execute time based on the SQL data type specified in `ParameterType`. Specifying SQL_CHAR, if the parameter is a date/time or numeric value, results in string data conversion as described in Section 4.8.3 on page 60.

* **ParameterType (input)**
  The SQL data type of the parameter. Integer Data Type identifier values are listed in Table 4-4 on page 58. For the date/time data types, the application can specify SQL_TYPE_DATE, SQL_TYPE_TIME, or SQL_TYPE_TIMESTAMP.

* **LengthPrecision (input)**
  Maximum size of the parameter. Its interpretation depends on the following values of `ParameterType`:
SQL_CHAR or SQL_VARCHAR
The maximum length in characters of the parameter.

SQL_DECIMAL or SQL_NUMERIC
The maximum decimal precision.

SQL_FLOAT
The maximum binary precision.

SQL_TYPE_TIME
LengthPrecision is ignored and the precision of the TIME data is assumed to be 0.

SQL_TYPE_TIMESTAMP
— For BindParam(), the value of LengthPrecision can be from 0 up to and including 6, and is interpreted as the length in characters of the fractional part of the seconds component of the timestamp.

— For SetParam(), the value of LengthPrecision can be from 16 up to and including 26, and is interpreted as the total length in characters of the character representation of the timestamp.

(Other values)
LengthPrecision is ignored.

ParameterScale (input)
If ParameterType is SQL_DECIMAL or SQL_NUMERIC, this is the scale of the parameter (the total number of digits to the right of the decimal point). For SetParam(), if ParameterType is SQL_TYPE_TIMESTAMP, this is the length in characters of the fractional part of the seconds component of the timestamp. Otherwise, ParameterScale is ignored.

ParameterValue (deferred input)
A variable whose value is used as the parameter data when StatementHandle is executed.

If the dynamic argument is to be supplied at execute time by calling ParamData() and PutData(), the application should set ParameterValue to an arbitrary value that uniquely identifies the dynamic parameter. This value will be returned to the application by ParamData().

StrLen_or_Ind (deferred input)
A variable whose value is interpreted when StatementHandle is executed.

— If a null value is to be used as the parameter, StrLen_or_Ind must contain the value SQL_NULL_DATA.

— If the dynamic argument is to be supplied at execute time by calling ParamData() and PutData(), StrLen_or_Ind must contain the value SQL_DATA_AT_EXEC.

— If ValueType is SQL_CHAR and the data in ParameterValue contains a null-terminated string, StrLen_or_Ind must either contain the length in octets of the string in ParameterValue or contain the value SQL_NTS.

— If ValueType is SQL_CHAR and the data in ParameterValue is not null-terminated, StrLen_or_Ind must contain the length in octets of the string in ParameterValue.

— Otherwise, StrLen_or_Ind is ignored.

Deferred Effects
The implementation evaluates ParameterValue and StrLen_or_Ind when StatementHandle is executed. At that time, StrLen_or_Ind must indicate the length in octets of the value in
**ParameterValue** or have one of the special values listed above. If **StrLen_or_Ind** is neither SQL_NULL_DATA nor SQL_DATA_AT_EXEC, then **ParameterValue** must at this time contain the actual data to use in place of parameter **ParameterNumber**.

The implementation performs any data conversion implied by the combination of **ValueType** and **ParameterType** (without modifying **ParameterValue**) and sends the resulting value to the server in place of the parameter marker.

**RETURN VALUE**

[SQL_SUCCESS], [SQL_ERROR], [SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

**DIAGNOSTICS**

**ParameterNumber** must be 1 or greater ('07009').

**ValueType** must be one of the Integer Data Type Identifier values listed in Section 4.8.2 on page 58, SQL_DEFAULT or an implementation-defined value ('HY003').

**ParameterType** must be one of the values discussed above in the DESCRIPTION, or an implementation-defined value ('HY004').

The specified **ValueType** and **ParameterType** must be compatible. The permitted combinations are defined in Section 4.8.3 on page 60 ('07006').

If **ParameterType** is SQL_TIME_TIMESTAMP, **LengthPrecision** must be as specified above in DESCRIPTION ('HY104').

Any descriptor implicitly referenced (see the conceptual description under **APPLICATION USAGE**) must be valid ('HY023').

The descriptor's consistency is checked; see Section 5.4.9 on page 75.

Execution of the SQL statement may produce the data conversion diagnostics specified in Section 5.4.10 on page 76.

**APPLICATION USAGE**

The application can call **BindParam()** to set in a single call all the descriptor fields commonly used when defining a parameter. Conceptually, **BindParam()** performs the following steps in sequence:

- calls **GetStmtAttr()** to obtain the application parameter descriptor handle
- calls **GetDescField()** to get this descriptor's COUNT field, and if **ColumnNumber** exceeds the value of COUNT, calls **SetDescField()** to increase the value of COUNT to **ColumnNumber**
- calls **SetDescField()** multiple times to assign values to the following fields of the application parameter descriptor:
  - sets TYPE to the value of **ValueType**
  - sets DATA_PTR to the value of **ParameterValue**
  - sets OCTET_LENGTH_PTR to the value of **StrLen_or_Ind**
  - sets INDICATOR_PTR also to the value of **StrLen_or_Ind**.

The **StrLen_or_Ind** parameter specifies both the indicator information and the length for the parameter value.

- calls **GetStmtAttr()** to obtain the implementation parameter descriptor handle
calls `GetDescField()` to get this descriptor’s COUNT field, and if `ColumnNumber` exceeds the value of COUNT, calls `SetDescField()` to increase the value of COUNT to `ColumnNumber`.

- calls `SetDescField()` multiple times to assign values to the following fields of the implementation parameter descriptor:
  - sets TYPE to the value of `ParameterType`, except that if `ParameterType` is one of the date/time identifiers SQL_TYPE_DATE, SQL_TYPE_TIME, or SQL_TYPE_TIMESTAMP, it sets TYPE to SQL_DATETIME and sets DATETIME_INTERVAL_CODE to SQL_CODE_DATE, SQL_CODE_TIME, or SQL_CODE_TIMESTAMP, respectively.
  - sets LENGTH or PRECISION or both (as appropriate for `ParameterType`)\(^{19}\)
  - sets SCALE to the value of `ParameterScale`.

To specify a nonzero PRECISION for a parameter of type TIME, the application must explicitly set the descriptor fields by calling `SetDescField()` or `SetDescRec()`.

**EXAMPLE USAGE (C)**
```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLSMALLINT ParameterNumber;
SQLSMALLINT ValueType;
SQLSMALLINT ParameterType;
SQLINTEGER LengthPrecision;
SQLSMALLINT ParameterScale;
SQLPOINTER ParameterValue;
SQLINTEGER StrLen_or_Ind;
...
ReturnCode = SQLBindParam(StatementHandle, ParameterNumber,
    ValueType, ParameterType, LengthPrecision, ParameterScale,
    ParameterValue, &StrLen_or_Ind);
```

**EXAMPLE USAGE (COBOL)**
```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 PARAMETERNUMBER PICTURE S9(4) BINARY.
01 VALUETYPE PICTURE S9(4) BINARY.
01 PARAMETERTYPE PICTURE S9(4) BINARY.
01 LENGTHPRECISION PICTURE S9(9) BINARY.
01 PARAMETERSCALE PICTURE S9(4) BINARY.
01 PARAMETERVALUE PICTURE TYPE.
01 STRLEN-OR-IND PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
```

\(^{19}\) For the following date/time data types, the effect depends on the value of `ParameterType` as follows:

**SQL_TYPE_TIME**
LENGTH is set to 8 and PRECISION is set to 0.

**SQL_TYPE_TIMESTAMP**
- For `BindParam()`, LENGTH is set to 19 (or to 20 + `LengthPrecision`, if `LengthPrecision` > 0) and PRECISION is set to `LengthPrecision`.
- For `SetParam()`, LENGTH is set to `LengthPrecision` and PRECISION is set to `ParameterScale`.  

---
CALL "SQLRBindParam" USING STATEMENTHANDLE, PARAMETERNUMBER, VALUETYPE, PARAMETERTYPE, LENGTHPRECISION, PARAMETERSCALE, PARAMETERVALUE, STRLEN-OR-IND, RETURNCODE.

SEE ALSO
   ExecDirect(), Execute(), ParamData(), Prepare(), PutData(), SetDescField(), SetDescRec()
NAME
Cancel — Attempt to cancel a CLI operation

SYNOPSIS
FUNCTION Cancel
  (StatementHandle INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The application can call Cancel() to cancel CLI operations in the following situations:

1. An application can call Cancel() to force an orderly exit from the data-at-execute dialogue described in Section 4.3.2 on page 48.

   The SQL statement defined on StatementHandle is not executed. If the application had prepared the SQL statement by calling Prepare(), then StatementHandle returns to the prepared state shown in Table B-2 on page 242. Otherwise (if the application had called ExecDirect()), StatementHandle returns to the allocated state.

   The Cancel() function does not affect the contents of any descriptor. The implementation discards any application data previously provided during the data-at-execute dialogue.

2. When an application initiates concurrent processing, one process or thread can call Cancel() to try to cancel execution of an SQL statement by another process or thread that is using the same connection. If the implementation accepts the cancel request and succeeds in cancelling the specified work, the CLI function that initiated the work returns [SQL_ERROR] and the diagnostics area for StatementHandle includes 'HY008'.

   Calling Cancel() does not delete any diagnostic information that execution of the statement has buffered for return to the application.

Otherwise — if StatementHandle is neither in a data-at-execute dialogue nor associated with a currently-executing SQL statement — Cancel() has no effect.

The function has the following parameter:

StatementHandle (input)
   The statement handle representing the work to be cancelled.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

In case 2 above, a return value of [SQL_SUCCESS] indicates only that the implementation has accepted the cancel request; if an SQL statement is executing on StatementHandle, [SQL_SUCCESS] does not guarantee that the execution will be aborted.

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

DIAGNOSTICS
The server may decline the cancel request ('HY018'). For example, communication errors might occur between a client and a server. Additional, implementation-defined diagnostics are also possible.

APPLICATION USAGE
The intent of case 2 above is to provide a way to cancel long-running queries. This document does not specify whether or how an application could get control of the processor during SQL statement execution in order to call Cancel().
Case 2 works as specified only if a process or thread is executing an SQL statement at the time the second process calls `Cancel()`. Calling `Cancel()` at other times could produce the effects described in case 1. This could result in a function sequence error (‘HY010’) in the affected process or thread.

**EXAMPLE USAGE (C)**
```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
...
ReturnCode = SQLCancel(StatementHandle);
```

**EXAMPLE USAGE (COBOL)**
```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRCancel" USING STATEMENTHANDLE, RETURNCODE.
```

**SEE ALSO**
`Execute()`, `ExecDirect()`
NAME
CloseCursor — Close a cursor

SYNOPSIS
FUNCTION CloseCursor
(StatementHandle INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The CloseCursor() function closes the open cursor on a statement handle after execution of a
cursor-specification. The function has the following parameter:
StatementHandle (input)
Statement handle.

Calling CloseCursor() closes any cursor associated with StatementHandle and discards any
pending results. The application can re-execute the statement associated with the cursor, with
the same or different dynamic arguments. If no open cursor is associated with StatementHandle,
the function has no effect.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_INVALID_HANDLE] or
[SQL_SUCCESS_WITH_INFO].
[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are
produced.

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
...  
ReturnCode = SQLCloseCursor(StatementHandle);

EXAMPLE USAGE (COBOL)
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...  
CALL "SQLRCloseCursor" USING STATEMENTHANDLE, RETURNCODE.
NAME

ColAttribute — Get an implementation row descriptor field (deprecated)

SYNOPSIS

FUNCTION ColAttribute

DE

(StatementHandle INTEGER,
  ColumnNumber SMALLINT,
  FieldIdentifier SMALLINT,
  CharacterAttribute ANY,
  BufferLength SMALLINT,
  StringLength SMALLINT,
  NumericAttribute ANY)

RETURNS (SMALLINT)

DESCRIPTION

The ColAttribute() function has the following parameters:

ConnectionHandle (input)
  Statement handle.

ColumnNumber (input)
  The record number from which the specified field in the descriptor is to be retrieved. The
  first record is number 1.

FieldIdentifier (input)
  The identifier of the field to be retrieved. This can be one of the following:

  Header Fields

  SQL_DESC_COUNT (type: SMALLINT)
  Ignore ColumnNumber. Set NumericAttribute to the number of records in the
  implementation row descriptor.

  Fields of a Descriptor Record

  SQL_DESC_TYPE (type: SMALLINT)
  Set NumericAttribute to the TYPE field of ColumnNumber. This is one of the Integer Data
  Type Identifier values in Table 4-4 on page 58, except that, if the TYPE field is
  SQL_DATETIME, it is SQL_TYPE_DATE, SQL_TYPE_TIME, or
  SQL_TYPE_TIMESTAMP, depending on the value of the
  DATETIME_INTERVAL_CODE field (see also Table 4-5 on page 58).

  SQL_DESC_LENGTH (type: INTEGER)
  Set NumericAttribute to the LENGTH field of ColumnNumber.

  SQL_DESC_OCTET_LENGTH (type: INTEGER)
  Set NumericAttribute to the OCTET_LENGTH field of ColumnNumber.

  SQL_DESC_PRECISION (type: SMALLINT)
  Set NumericAttribute to the PRECISION field of ColumnNumber.

  SQL_DESC_SCALE (type: SMALLINT)
  Set NumericAttribute to the SCALE field of ColumnNumber.

  SQL_DESC_NONNULL (type: SMALLINT)
  Set NumericAttribute to the NULLABLE field of ColumnNumber. This is
  SQL_NONNULL if the column’s definition allows null values, and SQL_NO_NULLS
  otherwise.
The ColAttribute() function returns in CharacterAttribute or NumericAttribute one field of ColumnNumber from the implementation row descriptor associated with StatementHandle. If FieldIdentifier denotes a character string, then the implementation does not set NumericAttribute; otherwise the implementation ignores BufferLength and does not set CharacterAttribute nor StringLength.

RETURN VALUE

[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO], [SQL_INVALID_HANDLE] or [SQL_NO_DATA].

[SQL_NO_DATA] is returned if ColumnNumber is greater than the number of descriptor records.

DIAGNOSTICS

The [SQL_SUCCESS_WITH_INFO] outcome occurs if the returned data is truncated ('01004') or if implementation-defined warnings are produced.

FieldIdentifier must be one of the valid values or an implementation-defined value ('HY091').

When FieldIdentifier indicates a field from a record, ColumnNumber must be 1 or greater ('07009').

The prepared statement associated with StatementHandle must be a cursor-specification ('07005').

The application must call ExecDirect() or Prepare() before calling this function ('HY010').

APPLICATION USAGE

The application can obtain fields of any record of the implementation row descriptor, in arbitrary order, by repeated calls to ColAttribute().

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLSMALLINT ColumnNumber;
SQLSMALLINT FieldIdentifier;
SQLPOINTER CharacterAttribute;
```
SQLSMALLINT BufferLength;
SQLSMALLINT StringLength;
SQLPOINTER NumericAttribute;

ReturnCode = SQLColAttribute(StatementHandle, ColumnNumber,
    FieldIdentifier, CharacterAttribute, BufferLength & StringLength,
    NumericAttribute);

EXAMPLE USAGE (COBOL)
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 COLUMNNUMBER PICTURE S9(4) BINARY.
01 FIELDIDENTIFIER PICTURE S9(4) BINARY.
01 CHARACTERATTRIBUTE PICTURE type.
01 BUFFERLENGTH PICTURE S9(4) BINARY.
01 STRINGLENGTH PICTURE S9(4) BINARY.
01 NUMERICATTRIBUTE PICTURE type.
01 RETURNCODE PICTURE S9(4) BINARY.

CALL "SQLRColAttribute" USING STATEMENTHANDLE, COLUMNNUMBER,
    FIELDIDENTIFIER, CHARACTERATTRIBUTE, BUFFERLENGTH, STRINGLENGTH,
    NUMERICATTRIBUTE, RETURNCODE.

SEE ALSO
GetDescField()
NAME
Columns — Get column information

SYNOPSIS
FUNCTION Columns
  (StatementHandle INTEGER,
   CatalogName CHAR,
   NameLength1 SMALLINT,
   SchemaName CHAR,
   NameLength2 SMALLINT,
   TableName CHAR,
   NameLength3 SMALLINT,
   ColumnName CHAR,
   NameLength4 SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION
The Columns() function obtains column information. The function has the following parameters:

StatementHandle (input)
  Statement handle.

CatalogName (input)
  Buffer containing catalog name to qualify the tables of the result set. If the SQL_ATTR_METADATA_ID attribute of StatementHandle is SQL_TRUE, then this is interpreted as an identifier (see Treatment as Identifier on page 54).

NameLength1 (input)
  Length in octets of CatalogName.

SchemaName (input)
  Buffer that may qualify the result set by schema name. If the SQL_ATTR_METADATA_ID attribute of StatementHandle is SQL_TRUE, then this is interpreted as an identifier (see Treatment as Identifier on page 54); otherwise, it is a pattern-value (see Qualification by Pattern-values on page 55).

NameLength2 (input)
  Length in octets of SchemaName.

TableName (input)
  Buffer that may qualify the result set by table name. If the SQL_ATTR_METADATA_ID attribute of StatementHandle is SQL_TRUE, then this is interpreted as an identifier (see Treatment as Identifier on page 54); otherwise, it is a pattern-value (see Qualification by Pattern-values on page 55).

NameLength3 (input)
  Length in octets of TableName.

ColumnName (input)
  Buffer that may qualify the result set by column name. If the SQL_ATTR_METADATA_ID attribute of StatementHandle is SQL_TRUE, then this is interpreted as an identifier (see Treatment as Identifier on page 54); otherwise, it is a pattern-value (see Qualification by Pattern-values on page 55).

NameLength4 (input)
  Length in octets of ColumnName.
Column information is returned in the form of a result set on `StatementHandle`, where each column is represented by one row of the result set. The result set contains many differences, in column names and content, from the COLUMNS system view specified in the X/Open SQL specification.

The result set has at least these columns in the following order:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_CAT</td>
<td>VARCHAR(128)</td>
<td>The name of the catalog containing TABLE_SCHEMA. See APPLICATION USAGE below.</td>
</tr>
<tr>
<td>TABLE_SCHEMA</td>
<td>VARCHAR(128) NOT NULL</td>
<td>The name of the schema containing TABLE_NAME.</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>VARCHAR(128) NOT NULL</td>
<td>The name of the table or view.</td>
</tr>
<tr>
<td>COLUMN_NAME</td>
<td>VARCHAR(128) NOT NULL</td>
<td>The name of the column of the specified table or view.</td>
</tr>
<tr>
<td>DATA_TYPE</td>
<td>SMALLINT NOT NULL</td>
<td>Identifies the type of the column and contains one of the Integer Data Type identifier values listed in Table 4-4 on page 58.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the identifier is SQL_DATETIME, the date/time subtypes can be distinguished by analysing the TYPE_NAME or DATETIME_CODE columns.</td>
</tr>
<tr>
<td>TYPE_NAME</td>
<td>VARCHAR(128) NOT NULL</td>
<td>The textual name of the column’s data type, which is one of the following values: ‘CHARACTER’ ‘CHARACTER VARYING’ ‘DATE’ ‘DECIMAL’ ‘DOUBLE PRECISION’ ‘FLOAT’ ‘INTEGER’ ‘NUMERIC’ ‘REAL’ ‘SMALLINT’ ‘TIME’ ‘TIMESTAMP’</td>
</tr>
<tr>
<td>COLUMN_SIZE</td>
<td>INTEGER</td>
<td>If DATA_TYPE is SQL_CHAR or SQL_VARCHAR, then this column contains the maximum length in characters of the column. For date/time data types, this is the total number of characters required to display the value when converted to character. For numeric data types, this is</td>
</tr>
<tr>
<td>Column Name</td>
<td>Type</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>BUFFER_LENGTH</td>
<td>INTEGER</td>
<td>The number of octets that would be required to hold the column data if SQL_DEFAULT were specified (see Default Transfers on page 60). This length excludes any null terminator. For exact numeric data types, this length accounts for the decimal point and the sign.</td>
</tr>
<tr>
<td>NUM_PREC_RADIX</td>
<td>SMALLINT</td>
<td>If DATA_TYPE is an approximate numeric data type, this column contains the value 2 because COLUMN_SIZE specifies a number of bits. For exact numeric data types, this column contains the value 10 because COLUMN_SIZE specifies a number of decimal digits. Otherwise, this column is null. By combining the precision with the radix, an application can calculate the maximum number that the column can hold.</td>
</tr>
<tr>
<td>NULLABLE</td>
<td>SMALLINT NOT NULL</td>
<td>The value SQL_NULLABLE if the column accepts null values; otherwise, the value SQL_NO_NULLS.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR(254)</td>
<td>May contain descriptive information about the column.</td>
</tr>
</tbody>
</table>
| COLUMN_DEF             | VARCHAR(254)     | The column’s default value, using legal syntax for default-value in the column-definition of the CREATE TABLE statement. If the default value is a character string, then this column is that string enclosed in single quotes. If the default value is a numeric literal, then this column contains the original character representation with no enclosing single quotes. If the default value is a date-time literal, then this column contains the
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME_CODE</td>
<td>INTEGER</td>
<td>For date/time data types, this column is the subtype code, using the same values as the DATETIME_INTERVAL_CODE field of the SQL descriptor of dynamic SQL. For other data types, this column is null.</td>
</tr>
<tr>
<td>CHAR_OCTET_LENGTH</td>
<td>INTEGER</td>
<td>Contains the maximum length in octets for a character data type column. For all other data types it is null. (For single-octet character sets, this is the same as COLUMN_SIZE.)</td>
</tr>
<tr>
<td>ORDINAL_POSITION</td>
<td>INTEGER NOT NULL</td>
<td>The ordinal position of the column in the table. The first column in the table is number 1.</td>
</tr>
<tr>
<td>IS_NULLABLE</td>
<td>VARCHAR(254)</td>
<td>Contains the value 'NO' if the column is known to be not nullable, according to the rules in the ISO SQL standard; and 'YES' otherwise.</td>
</tr>
</tbody>
</table>

**RETURN VALUE**

[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

**DIAGNOSTICS**

There must not be an open cursor on StatementHandle ('24000').

If the implementation does not support qualification by catalog name, then the value of CatalogName must be a null pointer or a zero-length string ('HYC00').
APPLICATION USAGE

In languages that allow the use of pointers, the application can provide a null pointer for any
pattern-value, in order to inhibit qualification of the result set on the respective basis.

The TABLE_CAT column may be null or may contain an empty string for several reasons
discussed in Section 2.4.3 on page 22. If TABLE_CAT contains an empty string, the application
should take care to not append '.' when forming a fully-qualified object name for use in an SQL
statement.

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLCHAR *CatalogName;
SQLSMALLINT NameLength1;
SQLCHAR *SchemaName;
SQLSMALLINT NameLength2;
SQLCHAR *TableName;
SQLSMALLINT NameLength3;
SQLCHAR *ColumnName;
SQLSMALLINT NameLength4;
...
ReturnCode = SQLColumns (StatementHandle,
    CatalogName, NameLength1,
    SchemaName, NameLength2,
    TableName, NameLength3,
    ColumnName, NameLength4);
```

EXAMPLE USAGE (COBOL)

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 CATALOGNAME PICTURE X(128).
01 NAMELENGTH1 PICTURE S9(4) BINARY.
01 SCHEMANAME PICTURE X(128).
01 NAMELENGTH2 PICTURE S9(4) BINARY.
01 TABLENAME PICTURE X(128).
01 NAMELENGTH3 PICTURE S9(4) BINARY.
01 COLUMNNAME PICTURE X(128).
01 NAMELENGTH4 PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRCColumns" USING STATEMENTHANDLE,
    CATALOGNAME, NAMELENGTH1,
    SCHEMANAME, NAMELENGTH2,
    TABLENAME, NAMELENGTH3,
    COLUMNNAME, NAMELENGTH4, RETURNCODE.
```

RELATION TO STANDARDS

This function is not present in the ISO CLI Draft International Standard.
NAME

Connect — Open a connection to a server

SYNOPSIS

FUNCTION Connect
  (ConnectionHandle INTEGER,
   ServerName CHAR,
   NameLength1 SMALLINT,
   UserName CHAR,
   NameLength2 SMALLINT,
   Authentication CHAR,
   NameLength3 SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION

Calling Connect() connects the application to a server. The function has the following parameters:

ConnectionHandle (input)
  Connection handle.

ServerName (input)
  Buffer containing server name. All leading and trailing blanks are significant. This is a literal, not an identifier, and the value is not enclosed in quotes (either single or double). If ServerName is a zero-length string, a null pointer or DEFAULT, then it indicates the default server. The length of ServerName must not exceed 128 characters.

NameLength1 (input)
  Length in octets of ServerName.

UserName (input)
  Buffer containing user identifier. The implementation trims any leading and trailing spaces and, unless the result begins and ends with double quotes (in which case it is a delimited identifier—see Delimited Identifiers on page 21), converts lower-case letters to upper case.

NameLength2 (input)
  Length in octets of UserName. This must be 0 if ServerName specifies the default server; for other servers, a value of 0 means that an undefined method is used to determine the user identifier.

Authentication (input)
  Buffer containing authentication string. All leading and trailing blanks are significant.

NameLength3 (input)
  Length in octets of Authentication. This must be 0 if ServerName specifies the default server; for other servers, a value of 0 means that an undefined method is used to determine the authentication string.

Calling Connect() establishes a connection to ServerName. The mapping from ServerName (or from the default server) to a physical database is implementation-defined.

The specified server uses the values of UserName and Authentication and may apply other criteria when the application calls Connect() to determine whether to accept or reject the connection. If the server accepts the connection, then UserName becomes the name of its current user. It is implementation-defined how the server selects a default catalog and schema.

The application can establish more than one connection, but only one connection can be current at one time. The connection that Connect() establishes becomes the current connection. If
another connection is already current and there is a transaction active on that connection, then it is implementation-defined whether the connection can be switched.

**RETURN VALUE**

[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

**DIAGNOSTICS**

There must not be a connection associated with `ConnectionHandle` (‘08002’).

If `ServerName` specifies the default server, the default connection must not be current or dormant (‘08002’).

If `ServerName` specifies the default server, then `NameLength2` and `NameLength3` must be 0 (‘HY090’).

The value of `UserName` with leading and trailing spaces removed must be a valid *user-name* as defined by the X/Open SQL specification, and must not violate any implementation-defined restrictions (‘28000’).

On implementations that do not allow connections to be switched within a transaction, a transaction must not already be active on `ConnectionHandle` (‘0A001’).

The function may fail because the client cannot establish a connection (‘08001’) or because the server rejects the connection for implementation-defined reasons (‘08004’).

**EXAMPLE USAGE (C)**

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDBC ConnectionHandle;
SQLCHAR *ServerName;
SQLSMALLINT NameLength1;
SQLCHAR *UserName;
SQLSMALLINT NameLength2;
SQLCHAR *Authentication;
SQLSMALLINT NameLength3;
...
ReturnCode = SQLConnect(ConnectionHandle, ServerName, NameLength1,
                         UserName, NameLength2, Authentication, NameLength3);
```

**EXAMPLE USAGE (COBOL)**

```cobol
01 CONNECTIONHANDLE PICTURE S9(9) BINARY.
01 SERVERNAME PICTURE X(128).
01 NAMELENGTH1 PICTURE S9(4) BINARY.
01 USERNAME PICTURE X(18).
01 NAMELENGTH2 PICTURE S9(4) BINARY.
01 AUTHENTICATION PICTURE X(254).
01 NAMELENGTH3 PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRConnect" USING CONNECTIONHANDLE,
     SERVERNAME, NAMELENGTH1, USERNAME, NAMELENGTH2,
     AUTHENTICATION, NAMELENGTH3, RETURNCODE.
```
Connect()

SEE ALSO
   Disconnect()
NAME
CopyDesc — Copy a descriptor

SYNOPSIS
FUNCTION CopyDesc
(SourceDescHandle INTEGER,
 TargetDescHandle INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The CopyDesc() function has the following parameters:

SourceDescHandle (input)
Source descriptor handle.

TargetDescHandle (input)
Target descriptor handle.

The function copies the fields of the data structure associated with SourceDescHandle to the data structure associated with TargetDescHandle. Any existing data in the data structure associated with TargetDescHandle is overwritten, except that the ALLOC_TYPE field of TargetDescHandle (which indicates whether the target descriptor was allocated automatically or by the application) is not changed.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_INVALID_HANDLE] or
[SQL_SUCCESS_WITH_INFO].
[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

DIAGNOSTICS
All diagnostics are associated with TargetDescHandle.

TargetDescHandle cannot reference an implementation row descriptor (‘HY016’).

APPLICATION USAGE
Handles for the automatically-generated row and parameter descriptors of a statement can be obtained by calling GetStmtAttr().

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDESC SourceDescHandle;
SQLHDESC TargetDescHandle;
...
ReturnCode = SQLCopyDesc(SourceDescHandle, TargetDescHandle);

EXAMPLE USAGE (COBOL)
01 SOURCEDESCHANDLE PICTURE S9(9) BINARY.
01 TARGETDESCHANDLE PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRCopyDesc" USING SOURCEDESCHANDLE,
   TARGETDESCHANDLE, RETURNCODE.

SEE ALSO
AllocHandle(), GetStmtAttr(), FreeHandle()
DataSources( )

NAME
DataSources — Get a list of server names

SYNOPSIS
FUNCTION DataSources
( EnvironmentHandle INTEGER,
  Direction SMALLINT,
  ServerName CHAR,
  BufferLength1 SMALLINT,
  NameLength1 SMALLINT,
  Description CHAR,
  BufferLength2 SMALLINT,
  NameLength2 SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION
The DataSources() function retrieves names of accessible servers. The function may also return information describing these servers.

The function has the following parameters:

EnvironmentHandle (input)
Environment handle.

Direction (input)
Used by the application to control the manner in which it gains access to the list of servers. This may be one of the following:

SQL_FETCH_FIRST
The function returns information on the first server in the list.

SQL_FETCH_NEXT
The function returns information on the next sequential server in the list; except that if the application has never successfully called DataSources() on EnvironmentHandle before, or if the previous call to DataSources() on EnvironmentHandle returned [SQL_NO_DATA], then the function returns the first server in the list.

ServerName (output)
The server name.

BufferLength1 (input)
Maximum number of octets to store in ServerName.

NameLength1 (output)
Length in octets of the output server name (even if it does not entirely fit in ServerName).

Description (output)
The description associated with ServerName. If no information is available for the current server, then this is a zero-length string.

BufferLength2 (input)
Maximum number of octets to store in Description.

NameLength2 (output)
Length in octets of the output description (even if it does not entirely fit in Description).

The application can call this function at any time that EnvironmentHandle is allocated.
RETURN VALUE

[SQL_SUCCESS], [SQL_SUCCESS_WITH_INFO], [SQL_INVALID_HANDLE], [SQL_NO_DATA] or [SQL_ERROR].

The [SQL_NO_DATA] outcome occurs if there are no more server names to retrieve.

DIAGNOSTICS

The [SQL_SUCCESS_WITH_INFO] outcome occurs if a returned string value is truncated ('01004') or if implementation-defined warnings are produced.

Directed must have one of the values listed above ('HY103').

APPLICATION USAGE

The application typically calls DataSources() repeatedly to retrieve information on all servers in the list of accessible servers. The application sets Direction to SQL_FETCH_FIRST on the first call in this series, and to SQL_FETCH_NEXT on subsequent calls.

The application might display a list and description of available servers, obtained by repeated calls to DataSources(), to prompt the user to select a desired data source.

The set of servers whose names are available from DataSources() is not necessarily the complete set to which the application can connect. Moreover, the fact that DataSources() provides the name of a server does not constitute a guarantee that the application can successfully connect to the server; for example, the server may require authentication information that the application does not have.

It is undefined how server information is added to the database that DataSources() draws upon.

EXAMPLE USAGE (C)

```
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHENV EnvironmentHandle;
SQLSMALLINT Direction;
SQLCHAR *ServerName;
SQLSMALLINT BufferLength1;
SQLSMALLINT NameLength1;
SQLCHAR *Description;
SQLSMALLINT BufferLength2;
SQLSMALLINT NameLength2;
...
ReturnCode = SQLDataSources (EnvironmentHandle, Direction,
    ServerName, BufferLength1, &NameLength1, Description,
    BufferLength2, &NameLength2);
```
EXAMPLE USAGE (COBOL)

01 ENVIRONMENTHANDLE PICTURE S9(9) BINARY.
01 DIRECTION PICTURE S9(4) BINARY.
01 SERVERNAME PICTURE X(128).
01 BUFFERLENGTH1 PICTURE S9(4) BINARY.
01 NAMELENGTH1 PICTURE S9(4) BINARY.
01 DESCRIPTION PICTURE X(254).
01 BUFFERLENGTH2 PICTURE S9(4) BINARY.
01 NAMELENGTH2 PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.

... 

CALL "SQLRDatasources" USING ENVIRONMENTHANDLE, DIRECTION,
        SERVERNAME, BUFFERLENGTH1, NAMELENGTH1, DESCRIPTION,
        BUFFERLENGTH2, NAMELENGTH2, RETURNCODE.
NAME

DescribeCol — Describe a column of a result set (concise)

SYNOPSIS

FUNCTION DescribeCol

(StatementHandle INTEGER,
 ColumnNumber SMALLINT,
 ColumnName CHAR,
 BufferLength SMALLINT,
 NameLength SMALLINT,
 DataType SMALLINT,
 ColumnSize INTEGER,
 DecimalDigits SMALLINT,
 Nullable SMALLINT)

RETURNS (SMALLINT)

DESCRIPTION

The DescribeCol() function describes a selected column from a result set. The function has the following parameters:

StatementHandle (input)
  Statement handle.

ColumnNumber (input)
  Column number. Columns are numbered sequentially from left to right and the leftmost is number 1.

ColumnName (output)
  Column name.

BufferLength (input)
  Length in octets of ColumnName buffer.

NameLength (output)
  Actual number of octets in ColumnName.

DataType (output)
  The SQL data type of the column. This is one of the Integer Data Type Identifier values listed in Table 4-4 on page 58 or an implementation-defined value. If the TYPE field is SQL_DATETIME, it is SQL_TYPE_DATE, SQL_TYPE_TIME, or SQL_TYPE_TIMESTAMP, depending on the value of the DATETIME_INTERVAL_CODE field. (see also Table 4-5 on page 58).

ColumnSize (output)
  Length or precision for column's data type in the database. If DataType is SQL_CHAR or SQL_VARCHAR, then ColumnSize is the maximum length in octets of the column. For approximate numeric data types, ColumnSize is the precision in bits. For exact numeric data types, ColumnSize is the precision in decimal digits. For TIME and TIMESTAMP, ColumnSize is the precision of the fractional part of the seconds component, in decimal digits. Otherwise ColumnSize is ignored.

DecimalDigits (output)
  For SQL_DECIMAL and SQL_NUMERIC, DecimalDigits is the total number of digits to the right of the decimal point. For SQL_INTEGER and SQL_SMALLINT, DecimalDigits is 0. Otherwise DecimalDigits is undefined.

Nullable (output)
  The function sets this parameter to one of the following:
The column's definition does not allow null values.

The column's definition allows null values.

Calling DescribeCol() obtains descriptor information (describing the column's name, type and length) for column number ColumnNumber of the result set.

RETURN VALUE

[SQL_SUCCESS], [SQL_ERROR], [SQL_INVALID_HANDLE] or [SQL_SUCCESS_WITH_INFO].

DIAGNOSTICS

The [SQL_SUCCESS_WITH_INFO] outcome occurs if the column name is truncated ('01004') or if implementation-defined warnings are produced.

The application must prepare StatementHandle (by calling Prepare() or ExecDirect()) before calling DescribeCol() ('HY010').

ColumnNumber must be in the range from 1 to and including the number of columns in the result set ('07009').

The prepared statement associated with StatementHandle must be a cursor-specification ('07005').

APPLICATION USAGE

The application may retrieve column descriptions in any order by repeated calls to DescribeCol(). The application can call DescribeCol() to retrieve, in a single call, all the descriptor fields commonly used when retrieving data from the database. Conceptually, DescribeCol() performs the following steps in sequence:

1. Calls GetStmtAttr() to obtain the implementation row descriptor handle.
2. Calls GetDescField() multiple times for that handle to retrieve values from the following fields of the implementation row descriptor:
   - sets ColumnName to the value of the NAME field and sets NameLength to the value of the stringLength parameter returned by GetDescField()
   - sets DataType depending on the value of the TYPE field
   - sets ColumnSize to the value of the OCTET_LENGTH or PRECISION field (as appropriate for the TYPE field)
   - sets DecimalDigits to the value of the SCALE field
   - sets Nullable to the value of the NULLABLE field.

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLSMALLINT ColumnNumber;
SQLCHAR *ColumnName;
SQLSMALLINT BufferLength;
SQLSMALLINT NameLength;
SQLSMALLINT DataType;
SQLINTEGER ColumnSize;
SQLSMALLINT DecimalDigits;
SQLSMALLINT Nullable;
```
... 
ReturnCode = SQLDescribeCol(StatementHandle, 
    ColumnNumber, ColumnName, BufferLength, &NameLength, 
    &DataType, &ColumnSize, &DecimalDigits, &Nullable);

EXAMPLE USAGE (COBOL)
  01 STATEMENTHANDLE PICTURE S9(9) BINARY.
  01 COLUMNNUMBER PICTURE S9(4) BINARY.
  01 COLUMNNAME PICTURE X(128).
  01 BUFFERLENGTH PICTURE S9(4) BINARY.
  01 NAMELENGTH PICTURE S9(4) BINARY.
  01 DATATYPE PICTURE S9(4) BINARY.
  01 COLUMNSIZE PICTURE S9(9) BINARY.
  01 DECIMALDIGITS PICTURE S9(4) BINARY.
  01 NULLABLE PICTURE S9(4) BINARY.
  01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRDescribeCol" USING STATEMENTHANDLE, COLUMNNUMBER, 
    COLUMNNAME, BUFFERLENGTH, NAMELENGTH, DATATYPE, 
    COLUMNSIZE, DECIMALDIGITS, NULLABLE, RETURNCODE.

SEE ALSO
   BindCol(), Fetch(), FetchScroll(), GetDescField(), GetDescRec(), NumResultCols()
Disconnect() 

NAME

Disconnect — Close a connection to a server

SYNOPSIS

FUNCTION Disconnect
( ConnectionHandle INTEGER)
RETURNS (SMALLINT)

DESCRIPTION

The Disconnect() function has the following parameter:

ConnectionHandle (input)
Connection handle.

The function closes the connection associated with ConnectionHandle. Disconnecting a connection that is not the current connection makes no change to the context of the current connection.

If an application calls Disconnect() before freeing all statement handles and descriptor handles associated with ConnectionHandle, any such handles that are still allocated are freed if Disconnect() successfully returns, but are not freed if the disconnection fails (for example, if the current transaction is incomplete).

RETURN VALUE

[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

DIAGNOSTICS

The transaction associated with ConnectionHandle cannot be active ('25000').

[SQL_SUCCESS_WITH_INFO] can occur if errors occur performing the disconnection ('01002'), or if the connection was disconnected before the call to Disconnect() because of an event that occurred independently of the application ('01002'), or if implementation-defined warnings are produced.

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDBC ConnectionHandle;
...
ReturnCode = SQLDisconnect(ConnectionHandle);
```

EXAMPLE USAGE (COBOL)

```cobol
01 CONNECTIONHANDLE PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRDisconnect" USING CONNECTIONHANDLE, RETURNCODE.
```

SEE ALSO

Connect()
NAME
EndTran — Commit or roll back a transaction

SYNOPSIS
FUNCTION EndTran
(HandleType SMALLINT,
Handle INTEGER,
CompletionType SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION
The EndTran() function completes (commits or rolls back) a transaction, enacting or undoing any changes to the database performed on any connection within Handle since the transaction began (see Section 4.9 on page 63).

The function has the following parameters:
HandleType (input)
A handle type identifier. It must contain SQL_HANDLE_ENV if Handle is an environment handle, or SQL_HANDLE_DBC if Handle is a connection handle.

Handle (input)
The handle indicating the scope of work to be completed.

CompletionType (input)
The desired completion outcome. This may be one of the following:
SQL_COMMIT
Work done is to be committed.

SQL_ROLLBACK
Work done is to be rolled back.

Successful completion of a transaction closes any open cursors.

It is implementation-defined whether a transaction can span connections. On an implementation where transactions can span connections, the following aspects of coordination of multiple-connection transactions are implementation-defined:
• the sequence in which the connections have their transaction completed
• the effect that the success or failure of commitment over one connection has on the success or failure over any other.

If the work at some connections is committed and the work at others is rolled back, administrative action is typically necessary to assess or restore the integrity of the database.

The effect of calling EndTran() is implementation-defined if HandleType is SQL_HANDLE_DBC (see Effects across Connections on page 63).

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

DIAGNOSTICS
HandleType must be SQL_HANDLE_ENV or SQL_HANDLE_DBC (HY092).
CompletionType must have one of the values listed above (’HY012’).

The function may fail because the connection fails (’08007’). In this case, the application might not be able to determine whether the requested transaction outcome took effect before the failure.

The server can decide to roll back the transaction even though the call to EndTran() specified SQL_COMMIT. (The SQLSTATE class is ’40’ and the subclass is one of those specified in Appendix A.)

Errors for Global Transaction Control
The EndTran() function may fail for any of the following reasons:

- The implementation is not able to guarantee that all work in the global transaction can be completed atomically. Under implementation-defined criteria, either the transaction is still active (’25S02’) or all work in the transaction active in Handle is rolled back (’25S03’).
- One or more of the connections in Handle fails to complete the transaction with the outcome the application specified (’25S01’). The transaction state is unknown.

The application must not complete by calling EndTran() a global transaction it started by calling a function from an explicit transaction demarcation API, such as the X/Open Distributed Transaction Processing (DTP) function tx_begin() (’2D000’).

APPLICATION USAGE
X/Open Distributed Transaction Processing (DTP) documents describe methods of coordinating databases so that changes to any recoverable resources are committed or rolled back atomically. For example, the emerging Transaction Demarcation API specifies functions by which an application declares the start, commitment or rollback of a global transaction. An application may use these functions in place of the implicit start of transactions described above, and in place of Transact(), in order to coordinate SQL work with non-SQL work.

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLSMALLINT HandleType;
SQLHENV Handle; /* or SQLHDBC */
SQLSMALLINT CompletionType;
...
ReturnCode = SQLEndTran(HandleType, Handle, CompletionType);

EXAMPLE USAGE (COBOL)
01 HANDLETYPE PICTURE S9(4) BINARY.
01 HANDLE PICTURE S9(9) BINARY.
01 COMPLETIONTYPE PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLREndTran" USING HANDLETYPE, HANDLE,
COMPLETIONTYPE, RETURNCODE.

RELATION TO STANDARDS
The ability to specify SQL_HANDLE_HDBC as HandleType represents an extension to the ISO CLI Draft International Standard, which only permits EndTran() to be applied to the entire environment.
NAME
Error — Return error information associated with a handle

SYNOPSIS

```plaintext
FUNCTION Error
  (EnvironmentHandle INTEGER,
   ConnectionHandle INTEGER,
   StatementHandle INTEGER,
   Sqlstate CHAR,
   NativeError INTEGER,
   MessageText CHAR,
   BufferLength SMALLINT,
   TextLength SMALLINT)
RETURNS (SMALLINT)
```

DESCRIPTION

An application calls `Error()` to obtain several commonly-used pieces of error or warning status from a previous call to a CLI function, especially one that returned [SQL_ERROR] or [SQL_SUCCESS_WITH_INFO]. (This function is referred to below as the underlying function.)

The data structure associated with each handle for an environment, connection and statement contains a diagnostics area. The area consists of header information and zero or more status records. Each status record includes values of `Sqlstate`, `NativeError` and `MessageText / TextLength`. Each status record describes a specific event in the execution of the underlying function.

The first three parameters to `Error()` specify a single handle from which the application seeks information from the diagnostic area:

- `EnvironmentHandle [input]` Environment handle.
- `ConnectionHandle [input]` Connection handle.
- `StatementHandle [input]` Statement handle.

To obtain status on an SQL statement, pass a valid statement handle as `StatementHandle`. To obtain status on a connection, set `StatementHandle` to SQL_NULL_HSTMT and pass a valid connection handle as `ConnectionHandle`. To obtain status on an environment, set `StatementHandle` to SQL_NULL_HSTMT, set `ConnectionHandle` to SQL_NULL_HDBC and pass a valid environment handle as `EnvironmentHandle`.

The remaining parameters convey the status information to the application:

- `Sqlstate [output]` A string of 5 characters. The first 2 indicate error class; the next 3 indicate subclass. The values are identical to SQLSTATE values in the X/Open SQL specification.
- `NativeError [output]` An implementation-defined error code. Portable applications should not base their behaviour on the value of `NativeError`.
- `MessageText [output]` Implementation-defined message text. Its length in octets never exceeds [SQL_MAX_MESSAGE_LENGTH] minus the number of octets in the null terminator if applicable.
BufferLength [input]
   Maximum number of octets to store in MessageText.

TextLength [output]
   Length in octets of the output data (even if it does not entirely fit in MessageText).

RETURN VALUE
   The function uses the following return values (instead of generated diagnostics) to report
   the outcome of its own execution:

   [SQL_SUCCESS]
      The function successfully returned error information.

   [SQL_INVALID_HANDLE]
      The combination of EnvironmentHandle, ConnectHandle and StatementHandle did not specify
      a valid handle.

   [SQL_ERROR]
      The BufferLength argument is negative (or BufferLength is zero and the environment attribute
      SQL_ATTR_OUTPUT_NTS is true).

   [SQL_NO_DATA]
      No further diagnostic records exist for the specified handle.

The function does not return [SQL_SUCCESS_WITH_INFO] even if it truncates a string output
argument. No implementation-defined diagnostics are possible.

DIAGNOSTICS
   None. This function never generates information in any handle's diagnostic area to describe its
   own execution. This function is never the underlying function as defined in DESCRIPTION
   above. Any diagnostics are reported using the RETURN VALUE (see above).

APPLICATION USAGE
   Production of status records by the underlying function is discussed in Section 5.2 on page 69.

   The application obtains one status record at a time by each successive call to Error(). Each such
   call returns [SQL_SUCCESS], returns the status record and removes that record from the list of
   records available via Error() using the specified handle.

   If a valid call to Error() does not obtain a status record, either because previous calls to Error()
   obtained them all or because the underlying CLI function for the specified handle returned
   success or [SQL_NO_DATA], then Error() returns [SQL_NO_DATA]; Sqlstate is '00000',
   NativeError, MessageText, and TextLength are undefined. However, if the call to Error() returns
   [SQL_ERROR] or [SQL_INVALID_HANDLE], all output parameters are undefined.

   The Error function retrieves only diagnostic information most recently associated with the
   specified handle. If an application calls another CLI function that writes diagnostic
   information to the same handle (even a success indication), any diagnostic information from
   previous calls is lost.

   In languages that allow the use of pointers, the application can provide a null pointer for any of
   Sqlstate, NativeError and MessageText, in order to inhibit return of that information.

   An application can call Error(), GetDiagField() or GetDiagRec() to obtain status information for a
   handle. These functions are implemented so that they do not interfere with the sequence of
   status records that the other functions return.
Even though the message text field in the X/Open SQL diagnostic area is specified as CHAR(254), some implementations may generate more than 254 octets of message text. A portable application should set aside a buffer whose length in octets is [SQL_MAX_MESSAGE_LENGTH].

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHENV EnvironmentHandle;
SQLHDBC ConnectionHandle;
SQLHSTMT StatementHandle;
SQLCHAR Sqlstate[6];
SQLINTEGER NativeError;
SQLCHAR *MessageText;
SQLSMALLINT BufferLength;
SQLSMALLINT TextLength;
...
ReturnCode = SQLError(EnvironmentHandle, ConnectionHandle,
    StatementHandle, Sqlstate, &NativeError,
    MessageText, BufferLength, &TextLength);
```

EXAMPLE USAGE (COBOL)

```cobol
01 ENVIRONMENTHANDLE PICTURE S9(9) BINARY.
01 CONNECTIONHANDLE PICTURE S9(9) BINARY.
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 SQLSTATE PICTURE X(5).
01 NATIVEERROR PICTURE S9(9) BINARY.
01 MESSAGETEXT PICTURE X(512).
01 BUFFERLENGTH PICTURE S9(4) BINARY.
01 TEXTLENGTH PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRError" USING ENVIRONMENTHANDLE, CONNECTIONHANDLE,
    STATEMENTHANDLE, SQLSTATE, NATIVEERROR, MESSAGETEXT,
    BUFFERLENGTH, TEXTLENGTH, RETURNCODE.
```

SEE ALSO

GetDiagField(), GetDiagRec(), RowCount()
NAME
ExecDirect — Execute an SQL statement directly

SYNOPSIS
FUNCTION ExecDirect
  (StatementHandle INTEGER, StatementText CHAR, TextLength INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The ExecDirect() function effects a one-time execution of an SQL statement. The function has the following parameters:

- **StatementHandle** (input)
  - Statement handle.

- **StatementText** (input)
  - SQL text string, using question-mark (?) characters as dynamic parameter markers.

- **TextLength** (input)
  - Length in octets of StatementText.

Calling the function executes the SQL statement in StatementText, using the current values of any variables bound to dynamic parameters. These values may be converted before being used as dynamic arguments depending on the TYPE field of the corresponding descriptor record. If the TYPE field is SQL_DEFAULT, then the value is converted from a type given by Table 4-9 on page 61 based on the data type of the dynamic parameter.

ExecDirect() populates the implementation row descriptor with information describing the columns of the result set.

### Dynamic Parameters

**OP**
It is implementation-defined which of the following occurs during a call to ExecDirect():

- The implementation sets the fields of the implementation parameter descriptor to describe the number, data type and type attributes of dynamic parameters in the SQL statement (this corresponds to the DESCRIBE INPUT feature of embedded SQL). If there are no dynamic parameters, the descriptor's COUNT field is set to 0.

- The implementation does not modify the implementation parameter descriptor.

Section 4.3.1 on page 47 describes the cases in which the application must gain access to the implementation parameter descriptor.

For a cursor-specification, ExecDirect() generates a cursor name, unless one already exists, and opens the cursor.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_INVALID_HANDLE], [SQL_NEED_DATA], [SQL_SUCCESS_WITH_INFO] or [SQL_NO_DATA].

DIAGNOSTICS
[SQL_SUCCESS_WITH_INFO] can result if the parameter data is truncated ('01004') or when implementation-defined warnings are produced.

There must not be an open cursor on StatementHandle ('24000').
If the application has specified an application parameter descriptor, its COUNT field must be non-negative (‘07008’).

The CLI implementation evaluates dynamic arguments only when it executes the SQL statement. At this time:

- The application must have provided a dynamic argument for each dynamic parameter in the SQL statement using the application parameter descriptor (‘07001’).
- For character-string dynamic arguments, the variable indicated by the LENGTH_PTR field (from which the implementation determines the argument’s length as specified in Section 4.3 on page 46) must be valid (‘22005’).
- The variable indicated by the DATA_PTR field (which contains the actual data to use as the dynamic argument) must be in the stored format specified by the TYPE and related fields. This checking may produce the diagnostics specified in Section 5.4.10 on page 76.
- If the application has set any fields of the implementation parameter buffer, the types and type attributes defined there must be compatible with the corresponding fields of the application parameter buffer. The permitted combinations are defined in Section 4.8.3 on page 60 (‘07006’).

**Statement Preparation Diagnostics**

The statement text must be a valid SQL statement other than COMMIT or ROLLBACK (‘42000’).

For a dynamic positioned DELETE or a dynamic positioned UPDATE statement, the cursor referenced by the statement must be open and must be defined on a separate statement handle under the same connection handle (‘34000’).

The following diagnostics, defined in the X/Open SQL specification, can occur based on the value of `StatementText`:

**Cardinality violation**
- ‘21S01’ — Insert value does not match column list.
- ‘21S02’ — Degree of derived table does not match column list.
- ‘42000’ **Syntax error or access violation†**
- ‘42S01’ — Base table or view already exists.
- ‘42S02’ — Base table or view not found.
- ‘42S11’ — Index already exists.
- ‘42S12’ — Index not found.
- ‘42S21’ — Column already exists.
- ‘42S22’ — Column not found.

---

20. **Statement Preparation Diagnostics** are those also listed for `Prepare()`; **Statement Execution Diagnostics** are those also listed for `Execute()`. The classification is not normative and the distinction is not visible to the caller of `ExecDirect()`.

† The codes of the form ‘42Sn’ are ‘S00nn’ in the X/Open SQL specification.
Statement Execution Diagnostics

The SQL statement must not contain a dynamic parameter (‘07001’) or literal (‘42000’) whose value is incompatible with the data type of the associated table column.

A character string assigned to a character-string table column must not exceed the maximum length in characters of the column (‘22001’).

The following diagnostics, defined in the X/Open SQL specification, can occur based on the value of *StatementText*:

- **Success with warning** [SQL_SUCCESS_WITH_INFO]
  - ‘01006’ — Privilege not revoked.
  - ‘01007’ — Privilege not granted.
- **Data exception**
  - ‘22012’ — Division by zero.
  - ‘23000’ **Integrity constraint violation**
  - ‘24000’ **Invalid cursor state**

Execution of the SQL statement may produce the data conversion diagnostics specified in Section 5.4.10 on page 76.

**APPLICATION USAGE**

The function returns [SQL_NO_DATA] if the SQL statement specified is searched DELETE, INSERT or searched UPDATE and the row count (see Section 5.3.2 on page 72) is zero.

The return value [SQL_NEED_DATA] asks the application to supply dynamic arguments by calling *ParamData()* and *PutData()*, as described in Section 4.3.2 on page 48.

**EXAMPLE USAGE (C)**

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLCHAR *StatementText;
SQLINTEGER TextLength;
...
ReturnCode = SQLExecDirect(StatementHandle,
                           StatementText, TextLength);
```

**EXAMPLE USAGE (COBOL)**

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 STATEMENTTEXT PICTURE type.
01 TEXTLENGTH PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRExecDirect" USING STATEMENTHANDLE,
     STATEMENTTEXT, TEXTLENGTH, RETURNCODE.
```

**SEE ALSO**

NAME

Execute — Execute a prepared SQL statement

SYNOPSIS

FUNCTION Execute
   (StatementHandle INTEGER)
RETURNS (SMALLINT)

DESCRIPTION

The Execute() function executes a prepared SQL statement. The function has the following parameter:

   StatementHandle (input)
     Statement handle.

Calling Execute() executes StatementHandle, as prepared by a previous call to Prepare(), using the current values of any variables bound to dynamic parameters. These values may be converted before being used as dynamic arguments depending on the TYPE field of the corresponding descriptor record. If the TYPE field is SQL_DEFAULT, then the value is converted from a type given by Table 4-9 on page 61 based on the data type of the dynamic parameter.

For a cursor-specification, calling Execute() opens the cursor using the cursor name generated by Prepare() or SetCursorName().

RETURN VALUE

[SQL_SUCCESS], [SQL_ERROR], [SQL_INVALID_HANDLE], [SQL_NEED_DATA] [SQL_SUCCESS_WITH_INFO] or [SQL_NO_DATA].

DIAGNOSTICS

[SQL_SUCCESS_WITH_INFO] can result if the parameter data is truncated ('01004') or when implementation-defined warnings are produced.

The application must have successfully prepared StatementHandle (by calling Prepare()) before the call to Execute() ('HY010'). Certain DIAGNOSTICS noted by * in the Prepare() manual page are not detected on some implementations until the call to Execute().

There must not be an open cursor on StatementHandle ('24000').

If the application has specified an application parameter descriptor, its COUNT field must be non-negative ('07008').

The CLI implementation evaluates dynamic arguments only when it executes the SQL statement. At this time:

• The application must have provided a dynamic argument for each dynamic parameter in the SQL statement using the application parameter descriptor ('07001').

• For character-string dynamic arguments, the variable indicated by the LENGTH_PTR field (from which the implementation determines the argument’s length as specified in Section 4.3 on page 46) must be valid ('22005').

• The variable indicated by the DATA_PTR field (which contains the actual data to use as the dynamic argument) must be in the stored format specified by the TYPE and related fields. This checking may produce the diagnostics specified in Section 5.4.10 on page 76.

• If the application has set any fields of the implementation parameter buffer, the types and type attributes defined there must be compatible with the corresponding fields of the application parameter buffer. The permitted combinations are defined in Section 4.8.3 on page 60 ('07006').
A character string assigned to a character-string table column must not exceed the maximum length in characters of the column (‘22001’).

The SQL statement must not contain a dynamic parameter (‘07001’) or literal (‘42000’) whose value is incompatible with the data type of the associated table column.

The following diagnostics, defined in the X/Open SQL specification, can occur based on the value of StatementText:

- **Success with warning [SQL_SUCCESS_WITH_INFO]**
  - ‘01006’ — Privilege not revoked.
  - ‘01007’ — Privilege not granted.

- **Data exception**
  - ‘22012’ — Division by zero.
  - ‘23000’ — Integrity constraint violation
  - ‘24000’ — Invalid cursor state

Execution of the SQL statement may produce the data conversion diagnostics specified in Section 5.4.10 on page 76.

**APPLICATION USAGE**

The function returns [SQL_NO_DATA] if the SQL statement specified is searched DELETE or searched UPDATE and the row count (see Section 5.3.2 on page 72) is zero.

After the application processes (or discards) all results, it can call `Execute()` again, with the same or different dynamic arguments.

The return value [SQL_NEED_DATA] asks the application to supply dynamic arguments by calling `ParamData()` and `PutData()`, as described in Section 4.3.2 on page 48.

**EXAMPLE USAGE (C)**

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
...
ReturnCode = SQLExecute(StatementHandle);
```

**EXAMPLE USAGE (COBOL)**

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRExecute" USING STATEMENTHANDLE, RETURNCODE.
```

**SEE ALSO**

`CloseCursor()`, `ExecDirect()`, `Fetch()`, `FetchScroll()`, `ParamData()`, `Prepare()`, `PutData()`, `SetCursorName()`
NAME

Fetch — Fetch the next row of a result set

SYNOPSIS

FUNCTION Fetch
   (StatementHandle INTEGER)
RETURNS (SMALLINT)

DESCRIPTION

The Fetch() function has the following parameter:

StatementHandle (input)
   Statement handle.

When a cursor-specification has been executed on StatementHandle, the application can call Fetch() to fetch the next row from the result set. This modifies any bound variables to reflect the values in the new row.

Data conversion of column data may occur before assignment to the bound variables depending on the TYPE field of the corresponding descriptor record. If the TYPE field is SQL_DEFAULT, then the column data is converted to a type given by Table 4-9 on page 61 based on the data type of the column.

If the row contains more columns than have been bound, fetching the row also makes unbound columns available for retrieval using GetData().

RETURN VALUE

[SQL_SUCCESS], [SQL_INVALID_HANDLE], [SQL_ERROR], [SQL_NO_DATA] or [SQL_SUCCESS_WITH_INFO].

The [SQL_NO_DATA] outcome occurs if there is no next row of the result set. In the case of [SQL_ERROR] or [SQL_NO_DATA], the resulting values of any bound variables are undefined.

DIAGNOSTICS

The [SQL_SUCCESS_WITH_INFO] outcome occurs if a returned string value is truncated ('01004') or if implementation-defined warnings are produced.

The last SQL statement executed on StatementHandle must be a cursor-specification ('24000').

If the application has specified an application row descriptor:

• Its COUNT field must be non-negative ('07008') and must not exceed the number of columns of the result set ('07002').

• The data type of each column must be assignment-compatible with the buffer type specified by the corresponding descriptor record ('07006'). Retrieving a value may produce the data conversion diagnostics described in Section 5.4.10 on page 76.

• If the column value for any bound column is null, the INDICATOR_PTR field of the corresponding descriptor record must not be a null pointer ('22002').

Retrieving a value from an arithmetic expression, and applying any specified conversion, must not result in a division by zero ('22012'), either at the time of assignment or in computing an intermediate result.

The function can fail if the server terminates the current transaction to prevent deadlock.
('40001'). In this case, the transaction is rolled back.

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
...
ReturnCode = SQLFetch(StatementHandle);

EXAMPLE USAGE (COBOL)
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRFetch" USING STATEMENTHANDLE, RETURNCODE.

SEE ALSO
Execute(), ExecDirect(), FetchScroll(), GetData()
NAME
FetchScroll — Move the cursor to, and fetch, a specified row of a result set

SYNOPSIS
FUNCTION FetchScroll
(StatementHandle INTEGER,
FetchOrientation SMALLINT,
FetchOffset INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The Fetch() function has the following parameters:

StatementHandle (input)
Statement handle.

FetchOrientation (input)
The cursor positioning operation to be performed before fetching a row of the result set.
This can be one of the following:

SQL_FETCH_NEXT
Move to the row following the current cursor position.

SQL_FETCH_FIRST
Move to the first row of the result set.

SQL_FETCH_LAST
Move to the last row of the result set.

SQL_FETCH_PRIOR
Move to the row preceding the current cursor position.

SQL_FETCH_ABSOLUTE
If FetchOffset is positive, move to row number FetchOffset of the result set, where the first row is number 1.
If FetchOffset is negative, it specifies a move relative to the last row of the result set, which is row −1. If the cardinality of the result set is n, then the cursor moves to row n + FetchOffset + 1. For example, if FetchOffset is −5 in a result set with 100 rows, the cursor moves to row 100 + (−5) + 1 or 96.
For the case where FetchOffset = 0, see RETURN VALUE below.

SQL_FETCH_RELATIVE
If FetchOffset is positive, advance the cursor that number of rows.
If FetchOffset is negative, back up the cursor −FetchOffset rows.
If FetchOffset is zero, do not move the cursor.

FetchOffset
When FetchOrientation is SQL_FETCH_ABSOLUTE, this is an absolute row number of the result set, as described above. When FetchOrientation is SQL_FETCH_RELATIVE, this is an offset to the current cursor position, as described above. Otherwise, FetchOffset is ignored.

When a cursor-specification has been executed on StatementHandle, the application can call FetchScroll() to move the cursor as specified by FetchOrientation and FetchOffset, and to fetch from the result set the row to which the cursor then points. This modifies any bound variables to reflect the values in the new row.
Data conversion of column data may occur before assignment to the bound variables depending on the TYPE field of the corresponding descriptor record. If the TYPE field is SQL_DEFAULT, then the column data is converted to a type given by Table 4-9 on page 61 based on the data type of the column.

If the row contains more columns than have been bound, fetching the row also makes unbound columns available for retrieval using GetData().

**RETURN VALUE**

[SQL_SUCCESS], [SQL_INVALID_HANDLE], [SQL_ERROR], [SQL_NO_DATA] or [SQL_SUCCESS_WITH_INFO].

The [SQL_NO_DATA] outcome occurs if there is no such row as the specified row of the result set. This generally indicates one of the following cases where the specified cursor motion exceeded the bounds of the result set:21

- An absolute fetch when FetchOffset is 0, or when the absolute value of FetchOffset is greater than the number of rows of the result set.
- A relative fetch, including SQL_FETCH_NEXT and SQL_FETCH_PRIOR, where the specified number of rows to move is more than the remaining rows of the result set in the specified direction.

In the case of [SQL_ERROR] or [SQL_NO_DATA], the resulting values of any bound variables are undefined.

**DIAGNOSTICS**

The [SQL_SUCCESS_WITH_INFO] outcome occurs if a returned string value is truncated ('01004') or if implementation-defined warnings are produced.

The last SQL statement executed on StatementHandle must be a cursor-specification ('24000').

**EX OP** FetchOrientation must be one of the valid values. If FetchOrientation is any value other than SQL_FETCH_NEXT, the application must set the SQL_ATTR_CURSOR_SCROLLABLE statement attribute to SQL_TRUE before preparing or executing StatementHandle('HY106').

If the application has specified an application row descriptor:

- Its COUNT field must be non-negative ('07008') and must not exceed the number of columns of the result set ('07002').
- The data type of each column must be assignment-compatible with the buffer type specified by the corresponding descriptor record ('07006'). Retrieving a value may produce the data conversion diagnostics described in Section 5.4.10 on page 76.
- If the column value for any bound column is null, the INDICATOR_PTR field of the corresponding descriptor record must not be a null pointer ('22002').

Retrieving a value from an arithmetic expression, and applying any specified conversion, must not result in a division by zero ('22012'), either at the time of assignment or in computing an intermediate result.

---

21. The cursor position resulting from the [SQL_NO_DATA] outcome is before the first row of the result set if the specified cursor movement was excessive and backward; or after the last row of the result set if the specified cursor movement was excessive and forward. When the application specifies an absolute move to the nonexistent row 0, the cursor moves before the first row.
The function can fail if the server terminates the current transaction to prevent deadlock ('40001'). In this case, the transaction is rolled back.

**EXAMPLE USAGE (C)**

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLSMALLINT FetchOrientation;
SQLINTEGER FetchOffset;
...
ReturnCode = SQLFetchScroll(StatementHandle, FetchOrientation, FetchOffset);
```

**EXAMPLE USAGE (COBOL)**

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 FETCHORIENTATION PICTURE S9(4) BINARY.
01 FETCHOFFSET PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRFetchScroll" USING STATEMENTHANDLE, FETCHORIENTATION, FETCHOFFSET, RETURNCODE.
```

**SEE ALSO**

`Execute()`, `ExecDirect()`, `Fetch()`, `GetData()`
NAME
FreeConnect — Free a connection handle (deprecated)

SYNOPSIS
FUNCTION FreeConnect
   (ConnectionHandle INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The FreeConnect function frees the connection handle ConnectionHandle. The function has the following parameter:

ConnectionHandle (input)
The connection handle to be freed.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR] or [SQL_INVALID_HANDLE].
If FreeConnect() returns [SQL_ERROR], ConnectionHandle is still valid.

DIAGNOSTICS
The application must call Disconnect() before trying to free a connection handle ('HY010').

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDBC ConnectionHandle;
...
ReturnCode = SQLFreeConnect(ConnectionHandle);

EXAMPLE USAGE (COBOL)
01 CONNECTIONHANDLE PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRFreeConnect" USING CONNECTIONHANDLE, RETURNCODE

SEE ALSO
AllocConnect(), AllocHandle(), FreeHandle(), Disconnect()
NAME
FreeEnv — Free an environment handle (deprecated)

SYNOPSIS

```c
FUNCTION FreeEnv
    (EnvironmentHandle INTEGER)
RETURNS (SMALLINT)
```

DESCRIPTION
The `FreeEnv` function frees `EnvironmentHandle`. The function has the following parameter:

`EnvironmentHandle (input)`
The environment handle to be freed.

RETURN VALUE

- `[SQL_SUCCESS]`, `[SQL_ERROR]` or `[SQL_INVALID_HANDLE]`
- If `FreeEnv()` returns `[SQL_ERROR]`, `EnvironmentHandle` is still valid.

DIAGNOSTICS
The application must release all subsidiary handles and other resources before trying to free an environment handle ("HY010").

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHENV EnvironmentHandle;
...
ReturnCode = SQLFreeEnv(EnvironmentHandle);
```

EXAMPLE USAGE (COBOL)

```cobol
01 ENVIRONMENTHANDLE PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRFreeEnv" USING ENVIRONMENTHANDLE, RETURNCODE.
```

SEE ALSO
AllocEnv(), AllocHandle(), FreeHandle()
NAME
FreeHandle — Free a handle

SYNOPSIS
FUNCTION FreeHandle
(HandleType SMALLINT,
Handle INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The FreeHandle() function has the following parameters:

HandleType (input)
A handle type identifier that describes the handle type of Handle. It must contain one of the
following values:

SQL_HANDLE_ENV
FreeHandle() frees the environment handle Handle.

SQL_HANDLE_DBC
FreeHandle() frees the connection handle Handle.

SQL_HANDLE_STMT
FreeHandle() frees the statement handle Handle, closing any open cursor on Handle. It
also frees any automatically-generated descriptors associated with Handle.

SQL_HANDLE_DESC
FreeHandle() frees the descriptor handle Handle. FreeHandle() does not release any
memory allocated by the application that may be referenced by the deferred fields of
any descriptor record of Handle.

Handle (input)
The handle to be freed.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR] or [SQL_INVALID_HANDLE].
If FreeHandle() returns [SQL_ERROR], Handle is still valid.

DIAGNOSTICS
HandleType must be one of the values defined above (‘HY092’).
The application must release all subsidiary handles and other resources before trying to free a
handle (‘HY010’). For example, the application must call Disconnect(), freeing all subsidiary
handles, before trying to free a connection handle.
Handle must not identify an automatically-generated descriptor (‘HY017’).

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCodeEnv;
SQLRETURN ReturnCodeDbc;
SQLRETURN ReturnCodeStmt;
SQLRETURN ReturnCodeDesc;
SQLHENV EnvironmentHandle;
SQLDBC ConnectionHandle;
SQLHSTMT StatementHandle;
SQLHDESC DescriptorHandle;
...
ReturnCodeStmt = SQLFreeHandle(SQL_HANDLE_STMT, StatementHandle);
...
ReturnCodeDesc = SQLFreeHandle(SQL_HANDLE_DESC, DescriptorHandle);
...
ReturnCodeDbc = SQLFreeHandle(SQL_HANDLE_DBC, ConnectionHandle);
...
ReturnCodeEnv = SQLFreeHandle(SQL_HANDLE_ENV, EnvironmentHandle);

**EXAMPLE USAGE (COBOL)**

01 ENVIRONMENTHANDLE PICTURE S9(9) BINARY.
01 CONNECTIONHANDLE PICTURE S9(9) BINARY.
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 DESCRIPTORHANDLE PICTURE S9(9) BINARY.
01 RETURNCODEENV PICTURE S9(4) BINARY.
01 RETURNCODEDBC PICTURE S9(4) BINARY.
01 RETURNCODESTMT PICTURE S9(4) BINARY.
01 RETURNCODEDESC PICTURE S9(4) BINARY.
...
CALL "SQLRFreeHandle" USING SQL-HANDLE-STMT, STATEMENTHANDLE, RETURNCODESTMT.
...
CALL "SQLRFreeHandle" USING SQL-HANDLE-DESC, DESCRIPTORHANDLE, RETURNCODEDESC.
...
CALL "SQLRFreeHandle" USING SQL-HANDLE-DBC, CONNECTIONHANDLE, RETURNCODEDBC.
...
CALL "SQLRFreeHandle" USING SQL-HANDLE-ENV, ENVIRONMENTHANDLE, RETURNCODEENV.

**SEE ALSO**

*AllocHandle()*
NAME
FreeStmt — Free a statement handle (deprecated)

SYNOPSIS
FUNCTION FreeStmt
  (StatementHandle INTEGER, Option SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION
The FreeStmt() function frees the statement handle StatementHandle or performs other operations on StatementHandle as specified by Option. The function has the following parameters:

StatementHandle (input)
The statement handle to be processed.

Option (input)
An option parameter that specifies the function to be performed on StatementHandle. This must be one of the following:

SQL_CLOSE
Closes any cursor associated with StatementHandle and discards any pending results. The application can re-execute the statement associated with the cursor, with the same or different dynamic arguments. If no open cursor is associated with StatementHandle, this option has no effect.

SQL_DROP
Frees StatementHandle, closes any open cursor on StatementHandle and frees any automatically-generated descriptors associated with StatementHandle.

SQL_UNBIND
Unbinds all descriptor records in the application row descriptor associated with StatementHandle.

SQL_RESET_PARAMS
Unbinds all descriptor records in the application parameter descriptor associated with StatementHandle.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR] or [SQL_INVALID_HANDLE]. If FreeStmt() returns [SQL_ERROR], StatementHandle is still valid.

DIAGNOSTICS
Option must be one of the defined options (‘HY092’).

EXAMPLE USAGE(C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLSMALLINT Option;
...
ReturnCode = SQLFreeStmt(StatementHandle, Option);

EXAMPLE USAGE (COBOL)
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 OPTION PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRFreeStmt" USING STATEMENTHANDLE, OPTION, RETURNCODE.

SEE ALSO
AllocHandle(), AllocStmt(), FreeHandle()
NAME
GetConnectAttr — Get the value of a connection attribute

SYNOPSIS
FUNCTION GetConnectAttr
(ConnectionHandle INTEGER,
Attribute INTEGER,
Value ANY,
BufferLength INTEGER,
StringLength INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The application can call GetConnectAttr() to obtain a value of an attribute for ConnectionHandle.

The function has the following parameters:
ConnectionHandle (input)
Connection handle.
Attribute (input)
The desired connection attribute. For a list of valid attributes, see Connection Attributes on page 29.
Value (output)
The current value of the attribute specified by Attribute. The type of the value returned depends on Attribute.
BufferLength (input)
Maximum number of octets to store in Value, if the attribute value is a character string; otherwise, unused.
StringLength (output)
Length in octets of the output data (even if it does not entirely fit in Value), if the attribute value is a character string; otherwise, unused.

If Attribute does not denote a string, then the implementation ignores BufferLength and does not set StringLength.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

DIAGNOSTICS
The [SQL_SUCCESS_WITH_INFO] outcome occurs if the returned data is truncated (’01004’) or if implementation-defined warnings are produced.

Attribute must be one of the defined attributes or an implementation-defined value (’HY092’).

If Attribute is SQL_ATTR_AUTO_IPD, then a connection must be established on ConnectionHandle (’08003’).

APPLICATION USAGE
GetConnectAttr() can be called at any time between the allocation and the freeing of ConnectionHandle. It obtains the current value of the connection attribute.

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDBC ConnectionHandle;
SQLINTEGER Attribute;
SQLPOINTER Value;
SQLINTEGER BufferLength;
SQLINTEGER StringLength;
...
ReturnCode = SQLGetConnectAttr(ConnectionHandle,
     Attribute, Value, BufferLength, &StringLength);

EXAMPLE USAGE (COBOL)
01 CONNECTIONHANDLE PICTURE S9(9) BINARY.
01 ATTRIBUTE PICTURE S9(9) BINARY.
01 VALUE PICTURE type.
01 BUFFERLENGTH PICTURE S9(9) BINARY.
01 STRINGLENGTH PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRGetConnectAttr" USING CONNECTIONHANDLE,
     ATTRIBUTE, VALUE, BUFFERLENGTH, STRINGLENGTH, RETURNCODE.

SEE ALSO
   AllocHandle(), SetConnectAttr()
NAME
GetConnectOption — Get the value of a connection attribute (deprecated)

SYNOPSIS

FUNCTION GetConnectOption
  (ConnectionHandle INTEGER, 
   Option SMALLINT, 
   Value ANY) 
RETURNS (SMALLINT)

DESCRIPTION

The application can call GetConnectOption() to obtain a value of an attribute for
ConnectionHandle. The function has the following parameters:

ConnectionHandle (input)
  Connection handle.

Option (input)
  The desired connection attribute. For a list of valid attributes, see Connection Attributes on
  page 29.

Value (output)
  The current value of the attribute specified by Option. Depending on the value of Option,
  this points either to a 32-bit integer or to a character string.

RETURN VALUE

[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or
[SQL_INVALID_HANDLE]. The [SQL_SUCCESS_WITH_INFO] outcome occurs if
implementation-defined warnings are produced.

DIAGNOSTICS

Option must be one of the defined attributes or an implementation-defined value ('HY092').

If Option is SQL_ATTR_AUTO_IPD, then a connection must be established on ConnectionHandle
('08003').

APPLICATION USAGE

The application can call GetConnectOption() at any time between the allocation and the freeing of
ConnectionHandle. It obtains the current value of the connection attribute.

EXAMPLE USAGE (C)

#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDBC ConnectionHandle;
SQLSMALLINT Option;
SQLPOINTER Value;
...
ReturnCode = SQLGetConnectOption(ConnectionHandle,
     Option, Value);

EXAMPLE USAGE (COBOL)

01 CONNECTIONHANDLE PICTURE S9(9) BINARY.
01 OPTION PICTURE S9(4) BINARY.
01 VALUE PICTURE type.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRGetConnectOption" USING CONNECTIONHANDLE,
     OPTION, VALUE, RETURNCODE.
SEE ALSO

AllocHandle(), GetConnectAttr(), SetConnectOption()
NAME
GetCursorName — Get the name of a cursor

SYNOPSIS

FUNCTION GetCursorName
(StatementHandle INTEGER,
 CursorName CHAR,
 BufferLength SMALLINT,
 NameLength SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION
The GetCursorName() function returns a cursor name associated with a statement handle. The function has the following parameters:

StatementHandle (input)
Statement handle.

CursorName (output)
Cursor name.

BufferLength (input)
Maximum number of octets to store in CursorName.

NameLength (output)
Length in octets of the output data (even if it does not entirely fit in CursorName).

Calling GetCursorName() sets CursorName to the cursor name associated with StatementHandle. The application can specify a cursor name explicitly by calling SetCursorName(); otherwise the cursor name is the one the implementation created (see Cursor Name in Section 4.5.1 on page 52). Implicit cursor names always start with 'SQLCUR' or 'SQL_CUR'.

RETURN VALUE
[SQL_SUCCESS], [SQL_SUCCESS_WITH_INFO], [SQL_INVALID_HANDLE] or [SQL_ERROR].

DIAGNOSTICS
The [SQL_SUCCESS_WITH_INFO] result occurs if the implementation truncated the cursor name ('01004') or if implementation-defined warnings are produced.

There must be a cursor name on StatementHandle ('HY015').

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLCHAR CursorName[SQL_MAX_ID_LENGTH+1];
SQLSMALLINT BufferLength;
SQLSMALLINT NameLength;
...
ReturnCode = SQLGetCursorName(StatementHandle,
   CursorName, BufferLength, &NameLength);
```

22. If the implementation implicitly generates a cursor name when the statement handle is allocated, this error never occurs.
EXAMPLE USAGE (COBOL)

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 CURSORNAME PICTURE X(18).
01 BUFFERLENGTH PICTURE S9(4) BINARY.
01 NAMELENGTH PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.

... 
CALL "SQLRGetCursorName" USING STATEMENTHANDLE, 
CURSORNAME, BUFFERLENGTH, NAMELENGTH, RETURNCODE.
```

SEE ALSO

`Prepare()`, `SetCursorName()`, `Execute()`, `ExecDirect()`
NAME
GetData — Retrieve one column of a row of the result set

SYNOPSIS
FUNCTION GetData
(StatementHandle INTEGER,
 ColumnNumber SMALLINT,
 TargetType SMALLINT,
 TargetValue ANY,
 BufferLength INTEGER,
 StrLen_or_Ind INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
After fetching a new row of a result set, the application can call GetData() to obtain result data for part or all of a single unbound column. The function has the following parameters:

StatementHandle (input)
Statement handle.

ColumnNumber (input)
Column number. The first column is number 1.

TargetType (input)
The data type of TargetValue in the application row buffer. This must be one of the following:

• The Integer Data Type Identifiers listed in Table 4-6 on page 59 or Table 4-7 on page 59.
• SQL_DEFAULT, to select a buffer type from Table 4-9 on page 61 based on the SQL data type of the source.
• SQL_ARC_TYPE, to use a buffer type that has been specified in the application row descriptor.

TargetValue (output)
Destination for the output data.

BufferLength (input)
Maximum number of octets to store in TargetValue, if the application row descriptor specifies a character-string data type; otherwise, ignored.

StrLen_or_Ind (output)
A variable whose value is interpreted as follows:

• If the output data is null, this is set to SQL_NULL_DATA.

• Otherwise, if the data type of the application buffer is SQL_CHAR, this is set to the length in octets of the output data (even if it does not entirely fit in TargetValue). This value excludes any data from this column that the application has already obtained from previous calls to GetData().

• Otherwise, the value is undefined.

In a programming language that supports pointers, the application can provide a null pointer to inhibit return of the above information.

This function retrieves data one column at a time. The application may also call GetData() to retrieve large data values in pieces.
Data conversion of column data may occur before assignment to the bound variables depending on the TYPE field of the corresponding descriptor record. If the TYPE field is SQL_DEFAULT, then the column data is converted to a type given by Table 4-9 on page 61 based on the data type of the column.

**RETURN VALUE**

[SQL_SUCCESS], [SQL_INVALID_HANDLE], [SQL_ERROR], [SQL_NO_DATA] or [SQL_SUCCESS_WITH_INFO].

The [SQL_NO_DATA] outcome occurs if the preceding fetch returned [SQL_NO_DATA]. In the case of [SQL_ERROR] and [SQL_NO_DATA], the resulting values of the output parameters are undefined.

**DIAGNOSTICS**

The [SQL_SUCCESS_WITH_INFO] outcome occurs if the column data is truncated (‘01004’). See **APPLICATION USAGE** below. [SQL_SUCCESS_WITH_INFO] can also result when implementation-defined warnings are produced.

The application must fetch a row before it calls GetData() to obtain columns (‘HY010’).

The value of ColumnNumber must conform to the following requirements (‘07009’):

- The value must be in the range from 1 to and including the number of columns in the result set.
- The value must be higher than the number of the highest bound column.*
- If the application called GetData() previously for a column of the same row:
  - The value must not specify a lower column number.*
  - The value must not specify the same column number if the previous call retrieved all the data for that column.*

If the application has specified an application row descriptor, then:

- Its COUNT field must be non-negative (‘07008’).
- ColumnNumber must not exceed COUNT (‘07009’) and the TYPE field in the descriptor record identified by ColumnNumber must be one of the Integer Data Type Identifier values listed in Section 4.8.2 on page 58, SQL_DEFAULT or an implementation-defined value (‘HY003’).
- The data type of each column must be assignment-compatible with the buffer type specified by the corresponding descriptor record (‘07006’). Retrieving a value may produce the data conversion diagnostics described in Section 5.4.10 on page 76.
- If the column value is null, then StrLen_or_Ind must not be a null pointer (‘22002’).
- If the descriptor record corresponding to ColumnNumber specifies a character-string data type, then BufferLength must be valid (see **String Termination** on page 19) (‘HY090’).

Retrieving a value from an arithmetic expression, and applying any specified conversion, must not result in a division by zero (‘22012’), either at the time of assignment or in computing an intermediate result.

* Implementations that support more permissive retrieval rules are not required to produce this error in this case.
APPLICATION USAGE
On a call that retrieves a character string and truncates it, the application can call `GetData()` again with the same `ColumnNumber`, to get subsequent data from the same unbound column, starting at the point of truncation. To obtain the entire column, the application repeats such calls until the function returns [SQL_SUCCESS].

Alternatively, the application may call `GetData()` with an incremented value of `ColumnNumber`, or fetch another row. This effectively discards the rest of the truncated column. The implementation also discards a truncated column if the application sets any field of the application row descriptor (even an ineffective change, such as changing between an explicit data type and a use of SQL_DEFAULT that effectively specifies the same data type).

EXAMPLE USAGE (C)
```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLSMALLINT ColumnNumber;
SQLSMALLINT TargetType;
SQLPOINTER TargetValue;
SQLINTEGER BufferLength;
SQLINTEGER StrLen_or_Ind;
...
ReturnCode = SQLGetData(StatementHandle, ColumnNumber,
    TargetType, TargetValue, BufferLength, &StrLen_or_Ind);
```

EXAMPLE USAGE (COBOL)
```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 COLUMNNUMBER   PICTURE S9(4) BINARY.
01 TARGETTYPE     PICTURE S9(4) BINARY.
01 TARGETVALUE    PICTURE type.
01 BUFFERLENGTH   PICTURE S9(9) BINARY.
01 STRLEN-OR-IND  PICTURE S9(9) BINARY.
01 RETURNCODE     PICTURE S9(4) BINARY.
...
CALL "SQLRGetData" USING STATEMENTHANDLE, COLUMNNUMBER,
    TARGETTYPE, TARGETVALUE, BUFFERLENGTH, STRLEN-OR-IND,
    RETURNCODE.
```

SEE ALSO
`Execute()`, `ExecDirect()`
NAME
GetDescField — Get a descriptor field (extensible)

SYNOPSIS
FUNCTION GetDescField
(DescriptorHandle INTEGER,
 RecNumber SMALLINT,
 FieldIdentifier SMALLINT,
 Value ANY,
 BufferLength INTEGER,
 StringLength INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The GetDescField() function has the following parameters:

DescriptorHandle (input)
Descriptor handle.

RecNumber (input)
The record number from which the specified field in the descriptor is to be retrieved. The first record is number 1.

FieldIdentifier (input)
The identifier of the field to be retrieved. This can be one of the following:

Header Fields
SQL_DESC_COUNT (type: SMALLINT)
Ignore RecNumber. Get the number of records in the descriptor.

SQL_DESC_ALLOC_TYPE (type: SMALLINT)
Ignore RecNumber. Store in Value either SQL_DESC_ALLOC_USER if the application explicitly allocated DescriptorHandle by calling AllocHandle(), or SQL_DESC_ALLOC_AUTO if the implementation automatically allocated DescriptorHandle for some statement handle. (The application cannot modify this field by calling SetDescField().)

Fields of a Descriptor Record
SQL_DESC_TYPE (type: SMALLINT)
Get the TYPE field of RecNumber. For application descriptors, Value is set to one of the Integer Data Type Identifier values in Section 4.8.2 on page 58. For implementation descriptors, Value is set to one of the Integer Data Type Identifier values in Table 4-4 on page 58.

SQL_DESC_LENGTH (type: INTEGER)
Get the LENGTH field of RecNumber.

SQL_DESC_OCTET_LENGTH (type: INTEGER)
Get the OCTET_LENGTH field of RecNumber.

SQL_DESC_PRECISION (type: SMALLINT)
Get the PRECISION field of RecNumber.

SQL_DESC_SCALE (type: SMALLINT)
Get the SCALE field of RecNumber.
GetDescField()  

**SQL_DESC_DATETIME_INTERVAL_CODE (type: SMALLINT)**

Get the DATETIME_INTERVAL_CODE field of RecNumber. Value is set to SQL_CODE_DATE, SQL_CODE_TIME or SQL_CODE_TIMESTAMP. See also Table 4-4 on page 58.

**SQL_DESC_NULLABLE (type: SMALLINT)**

Get the NULLABLE field of RecNumber. For implementation row descriptors, Value is set to SQL_NULLABLE if the column's definition allows null values, and to SQL_NO_NULLS otherwise. For implementation parameter descriptors, Value is always set to SQL_NULLABLE.

**SQL_DESC_NAME (type: CHAR(128))**

Get the NAME field of RecNumber.

**SQL_DESC_UNNAMED (type: SMALLINT)**

Get the UNNAMED field of RecNumber. For implementation row descriptors, Value is set to SQL_NAMED if the NAME field is an actual name, or SQL_UNNAMED if the NAME field is an implementation-generated column name that X/Open does not define (for example, where the subject column represents the UNION of two columns with different names). For implementation parameter descriptors, Value is always set to SQL_UNNAMED.

In addition, FieldIdentifier may be one of the following to specify deferred fields. In a language that supports pointers, GetDescField() returns a pointer to an application variable (or the null pointer if the application has not set the field), through which the application can obtain data, length or indicator information.

**SQL_DESC_DATA_PTR (type: SQLPOINTER)**

Get the DATA_PTR field of RecNumber.

**SQL_DESC_OCTET_LENGTH_PTR (type: SQLPOINTER)**

Get the OCTET_LENGTH_PTR field of RecNumber.

**SQL_DESC_INDICATOR_PTR (type: SQLPOINTER)**

Get the INDICATOR_PTR field of RecNumber.

**Value (output)**

The value for FieldIdentifier of RecNumber.

**BufferLength (input)**

Maximum number of octets to store in Value, if the field is a character string; otherwise, unused.

**StringLength (output)**

Length in octets of the output data (even if it does not entirely fit in Value), if the field is a character string; otherwise, unused.

The function returns in Value one field of RecNumber from the descriptor associated with DescriptorHandle. If FieldIdentifier does not denote a string, then the implementation ignores BufferLength and does not set StringLength.

**RETURN VALUE**

[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO], [SQL_INVALID_HANDLE] or [SQL_NO_DATA].

[SQL_NO_DATA] is returned if RecNumber is greater than the number of descriptor records.

**DIAGNOSTICS**

The [SQL_SUCCESS_WITH_INFO] outcome occurs if the returned data is truncated (’01004’) or
if implementation-defined warnings are produced.

FieldIdentifier must be one of the defined values or an implementation-defined value (’HY091’).

When FieldIdentifier indicates a header field, RecNumber must be zero; when FieldIdentifier indicates a field from a record, RecNumber must be 1 or greater (’07009’).

If DescriptorHandle is an implementation row descriptor, the associated statement handle must be in the prepared or executed state (’HY007’).

APPLICATION USAGE
The application can obtain any field of any record in the descriptor, in arbitrary order, by repeated calls to GetDescField().

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDESC DescriptorHandle;
SQLSMALLINT RecNumber;
SQLSMALLINT FieldIdentifier;
SQLPOINTER Value;
SQLINTEGER BufferLength;
SQLINTEGER StringLength;
...
ReturnCode = SQLGetDescField(DescriptorHandle, RecNumber,
FieldIdentifier, Value, BufferLength, &StringLength);

EXAMPLE USAGE (COBOL)
01 DESCRIPTORHANDLE PICTURE S9(9) BINARY.
01 RECNamber PICTURE S9(4) BINARY.
01 FIELDIDENTIFIER PICTURE S9(4) BINARY.
01 VALUE PICTURE type.
01 BUFFERLENGTH PICTURE S9(9) BINARY.
01 STRINGLENGTH PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRGetDescField" USING DESCRIPTORHANDLE, RECNamber,
FIELDIDENTIFIER, VALUE, BUFFERLENGTH, STRINGLENGTH, RETURNCODE.

SEE ALSO
GetDescRec()
NAME
GetDescRec — Get a descriptor record (concise)

SYNOPSIS
FUNCTION GetDescRec
(DescriptorHandle INTEGER,
  RecNumber SMALLINT,
  Name CHAR,
  BufferLength SMALLINT,
  NameLength SMALLINT,
  Type SMALLINT,
  SubType SMALLINT,
  Length INTEGER,
  Precision SMALLINT,
  Scale SMALLINT,
  Nullable SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION
The GetDescRec() function has the following parameters:

DescriptorHandle (input)
Descriptor handle.

RecNumber (input)
The record number from which the description in the descriptor is to be retrieved. The first
record is number 1.

Name (output)
The NAME field for the record.

BufferLength (input)
Maximum number of octets to store in Name.

NameLength (output)
Length in octets of the output data (even if it does not entirely fit in Name).

Type (output)
The TYPE field for the record. For application descriptors, this is one of the Integer Data
Type Identifier values in Section 4.8.2 on page 58. For implementation descriptors, this is
one of the Integer Data Type Identifier values in Table 4-4 on page 58.

SubType (output)
The DATETIME_INTERVAL_CODE field, for records whose TYPE is SQL_DATETIME.

Length (output)
The OCTET_LENGTH field for the record.

Precision (output)
The PRECISION field for the record.

Scale (output)
The SCALE field for the record.

Nullable (output)
The NULLABLE field for the record.

Calling GetDescRec() sets the output arguments to describe RecNumber of the descriptor
associated with DescriptorHandle.
**RETURN VALUE**

- [SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO], [SQL_INVALID_HANDLE]
or [SQL_NO_DATA].

[SQL_NO_DATA] is returned if `RecNumber` is greater than the number of descriptor records.

**DIAGNOSTICS**

The [SQL_SUCCESS_WITH_INFO] outcome occurs if the returned data is truncated ('01004') or if implementation-defined warnings are produced.

`RecNumber` must be 1 or greater ('07009').

If `DescriptorHandle` is an implementation row descriptor, the associated statement handle must be in the prepared or executed state ('HY007').

**APPLICATION USAGE**

In languages that allow the use of pointers, the application can provide a null pointer for any of `Name`, `Type`, `SubType`, `Length`, `Precision`, `Scale` or `Nullable`, in order to inhibit return of that information.

**EXAMPLE USAGE (C)**

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDESC DescriptorHandle;
SQLSMALLINT RecNumber;
SQLCHAR  *Name;
SQLSMALLINT BufferLength;
SQLSMALLINT NameLength;
SQLSMALLINT Type;
SQLSMALLINT SubType;
SQLINTEGER Length;
SQLSMALLINT Precision;
SQLSMALLINT Scale;
SQLSMALLINT Nullable;
...
ReturnCode = SQLGetDescRec(DescriptorHandle, RecNumber, Name, BufferLength, &NameLength, &Type, &SubType, &Length, &Precision, &Scale, &Nullable);
```

**EXAMPLE USAGE (COBOL)**

```cobol
01 DESCRIPTORHANDLE PICTURE S9(9) BINARY.
01 RECNUMBER PICTURE S9(4) BINARY.
01 NAME PICTURE X(128).
01 BUFFERLENGTH PICTURE S9(4) BINARY.
01 NAMELENGTH PICTURE S9(4) BINARY.
01 TYPE PICTURE S9(4) BINARY.
01 SUBTYPE PICTURE S9(4) BINARY.
01 LENGTH PICTURE S9(9) BINARY.
01 PRECISION PICTURE S9(4) BINARY.
01 SCALE PICTURE S9(4) BINARY.
01 NULLABLE PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRGetDescRec" USING DESCRIPTORHANDLE, RECNUMBER,
   NAME, BUFFERLENGTH, NAMELENGTH, TYPE, SUBTYPE, LENGTH,
   PRECISION, SCALE, NULLABLE, RETURNCODE.
```

Data Management: SQL Call Level Interface (CLI) 161
SEE ALSO

GetDescField()
NAME
GetDiagField — Return diagnostic information (extensible).

SYNOPSIS
FUNCTION GetDiagField
(HandleType SMALLINT,
 Handle INTEGER,
 RecNumber SMALLINT,
 DiagIdentifier SMALLINT,
 DiagInfo ANY,
 BufferLength SMALLINT,
 StringLength SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION
An application calls GetDiagField() to obtain a single piece of error or warning status from a
previous call to a CLI function, especially one that returned [SQL_ERROR] or
[SQL_SUCCESS_WITH_INFO]. (This is referred to below as the underlying function.)

The data structure associated with each handle contains a diagnostics area. The area consists of
header information and zero or more status records. Header information (which includes the
number of status records) pertains at large to the function’s execution, while each status record
describes a specific event in the execution.

The function has the following parameters:

HandleType (input)
A handle type identifier that describes the handle type of Handle. Valid values are the same
as for AllocHandle().

Handle (input)
A handle for which diagnostic information is desired.

RecNumber (input)
Indicates the status record from which the application seeks information. The first record is
number 1. If the value of DiagIdentifier indicates header information, then RecNumber must
be 0. If the value of DiagIdentifier indicates a field from a record, then RecNumber must be 1
or greater.

DiagIdentifier (input)
Indicates the desired piece of diagnostic information, and implicitly indicates whether the
information comes from the header or from record RecNumber. DiagIdentifier must contain
one of the following values:

Header Fields

SQL_DIAG_RETURNCODE (type: SMALLINT)
Return in DiagInfo the return code that the underlying function returned. This is one of
[SQL_SUCCESS], [SQL_SUCCESS_WITH_INFO], [SQL_NO_DATA] or [SQL_ERROR].
(It is never [SQL_INVALID_HANDLE].)

SQL_DIAG_NUMBER (type: INTEGER)
Return in DiagInfo the number of diagnostic records that are available for the specified
handle (see Number of Status Records on page 69).

SQL_DIAG_ROW_COUNT (type: INTEGER)
If HandleType is SQL_HANDLE_STMT, return in DiagInfo the row count (see Section
5.3.2 on page 72).
If Handle is not a statement handle (or in certain other cases described in Section 5.3.2 on page 72), the value of DiagInfo is undefined.

**EX**

**SQL_DIAG_DYNAMIC_FUNCTION (type: CHAR(254))**

If HandleType is SQL_HANDLE_STMT and the underlying function was ExecDirect() or Execute(), return in DiagInfo a string* describing the SQL statement that the underlying function executed, using values defined below; otherwise the value is a zero-length string.

<table>
<thead>
<tr>
<th>SQL Statement Executed</th>
<th>Value of SQL_DIAG_DYNAMIC_FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER TABLE</td>
<td>ALTER TABLE</td>
</tr>
<tr>
<td>CREATE INDEX</td>
<td>CREATE INDEX</td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>CREATE TABLE</td>
</tr>
<tr>
<td>CREATE VIEW</td>
<td>CREATE VIEW</td>
</tr>
<tr>
<td>cursor-specification</td>
<td>SELECT CURSOR</td>
</tr>
<tr>
<td>searched DELETE</td>
<td>DELETE WHERE</td>
</tr>
<tr>
<td>dynamic positioned DELETE</td>
<td>DYNAMIC DELETE CURSOR</td>
</tr>
<tr>
<td>DROP INDEX</td>
<td>DROP INDEX</td>
</tr>
<tr>
<td>DROP TABLE</td>
<td>DROP TABLE</td>
</tr>
<tr>
<td>DROP VIEW</td>
<td>DROP VIEW</td>
</tr>
<tr>
<td>GRANT</td>
<td>GRANT</td>
</tr>
<tr>
<td>INSERT</td>
<td>INSERT</td>
</tr>
<tr>
<td>REVOKE</td>
<td>REVOKE</td>
</tr>
<tr>
<td>searched UPDATE</td>
<td>UPDATE WHERE</td>
</tr>
<tr>
<td>dynamic positioned UPDATE</td>
<td>DYNAMIC UPDATE CURSOR</td>
</tr>
</tbody>
</table>

**EX**

**SQL_DIAG_DYNAMIC_FUNCTION_CODE (type: INTEGER)**

If HandleType is SQL_HANDLE_STMT and the underlying function was ExecDirect() or Execute(), return in DiagInfo a numeric code* describing the SQL statement that the underlying function executed (from the list of dynamic function codes in Appendix C); otherwise the value is SQL_UNKNOWN_STATEMENT.

**Fields of a Status Record**

**SQL_DIAG_SQLSTATE (type: CHAR(5))**

Return in DiagInfo a 5-character SQLSTATE code pertaining to diagnostic record RecNumber. The first 2 characters indicate class; the next 3 indicate subclass. The SQLSTATE code provides a portable diagnostic indication. See Appendix A for a list of X/Open-defined SQLSTATE codes.

**SQL_DIAG_NATIVE (type: INTEGER)**

Return in DiagInfo an implementation-defined error code pertaining to diagnostic record RecNumber. Portable applications should not base their behaviour on this value.

---

* This value must not reflect any SQL statements that a CLI implementation may have executed invisibly to the application. For example, this value never reports execution of a SET CONNECTION statement, which some CLI implementations may call from time to time based on the application's use of connection handles. The value never reports execution of a FETCH statement based on a CLI fetch operation. Metadata functions, such as Columns(), return a zero-length string.

If the above conditions are met but the relevant SQL statement was not recognised or was an implementation-defined SQL statement, the value returned is implementation-defined and can vary based on implementation-defined criteria.
SQL_DIAG_MESSAGE_TEXT (type: CHAR(254))
    Return in DiagInfo implementation-defined message text pertaining to diagnostic record RecNumber.

SQL_DIAG_CLASS_ORIGIN (type: CHAR(254))
    Return in DiagInfo a string that identifies the defining source of the class portion of the SQLSTATE code in RecNumber. Its value is 'ISO 9075' if the ISO SQL standard defines the class. Otherwise, its value is implementation-defined and must not be 'ISO 9075'.

SQL_DIAG_SUBCLASS_ORIGIN (type: CHAR(254))
    Return in DiagInfo a string with the same format and valid values as SQL_DESC_CLASS_ORIGIN above, that identifies the defining source of the subclass portion of the SQLSTATE code in RecNumber.

SQL_DIAG_CONNECTION_NAME (type: CHAR(128))
    Return in DiagInfo the connection name for the connection that the diagnostic record relates to. The connection name helps identify the connection in the case that there are multiple connections to the same server. In embedded SQL, the application can specify a connection name. In CLI, the application uses connection handles instead of names, and the value of this field is implementation-defined. However, for diagnostics associated with an environment handle, and for diagnostics that do not relate to any server, this field is a zero-length string.

SQL_DIAG_SERVER_NAME (type: CHAR(128))
    Return in DiagInfo the server name that the diagnostic record relates to, as it was supplied as the ServerName parameter in the call to Connect() that established the connection. The server name is also available as SQL_DATA_SOURCE_NAME by calling GetInfo().

For diagnostics that relate to the default server, this field is the actual server name. For diagnostics associated with an environment handle, and for diagnostics that do not relate to any server, this field is a zero-length string.

Additional values of DiagIdentifier may be defined in the future, or may be implementation-defined, to return other diagnostic information. The definition of these values specifies the type and meaning of the data returned in DiagInfo.

DiagInfo (output)
    Buffer for diagnostic information.

BufferLength (input)
    Length in octets of DiagInfo, if the requested diagnostic data is a character string; otherwise, unused.

StringLength (output)
    Length in octets of complete diagnostic information (even if it does not entirely fit in DiagInfo), if the requested diagnostic data is a character string; otherwise, unused.

If DiagInfo does not denote a string, then the implementation ignores BufferLength and does not set StringLength.

RETURN VALUE
    The function uses the following return values (instead of generating diagnostics) to report the outcome of its own execution:

[SQL_SUCCESS]
    The function successfully returned diagnostic information.
[SQL_INVALID_HANDLE]

HandleType and Handle, taken together, do not denote a valid handle.

[SQL_ERROR]

One of the following occurred:

- DiagIdentifier is not one of the values listed above.
- DiagIdentifier is SQL_DIAG_ROW_COUNT but Handle is not a statement handle on which an SQL statement has been executed.
- RecNumber is negative or 0 when DiagIdentifier indicates a field from a record; or RecNumber is non-zero when DiagIdentifier indicates header information.
- The value requested is a character string and BufferLength is invalid (see String Termination on page 19).

[SQL_NO_DATA]

RecNumber is greater than the number of diagnostic records that exist for Handle.\(^{23}\)

The function does not return [SQL_SUCCESS_WITH_INFO] even if it truncates a string output argument. No implementation-defined diagnostics are possible.

DIAGNOSTICS

None. This function never generates information in any handle’s diagnostics area to describe its own execution. This function is never the underlying function as defined in DESCRIPTION above. Any diagnostics are reported using the RETURN VALUE (see above).

APPLICATION USAGE

Production of status records by the underlying function is discussed in Section 5.2 on page 69.

GetDiagField() retrieves only the diagnostic information most recently associated with Handle. If an application calls another CLI function that writes diagnostic information to the same handle (even a success indication), any diagnostic information from previous calls is lost. For example, the sequence Execute(), GetCursorName(), GetDiagField(SQL_DIAG_ROW_COUNT) fails to obtain the row count because GetCursorName() has become the underlying function.

For the fields specified as CHAR(254), especially SQL_DIAG_MESSAGE_TEXT, some implementations may generate more than 254 characters of text. Truncation of this text is not reported. A portable application should set aside a buffer whose length in octets is \{SQL_MAX_MESSAGE_LENGTH\}. If, despite doing so, the application retrieves character-string data that entirely fills the buffer, it should call GetDiagField() again with a larger buffer to ensure retrieval of the complete text.

EXAMPLE USAGE (C)

```
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLSMALLINT HandleType;
SQLHENV Handle;        /* or SQLHDBC, SQLHSTMT or SQLHDESC */
SQLSMALLINT RecNumber;
SQLSMALLINT DiagIdentifier;
SQLPOINTER DiagInfo;
```

\(^{23}\) The error conditions described in the previous list items take precedence. The function also returns [SQL_NO_DATA] for any positive RecNumber if there are no diagnostic records for Handle. An application can scan all diagnostic records by looping, incrementing RecNumber, as long as the function returns [SQL_SUCCESS].
SQLSMALLINT BufferLength;
SQLSMALLINT StringLength;
...
/* Example uses environment handle */
HandleType = SQL_HANDLE_ENV;
ReturnCode = SQLGetDiagField(HandleType, Handle, RecNumber,
    DiagIdentifier, DiagInfo, BufferLength, &StringLength);

**EXAMPLE USAGE (COBOL)**

```cobol
01 HANDLETYPE PICTURE S9(4) BINARY.
01 HANDLE PICTURE S9(9) BINARY.
01 RECNBER PICTURE S9(4) BINARY.
01 DIAGIDENTIFIER PICTURE S9(4) BINARY.
01 DIAGINFO PICTURE type.
01 BUFFERLENGTH PICTURE S9(4) BINARY.
01 STRINGLENGTH PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
* EXAMPLE USES ENVIRONMENT HANDLE
MOVE SQL-HANDLE-ENV TO HANDLETYPE.
CALL "SQLRGetDiagField" USING HANDLETYPE, HANDLE, RECNBER,
    DIAGIDENTIFIER, DIAGINFO, BUFFERLENGTH, STRINGLENGTH, RETURNCODE.
```

**SEE ALSO**

* GetDiagRec()
NAME
   GetDiagRec — Return diagnostic information (concise).

SYNOPSIS
FUNCTION GetDiagRec
   (HandleType SMALLINT,
    Handle INTEGER,
    RecNumber SMALLINT,
    Sqlstate CHAR,
    NativeError INTEGER,
    MessageText CHAR,
    BufferLength SMALLINT,
    TextLength SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION
An application calls GetDiagRec() to obtain several commonly-used pieces of error or warning status from a previous call to a CLI function, especially one that returned [SQL_ERROR] or [SQL_SUCCESS_WITH_INFO]. (This is referred to below as the underlying function.)

The data structure associated with each handle contains a diagnostics area. The area consists of header information and zero or more status records. (Header information is available by calling GetDiagField().) Each status record includes values of Sqlstate, NativeError and MessageText/TextLength. Each status record describes a specific event in the execution of the underlying function.

The function has the following parameters:

HandleType (input)
   A handle type identifier that describes the handle type of Handle. Valid values are the same as for AllocHandle().

Handle (input)
   A handle for which diagnostic information is desired.

RecNumber (input)
   Indicates the status record from which the application seeks information. The first record is number 1.

Sqlstate (output)
   A 5-character SQLSTATE code pertaining to diagnostic record RecNumber. The first 2 characters indicate class; the next 3 indicate subclass. The SQLSTATE code provides a portable diagnostic indication. See Appendix A for a list of X/Open-defined SQLSTATE codes.

NativeError (output)
   An implementation-defined error code pertaining to diagnostic record RecNumber. Portable applications should not base their behaviour on this value.

MessageText (output)
   Implementation-defined message text pertaining to diagnostic record RecNumber.

BufferLength (input)
   Maximum number of octets to store in MessageText.

TextLength (output)
   Length in octets of the output data (even if it does not entirely fit in MessageText).
RETURN VALUE
The function uses the following return values (instead of generating diagnostics) to report the outcome of its own execution:

[SQL_SUCCESS]
The function successfully returned diagnostic information.

[SQL_INVALID_HANDLE]
HandleType and Handle, taken together, do not denote a valid handle.

[SQL_ERROR]
One of the following occurred:

- RecNumber is negative or 0.
- BufferLength is negative (or BufferLength is zero and the environment attribute SQL_ATTR_OUTPUT_NTS is true).

[SQL_NO_DATA]
RecNumber is greater than the number of diagnostic records that exist for Handle.\textsuperscript{24}

The function does not return [SQL_SUCCESS_WITH_INFO] even if it truncates a string output argument. No implementation-defined diagnostics are possible.

DIAGNOSTICS
None. This function never generates information in any handle’s diagnostics area to describe its own execution. This function is never the underlying function as defined in DESCRIPTION above. Any diagnostics are reported using the RETURN VALUE (see above).

APPLICATION USAGE
Production of status records by the underlying function is discussed in Section 5.2 on page 69.

In languages that allow the use of pointers, the application can provide a null pointer for any of Sqlstate, NativeError and MessageText, in order to inhibit return of that information.

GetDiagRec() retrieves only the diagnostic information most recently associated with Handle. If an application calls another CLI function that writes diagnostic information to the same handle (even a success indication), any diagnostic information from previous calls is lost.

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLSMALLINT HandleType;
SQLHENV Handle; /* or SQLHDBC, SQLHSTMT or SQLHDESC */
SQLSMALLINT RecNumber;
SQLCHAR Sqlstate[6];
SQLINTEGER NativeError;
SQLCHAR *MessageText;
SQLSMALLINT BufferLength;
SQLSMALLINT TextLength;
...
/* Example uses environment handle */

\textsuperscript{24} The error conditions described in the previous list items take precedence. The function also returns [SQL_NO_DATA] for any positive RecNumber if there are no diagnostic records for Handle. An application can scan all diagnostic records by looping, incrementing RecNumber, as long as the function returns [SQL_SUCCESS].
HandleType = SQL_HANDLE_ENV;
ReturnCode = SQLGetDiagRec(HandleType, Handle, RecNumber,
    Sqlstate, &NativeError, MessageText, BufferLength, &TextLength);

EXAMPLE USAGE (COBOL)
  01 HANDLETYPE  PICTURE S9(4) BINARY.
  01 HANDLE      PICTURE S9(9) BINARY.
  01 RECNUMBER   PICTURE S9(4) BINARY.
  01 SQLSTATE    PICTURE X(5).
  01 NATIVEERROR PICTURE S9(9) BINARY.
  01 MESSAGETEXT PICTURE X(512).
  01 BUFFERLENGTH PICTURE S9(4) BINARY.
  01 TEXTLENGTH  PICTURE S9(4) BINARY.
  01 RETURNCODE PICTURE S9(4) BINARY.

  ...  
  * EXAMPLE USES ENVIRONMENT HANDLE  
  MOVE SQL-HANDLE-ENV TO HANDLETYPE.
  CALL "SQLRGetDiagRec" USING HANDLETYPE, HANDLE, RECNUMBER, 
    SQLSTATE, NATIVEERROR, MESSAGETEXT, BUFFERLENGTH, 
    TEXTLENGTH, RETURNCODE.

SEE ALSO
  GetDiagField()
NAME
GetEnvAttr — Get the value of an environment attribute

SYNOPSIS
FUNCTION GetEnvAttr
  (EnvironmentHandle INTEGER,
   Attribute INTEGER,
   Value ANY,
   BufferLength INTEGER,
   StringLength INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The application can call GetEnvAttr() to obtain the value of an attribute for EnvironmentHandle.

The function has the following parameters:

EnvironmentHandle (input)
  Environment handle.

Attribute (input)
  The desired environment attribute. For a list of valid attributes, see Environment Attribute on page 28.

Value (output)
  The current value of the attribute specified by Attribute. The type of the value returned depends on Attribute.

BufferLength (input)
  Maximum number of octets to store in Value, if the attribute value is a character string; otherwise, unused.

StringLength (output)
  Length in octets of the output data (even if it does not entirely fit in Value), if the attribute value is a character string; otherwise, unused.

If Attribute does not denote a string, then the implementation ignores BufferLength and does not set StringLength.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

DIAGNOSTICS
The [SQL_SUCCESS_WITH_INFO] outcome occurs if the returned data is truncated ('01004') or if implementation-defined warnings are produced.

Attribute must be one of the defined attributes or an implementation-defined value ('HY092').

APPLICATION USAGE
GetEnvAttr() can be called at any time between the allocation and the freeing of EnvironmentHandle. It obtains the current value of the environment attribute. The manual page for SetEnvAttr() specifies the environment attributes and their effects. If the application has called SetEnvAttr() since calling a CLI function affected by the attribute, then the value GetEnvAttr() returns might not be the value that affected that CLI function.

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
GetEnvAttr()

SQLHENV EnvironmentHandle;
SQLINTEGER Attribute;
SQLPOINTER Value;
SQLINTEGER BufferLength;
SQLINTEGER StringLength;
...

ReturnCode = SQLGetEnvAttr(EnvironmentHandle,
   Attribute, Value, BufferLength, &StringLength);

EXAMPLE USAGE (COBOL)

01 ENVIRONMENTHANDLE PICTURE S9(9) BINARY.
01 ATTRIBUTE PICTURE S9(9) BINARY.
01 VALUE PICTURE type.
01 BUFFERLENGTH PICTURE S9(9) BINARY.
01 STRINGLENGTH PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...

CALL "SQLRGetEnvAttr" USING ENVIRONMENTHANDLE,
   ATTRIBUTE, VALUE, BUFFERLENGTH, STRINGLENGTH, RETURNCODE.

SEE ALSO

AllocHandle(), SetEnvAttr()
NAME
GetFunctions — Determine whether a function is supported

SYNOPSIS
FUNCTION GetFunctions
    (ConnectionHandle INTEGER,
     FunctionId SMALLINT,
     Supported SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION
The application can call GetFunctions() to find out whether or not the implementation supports a
specified CLI function.

The function has the following parameters:

ConnectionHandle (input)
    Connection handle.

FunctionId (input)
    A number that represents the CLI function in question. The following values represent the
    functions described in this document:

    SQL_API_SQLALLOCCONNECT
    SQL_API_SQLALLOCENV
    SQL_API_SQLALLOCHANDLE
    SQL_API_SQLALLOCSTMT
    SQL_API_SQLBINDCOL
    SQL_API_SQLBINDPARAM
    SQL_API_SQLCANCEL
    SQL_API_SQLCLOSECURSOR
    SQL_API_SQLCOLATTRIBUTE
    SQL_API_SQLCOLUMNS
    SQL_API_SQLCONNECT
    SQL_API_SQLCOPYDESC
    SQL_API_SQLDATASOURCES
    SQL_API_SQLDESCRIBECOL
    SQL_API_SQLDISCONNECT
    SQL_API_SQLENDTRAN
    SQL_API_SQLERROR
    SQL_API_SQLEXECDIRECT
    SQL_API_SQLEXECUTE
    SQL_API_SQLFETCH
    SQL_API_SQLFETCHSCROLL
    SQL_API_SQLFREECONNECT
    SQL_API_SQLFREEENV
    SQL_API_SQLFREEHANDLE
    SQL_API_SQLFREESTMT
    SQL_API_SQLGETCONNECTATTR
    SQL_API_SQLGETCONNECTOPTION
    SQL_API_SQLGETCURSORNAME
    SQL_API_SQLGETDATA
    SQL_API_SQLGETDESCFIELD
    SQL_API_SQLGETDESCREC
    SQL_API_SQLGETDIAGFIELD
GetFunctions()

Supported (output)
The value SQL_TRUE if the CLI implementation supports FunctionId; otherwise, the value SQL_FALSE.

RETURN VALUE
[SQL_SUCCESS], [SQL_SUCCESS_WITH_INFO], or [SQL_INVALID_HANDLE]. [SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

APPLICATION USAGE
The application can call GetFunctions() at any time that ConnectionHandle is in the connected state. It is called for a single CLI function at a time and obtains information about whether or not that CLI function is supported.

If GetFunctions() returns SQL_TRUE in the Supported argument, it asserts that both the CLI client and server components active on ConnectionHandle are equipped to support a valid call to the specified function.

COMPLIANCE POLICY
X/Open includes GetFunctions() in this document to let applications adapt to partial or extended CLI implementations. However, its presence does not affect the compliance requirements for CLI functions. X/Open-compliant CLI implementations must return SQL_TRUE when queried regarding any function in this document not marked as optional.

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDBC ConnectionHandle;
SQLSMALLINT FunctionId;
SQLSMALLINT Supported;
...

ReturnCode = SQLGetFunctions(ConnectionHandle,
FunctionId, &Supported);

**EXAMPLE USAGE (COBOL)**

```
01 CONNECTIONHANDLE PICTURE S9(9) BINARY.
01 FUNCTIONID PICTURE S9(4) BINARY.
01 SUPPORTED PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRGetFunctions" USING CONNECTIONHANDLE,
     FUNCTIONID, SUPPORTED, RETURNCODE.
```
NAME

GetInfo — Get information about the CLI implementation and the currently-connected server

SYNOPSIS

```ex
FUNCTION GetInfo
(ConnectionHandle INTEGER,
InfoType SMALLINT,
InfoValue ANY,
BufferLength SMALLINT,
StringLength SMALLINT)
RETURNS (SMALLINT)
```

DESCRIPTION

The application can call GetInfo() to find information about the support of various characteristics provided by a currently-connected server. The application specifies a value that identifies a specific characteristic and receives information about the support of that characteristic.

The function has the following parameters:

- **ConnectionHandle** (input)
  Connection handle.

- **InfoType** (input)
  The type of information being requested. It must be one of the following values or another value that the implementation recognises:

  - **SQL_ACCESSIBLE_TABLES** (type: CHAR(1))
    This is ‘Y’ if the user is guaranteed SELECT privileges to all tables that `SQLTables()` returns, and ‘N’ otherwise.

  - **SQL_ALTER_TABLE** (type: INTEGER)
    This indicates the clauses of ALTER TABLE that the server supports. The value returned is a 32-bit bitmask with the low-order bits identified as follows:
    - **SQL_AT_ADD_COLUMN**
    - **SQL_AT_DROP_COLUMN**
    - **SQL_AT_ALTER_COLUMN**
    - **SQL_AT_ADD_CONSTRAINT**
    - **SQL_AT_DROP_CONSTRAINT**

  - **SQL_CATALOG_NAME** (type: CHAR(1))
    This is either ‘Y’ if the server supports catalog names, or ‘N’ if it does not. Object naming is discussed in Section 2.4.3 on page 22.

  - **SQL_COLLATION_SEQ** (type: CHAR(254))
    The assumed ordering of the character set for this server. Examples are ‘ISO 8859-1’ and ‘EBCDIC’.

  - **SQL_CURSOR_COMMIT_BEHAVIOR** (type: SMALLINT)
    This indicates the effect on cursors and prepared statements of transaction commitment. It contains one of the following values:
    - **SQL_CB_DELETE**
      Commitment closes cursors and deletes prepared statements.
    - **SQL_CB_CLOSE**
      Commitment destroys cursors, and retains prepared statements.
SQL_CB_PRESERVE
   Commitment retains cursors and prepared statements.

SQL_CURSORSENSITIVITY: (type: INTEGER)
   This indicates the support for cursor sensitivity. Its value is one of the following:
   SQL_INSENSITIVE
      Cursor sensitivity is unsupported
   SQL_UNSPECIFIED
      Support for cursor sensitivity is unspecified
   SQL_SENSITIVE
      Cursor sensitivity is supported.

SQL_DATASOURCENAME (type: CHAR(128))
   This is the name used as the ServerName argument to the Connect() call that established
   the connection to ConnectionHandle. An application routine might require this
   information when a separate routine has made the connection.

SQL_DATASOURCE_READONLY (type: CHAR(1))
   This is ‘Y’ if data from the connected data source can only be read, or ‘N’ if data can
   both be read and updated.

SQL_DBMS_NAME (type: CHAR(254))
   This is the name of the server product in use on ConnectionHandle.

SQL_DBMS_VER (type: CHAR(254))
   This is the version number of the server product in use on ConnectionHandle. The
   format is:

   mm.nn.rrrr [opt]

   where mm is the major version, nn is the minor version, rrrr is the release version (these
   fields are exclusively numeric), and opt represents an implementation-specific release
   identification.

SQL_DEFAULT_TXN_ISOLATION (type: INTEGER)
   This is the server’s default isolation level. The following values indicate the four levels
   described in the X/Open SQL specification:

   SQL_TXN_READ_UNCOMMITTED
   SQL_TXN_READ_COMMITTED
   SQL_TXN_REPEATABLEREAD
   SQL_TXN_SERIALIZABLE

SQL_DESCRIBE_PARAMETER (type: CHAR(1))
   This indicates the implementation’s support for a capability analogous to the
   DESCRIBE INPUT statement of the X/Open SQL specification. Its value is either ‘Y’ if
   dynamic parameters can be described, or ‘N’ otherwise.

SQL_FETCH_DIRECTION (type: INTEGER)
   This indicates the type of cursor movement the implementation supports. The value is
   a 32-bit bitmask with the low-order bits identified as follows:
SQL_FD_ABSOLUTE
SQL_FD_FIRST
SQL_FD_LAST
SQL_FD_NEXT
SQL_FD_PRIOR
SQL_FD_RELATIVE

SQL_GETDATA_EXTENSIONS (type: INTEGER)
This indicates whether the implementation supports certain extensions to GetData().
This is a 32-bit bitmask.

SQL_GD_ANY_COLUMN
GetData() can be called for to obtain columns that precede the last bound column.

SQL_GD_ANY_ORDER
GetData() can be called for columns in any order.

SQL_IDENTIFIER_CASE (type: SMALLINT)
An indication of what the server does when placing a non-delimited identifier into the
catalog. The values are:

SQL_IC_UPPER
Store in upper case, as X/Open specifies

SQL_IC_LOWER
Store in lower case

SQL_IC_SENSITIVE
Store in mixed case

SQL_IC_MIXED
Store in mixed case but treat as CASE INSENSITIVE when used.

SQL_IDENTIFIER_QUOTE_CHAR (type: CHAR(1))
An indication of the character that delimits a delimited identifier. It is blank if the
implementation does not support delimited identifiers. X/Open-compliant
implementations return the quote (" ) character.

SQL_INTEGRITY (type: CHAR(1))
This is ‘Y’ if the server supports the Integrity Enhancement Feature (IEF) of the X/Open
SQL specification, or ‘N’ otherwise.

SQL_MAX_CATALOG_NAME_LEN (type: SMALLINT)
This is the maximum size in characters that the server supports for a catalog name, or
zero if this is unknown.

SQL_MAX_COLUMN_NAME_LEN (type: SMALLINT)
This is the maximum length in characters of a column name.

SQL_MAX_COLUMNS_IN_GROUP_BY (type: SMALLINT)
This is the maximum number of columns that the server supports in a GROUP BY
clause, or zero if there is no limit.

SQL_MAX_COLUMNS_IN_INDEX (type: SMALLINT)
This is the maximum number of columns that the server supports in an index, or zero if
there is no limit.

SQL_MAX_COLUMNS_IN_ORDER_BY (type: SMALLINT)
This is the maximum number of columns that the server supports in an ORDER BY
clause, or zero if there is no limit.
SQL_MAX_COLUMNS_IN_SELECT (type: SMALLINT)
This is the maximum number of columns that the server supports in a select list, or zero if there is no limit.

SQL_MAX_COLUMNS_IN_TABLE (type: SMALLINT)
This is the maximum number of columns that the server supports in a base table, or zero if there is no limit.

SQL_MAX_CONCURRENT_ACTIVITIES (type: SMALLINT)
This is the maximum number of statement handles on ConnectionHandle that can either be in a need-data state or have pending results, or 0 if this is unknown or not a constant.

SQL_MAX_CURSOR_NAME_LEN (type: SMALLINT)
This is the maximum length in characters of a cursor name.

SQL_MAX_DRIVER_CONNECTIONS (type: SMALLINT)
This is the maximum number of active connections of the same type as ConnectionHandle that the application can obtain, or 0 if this is unknown or is not a constant.

A response to this request type may obtain information on limitations of the server or of any internal mechanism used to gain access to the server. The response reflects the most restrictive of these limits that applies. This limit does not constitute a guarantee that that number of calls to Connect() will succeed, as either type of limit may vary dynamically based on availability of resources or on other connections the application may make.

SQL_MAX_IDENTIFIER_LEN (type: SMALLINT)
The maximum size in characters that the server supports for user-defined names.

SQL_MAX_INDEX_SIZE (type: SMALLINT)
This is the maximum size that the server supports for the combined columns in an index, or zero if there is no limit. The units are the standard storage units described in the X/Open SQL specification.

SQL_MAX_ROW_SIZE (type: INTEGER)
This is the maximum length that the server supports in single row of a base table, or zero if there is no limit. The units are the standard storage units described in the X/Open SQL specification.

SQL_MAX_SCHEMA_NAME_LEN (type: SMALLINT)
This is the maximum size in characters that the server supports for a schema name, or zero if this is unknown.

SQL_MAX_STATEMENT_LEN (type: INTEGER)
This is the maximum length in characters of an SQL statement string, or zero if this is unknown.

SQL_MAX_TABLE_NAME_LEN (type: SMALLINT)
This is the maximum length in characters of an unqualified table name.

SQL_MAX_TABLES_IN_SELECT (type: SMALLINT)
This is the maximum number of table names allowed in a FROM clause in a query specification.

SQL_MAX_USER_NAME_LEN (type: SMALLINT)
This is the maximum size in characters allowed for a user identifier, or zero if this is unknown.
SQL_NULL_COLLATION (type: SMALLINT)
This is SQL_NC_HIGH if null values sort high, or SQL_NC_LOW if null values sort low.

SQL_OJ_CAPABILITIES (type: CHAR(1))
This is zero, one or more of the following values:

SQL_OJ_LEFT
The implementation supports LEFT OUTER JOIN.

SQL_OJ_RIGHT
The implementation supports RIGHT OUTER JOIN.

SQL_OJ_FULL
The implementation supports FULL OUTER JOIN.

SQL_OJ_NESTED
The implementation supports nested OUTER JOINs.

SQL_OJ_NOT_ORDERED
The order of the tables underlying the columns in the ON clause need not be in the same order as tables in the JOIN clause. This bit must be set for all X/Open-compliant implementations that support OUTER JOIN.

SQL_OJ_INNER
The inner table of an OUTER JOIN can also be an inner join. This bit must be set for all X/Open-compliant implementations that support OUTER JOIN.

SQL_OJ_ALL_COMPARISON_OPS
Any predicate may be used in the ON clause. This bit must be set for all X/Open-compliant implementations that support OUTER JOIN. If this bit is not set, the = (equality) operator is the only valid comparison operator in the ON clause.

SQL_ORDER_BY_COLUMNS_IN_SELECT (type: CHAR(1))
This is ‘Y’ if columns in ORDER BY clauses must be in the select list, as specified in the X/Open SQL specification, or ‘N’ otherwise.

SQL_SCROLL_CONCURRENCY (type: INTEGER)
This indicates the concurrency control capabilities that the implementation supports for scrollable cursors. The value is a 32-bit bitmask with the low-order bits identified as follows:

SQL_SCCO_READ_ONLY
The cursor can be read, but no updates are allowed.

SQL_SCCO_LOCK
The cursor can use the lowest level of locking that ensures that the row can be updated.

SQL_SCCO_OPT_ROWVER
The cursor can use optimistic concurrency with row identifiers or timestamps.

SQL_SCCO_OPT_VALUES
The cursor can use optimistic concurrency comparing values.

SQL_SEARCH_PATTERN_ESCAPE (type: CHAR(1))
This is the character to be used as the escape character in the search patterns of the metadata functions, or an empty string if such an escape character is not supported.
SQL_SERVER_NAME (type: CHAR(128))
This is the actual name of the server that is being accessed via the connection. This can
be different than the value that was specified for ServerName in Connect().

SQL_SPECIAL_CHARACTERS (type: CHAR(254))
This is a character string that contains all the characters that the server allows in
identifiers other than non-delimited identifiers. Upper-case letters, lower-case letters,
numerals, and '_' are not included in this string and are always permitted in such
identifiers.

SQL_TXN_CAPABLE (type: SMALLINT) This indicates any transaction restrictions the
server imposes. The value is one of the following:

SQL_TC_ALL
Transactions can contain both DML and DDL without restriction.

SQL_TC_DDL_COMMIT
Transactions can contain only DML. Execution of a DDL statement during a
transaction causes the completion of the transaction before the DDL statement.

SQL_TC_DDL_IGNORE
Transactions can contain only DML. Any DDL statement is ignored.

SQL_TC_DML
Transactions can contain only DML. Any DDL in a transaction results in an error.

SQL_TC_NONE
Transactions are not supported at all. Each statement is self-committing.

An X/Open-compliant implementation returns either SQL_TC_ALL,
SQL_TC_DDL_COMMIT or SQL_TC_DML.

SQL_TXN_ISOLATION_OPTION (type: INTEGER)
This indicates the isolation levels the server supports. The value is a 32-bit bitmask
with the low-order bits identified as follows:

SQL_TXN_READ_UNCOMMITTED
SQL_TXN_READ_COMMITTED
SQL_TXN_REPEATABLE_READ
SQL_TXN_SERIALIZABLE

These four transaction isolation levels are defined in the X/Open SQL specification.

SQL_USER_NAME (type CHAR(128))
This is the current user name. It is the same value as the USER pseudo-literal, but can
be obtained without executing a cursor-specification.

SQL_XOPEN_CLI_YEAR
This is the year of publication of the X/Open specification with which the CLI
implementation fully complies. To indicate compliance with this document, the value
is '1995'. If the implementation does not fully comply with any published X/Open
specification, the value is an empty string.

InfoValue (output)
The current value of the characteristic specified by InfoType. The type of the value returned
depends on InfoType.

BufferLength (input)
The maximum number of octets to store in InfoValue, if Value is a character string; otherwise,
unused.
GetInfo()

StringLength (output)
The length in octets of the output data (even if it does not entirely fit into InfoValue) if InfoValue is a character string; otherwise, unused.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO], or [SQL_INVALID_HANDLE].

DIAGNOSTICS
The [SQL_SUCCESS_WITH_INFO] outcome occurs if the returned data is truncated ('01004') or if implementation-defined warnings are produced.

InfoType argument must be one of the defined values or an implementation-defined value ('HY096').

APPLICATION USAGE
The application can call GetInfo() at any time that ConnectionHandle is in the connected state. It is called for a single characteristic at a time and obtains information about the support of that characteristic on the server.

Many types of information returned by GetInfo() describe implementation-defined characteristics. Applications that require particular values inhibit their own portability.

COMPLIANCE POLICY
X/Open includes GetInfo() in this document to let applications adapt to partial or extended CLI implementations. However, its presence does not affect the CLI compliance requirements. X/Open constrains the values that compliant implementations can report for some of the GetInfo() characteristics. See the X/Open SQL specification.

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDBC ConnectionHandle;
SQLSMALLINT InfoType;
SQLPOINTER InfoValue;
SQLSMALLINT BufferLength;
SQLSMALLINT StringLength;
...
ReturnCode = SQLGetInfo(ConnectionHandle, InfoType, InfoValue, BufferLength, &StringLength);

EXAMPLE USAGE (COBOL)
01 CONNECTIONHANDLE PICTURE S9(9) BINARY.
01 INFOTYPE PICTURE S9(4) BINARY.
01 INFOVALUE PICTURE type.
01 BUFFERLENGTH PICTURE S9(4) BINARY.
01 STRINGLENGTH PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRGetInfo" USING CONNECTIONHANDLE,
   INFOTYPE, INFOVALUE, BUFFERLENGTH, STRINGLENGTH, RETURNCODE.
NAME

GetStmtAttr — Get the value of a statement attribute

SYNOPSIS

FUNCTION GetStmtAttr
(StatementHandle INTEGER,
 Attribute INTEGER,
 Value ANY,
 BufferLength INTEGER,
 StringLength INTEGER)
RETURNS (SMALLINT)

DESCRIPTION

The application can call GetStmtAttr() to obtain the value of an attribute for StatementHandle.

The function has the following parameters:

StatementHandle (input)
Statement handle.

Attribute (input)
The desired statement attribute. For a list of valid attributes, see Statement Attributes on page 30.

Value (output)
The current value of the attribute specified by Attribute. The type of the value returned depends on Attribute.

BufferLength (input)
Maximum number of octets to store in Value, if the attribute value is a character string; otherwise, unused.

StringLength (output)
Length in octets of the output data (even if it does not entirely fit in Value), if the attribute value is a character string; otherwise, unused.

If Attribute does not denote a string, then the implementation ignores BufferLength and does not set StringLength.

RETURN VALUE

[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

DIAGNOSTICS

The [SQL_SUCCESS_WITH_INFO] outcome occurs if the returned data is truncated ('01004') or if implementation-defined warnings are produced.

Attribute must be one of the defined attributes or an implementation-defined value ('HY092'). (If Attribute is an attribute defined in this document but marked optional, and if the implementation does not support that option, the diagnostic is 'HYC00'.)

APPLICATION USAGE

GetStmtAttr() can be called at any time between the allocation and the freeing of StatementHandle. It obtains the current value of the statement attribute. The manual page for SetStmtAttr() specifies the point in time at which statement attributes affect processing of StatementHandle; if SetStmtAttr() has been called since this time, the attribute value that a subsequent GetStmtAttr() returns might not be the value that affected the processing of the statement text associated with StatementHandle.
EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLINTEGER Attribute;
SQLPOINTER Value;
SQLINTEGER BufferLength;
SQLINTEGER StringLength;
...
ReturnCode = SQLGetStmtAttr(StatementHandle,
   Attribute, Value, BufferLength, &StringLength);

EXAMPLE USAGE (COBOL)
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 ATTRIBUTE PICTURE S9(9) BINARY.
01 VALUE PICTURE type.
01 BUFFERLENGTH PICTURE S9(9) BINARY.
01 STRINGLENGTH PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRGetStmtAttr" USING STATEMENTHANDLE,
   ATTRIBUTE, VALUE, BUFFERLENGTH, STRINGLENGTH, RETURNCODE.

SEE ALSO
SetStmtAttr()
NAME
GetStmtOption — Get the value of a statement attribute (deprecated)

SYNOPSIS

FUNCTION GetStmtOption
  (StatementHandle INTEGER,
   Option SMALLINT,
   Value ANY)
RETURNS (SMALLINT)

DESCRIPTION
The application can call GetStmtOption() to obtain a value of an attribute for StatementHandle.

The function has the following parameters:

StatementHandle (input)
Statement handle.

Option (input)
The desired statement attribute. For a list of valid attributes, see Statement Attributes on page 30.

Value (output)
The current value of the attribute specified by Option. Depending on the value of Option, this points either to a 32-bit integer value or to a character string.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO], or [SQL_INVALID_HANDLE]. The [SQL_SUCCESS_WITH_INFO] outcome occurs if implementation-defined warnings are produced.

DIAGNOSTICS
Option must be one of the defined attributes or an implementation-defined value (‘HY092’). (If Option is an attribute defined in this document but marked optional, and if the implementation does not support that option, the diagnostic is ‘HYC00’.)

APPLICATION USAGE
The application can call GetStmtOption() at any time between the allocation and the freeing of StatementHandle. It obtains the current value of the statement attribute. The manual page for SetStmtAttr() specifies the point in time at which statement attributes affect processing of StatementHandle; if SetStmtAttr() or SetStmtOption() has been called since this time, the attribute value that a subsequent SetStmtOption() returns might not be the value that affected processing of the statement text associated with StatementHandle.

EXAMPLE USAGE(C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLSMALLINT Option;
SQLPOINTER Value;
...
ReturnCode = SQLGetStmtOption(StatementHandle, Option Value);

EXAMPLE USAGE (COBOL)
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 OPTION PICTURE S9(4) BINARY.
01 VALUE PICTURE type.
GetStmtOption()

01 RETURNCODE    PICTURE S9(4) BINARY.

...  
CALL "SQLRGetStmtOption" USING STATEMENTHANDLE,
      OPTION, VALUE, RETURNCODE.

SEE ALSO
  AllocHandle(), GetStmtAttr(), SetStmtOption()
NAME
GetTypeInfo — Get information on server data types

SYNOPSIS
FUNCTION GetTypeInfo
(StatementHandle INTEGER,
 DataTypes SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION
The application can call GetTypeInfo() to find what data types are supported by the server represented by the connection on which StatementHandle was allocated. The application specifies a single data type or requests information on all data types.

The function has the following parameters:

StatementHandle (input)
Statement handle.

DataType (input)
The SQL data type being queried. It must be one of the following values or a value that the implementation supports:

- SQL_ALL_TYPES
- SQL_CHAR
- SQL_DECIMAL
- SQL_DOUBLE
- SQL_FLOAT
- SQL_INTEGER
- SQL_NUMERIC
- SQL_REAL
- SQL_SMALLINT
- SQL_TYPE_DATE
- SQL_TYPE_TIME
- SQL_TYPE_TIMESTAMP
- SQL_VARCHAR

Data type information is returned in the form of a result set on StatementHandle, where each data type is represented by at least one row of the result set.

If there is more than one name by which the data type can be specified (as data-type in the CREATE TABLE or ALTER TABLE statement), then there may be one row for each name in the result set. For example, for SQL_VARCHAR, there may be a row for VARCHAR and one for CHARACTER VARYING.

The result set has at least these columns in the following order:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_NAME</td>
<td>VARCHAR(128) NOT NULL</td>
<td>A name for the SQL data type identified by DataType.</td>
</tr>
<tr>
<td>DATA_TYPE</td>
<td>SMALLINT NOT NULL</td>
<td>The value of data type as it should be specified for the DataType parameter. This is the actual DataType argument unless the argument is SQL_ALLTYPES. See also the SQL_DATA_TYPE column.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Type</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------</td>
<td>------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PRECISION</td>
<td>INTEGER</td>
<td>The maximum precision/length that the server supports for this data type: For numeric data, this is the maximum precision. For string data, this is the length in characters. For date/time data types, this is the length in characters of the string representation (assuming the maximum allowed precision of the fractional seconds component). For some implementation-defined data types, this column may be null.</td>
</tr>
<tr>
<td>LITERAL_PREFIX</td>
<td>VARCHAR(128)</td>
<td>Character or characters that the implementation recognises as a prefix for a literal of this data type for the server. This column is null for data types where a literal prefix is not applicable.</td>
</tr>
<tr>
<td>LITERAL_SUFFIX</td>
<td>VARCHAR(128)</td>
<td>Character or characters that the implementation recognises as a suffix for a literal of this data type for the server. This column is null for data types where a literal suffix is not applicable.</td>
</tr>
<tr>
<td>CREATE_PARAMS</td>
<td>VARCHAR(128)</td>
<td>The text of this column contains a list of keywords, separated by commas, corresponding to each parameter the application may specify in parentheses when using the name in TYPE_NAME as a data-type in X/Open SQL. The keywords in the list can be any of the following: LENGTH, PRECISION, SCALE. They appear in the order that the syntax requires that they be used. If the application would never follow the data type by parenthesised parameters, then this column is null.</td>
</tr>
<tr>
<td>NULLABLE</td>
<td>SMALLINT</td>
<td>This column contains the value SQL_NULLABLE if the data type accepts null values; it contains SQL_NO_NULLS if the data type cannot accept null values; and it contains SQL_NULLABLE_UNKNOWN if it is not known whether the data type accepts null values. For all the data types defined in this document, the value is SQL_NULLABLE.</td>
</tr>
<tr>
<td>CASE_SENSITIVE</td>
<td>SMALLINT</td>
<td>This column contains the value SQL_TRUE if the data type can be treated as case-sensitive for collation purposes; otherwise, it contains SQL_FALSE.</td>
</tr>
<tr>
<td>SEARCHABLE</td>
<td>SMALLINT</td>
<td>Specifies the types of predicate where the data type can be used. Its value is a bitwise-inclusive OR of zero or more of the following: SQL_PRED_CHAR The data type can be used in the LIKE predicate SQL_PRED_BASIC The data type can be used in the comparison, quantified comparison, BETWEEN, DISTINCT, IN, MATCH and UNIQUE predicates.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Type</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UNSIGNED_ATTRIBUTE</td>
<td>SMALLINT</td>
<td>If none of the above bits are set, this column’s value is SQL_PRED_NONE. This column contains the value SQL_TRUE if the data type is unsigned; otherwise, it contains the value SQL_FALSE. If sign is not applicable (such as for non-numeric data types), this column is null.</td>
</tr>
<tr>
<td>FIXED_PREC_SCALE</td>
<td>SMALLINT</td>
<td>This column contains the value SQL_TRUE if the data type is exact numeric and has a fixed precision and scale; otherwise, it contains the value SQL_FALSE.</td>
</tr>
<tr>
<td>AUTO_INCREMENT</td>
<td>SMALLINT</td>
<td>This column contains SQL_TRUE if a column of this data type is automatically set to a unique value when a row is inserted.* Otherwise, this column contains SQL_FALSE.</td>
</tr>
<tr>
<td>LOCAL_TYPE_NAME</td>
<td>VARCHAR(128)</td>
<td>This column contains any localised (native language) name for the data type that is different from the regular name of the data type. If there is no localised name, this column is null. This column is intended for display only. The character set of the string is locale-dependent and is typically the default character set of the server.</td>
</tr>
<tr>
<td>MINIMUM_SCALE</td>
<td>INTEGER</td>
<td>The minimum scale of the data type. For data types with a fixed scale, MINIMUM_SCALE and MAXIMUM_SCALE are the same. For date/time data types, this column’s value is 0. If sign is not applicable (such as for non-numeric data types), this column is null.</td>
</tr>
<tr>
<td>MAXIMUM_SCALE</td>
<td>INTEGER</td>
<td>The maximum scale of the data type. For data types with a fixed scale, MINIMUM_SCALE and MAXIMUM_SCALE are the same. For the date/time data types TIME and TIMESTAMP, this is the maximum value of the precision of the fractional seconds component. If sign is not applicable (such as for non-numeric data types), this column is null.</td>
</tr>
<tr>
<td>SQL_DATA_TYPE</td>
<td>SMALLINT</td>
<td>The value of the data type as it appears in the TYPE field in the descriptor. This column is the same as DATA_TYPE except for the date/time data types.</td>
</tr>
<tr>
<td>SQL_DATETIME_SUB</td>
<td>SMALLINT</td>
<td>When the value in SQL_DATA_TYPE is SQL_DATETIME, this column contains the date/time subcode, as it appears in the DATETIME_INTERVAL_CODE field in the descriptor. For data types other than date/time types, this column is null.</td>
</tr>
<tr>
<td>NUM_PREC_RADIX</td>
<td>INTEGER</td>
<td>If DATA_TYPE is an approximate numeric data type, this column contains the value 2 because NUM_PREC specifies a number of bits. For exact numeric data types, this column contains the value 10 because NUM_PREC specifies a number of decimal digits. Otherwise, this column is null. By combining...</td>
</tr>
</tbody>
</table>
### GetTypeInfo()

#### Column Name | Type | Meaning
--- | --- | ---
 |  |  | the precision with the radix, an application can calculate the maximum number that the column can hold.

#### RETURN VALUE

- `[SQL_SUCCESS_WITH_INFO]` can result when implementation-defined warnings are produced.

#### DIAGNOSTICS

There must not be an open cursor on `StatementHandle` ("24000").

The `DataType` argument must be one of the defined values or an implementation-defined value ("HY004").

#### APPLICATION USAGE

The application can call `GetTypeInfo()` at any time that the connection handle on which `StatementHandle` was allocated is in the connected state. It may be called for a single data type or for all data types supported by the server. The result set may contain more than one row for a single SQL data type.

#### EXAMPLE USAGE (C)

```c
#include <sqlcli.h>

SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLSMALLINT DataType;
...
ReturnCode = SQLGetInfo(StatementHandle, DataType);
```

#### EXAMPLE USAGE (COBOL)

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 DATATYPE PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRGetTypeInfo" USING STATEMENTHANDLE, DATATYPE, RETURNCODE.
```

* Columns of this data type are generally not updatable and the value remains constant for the life of the row.
NAME

NumResultCols — Get the number of columns in a result set (concise)

SYNOPSIS

FUNCTION NumResultCols
   (StatementHandle INTEGER,
    ColumnCount SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION

The NumResultCols() function has the following parameters:

StatementHandle (input)
   Statement handle.

ColumnCount (output)
   Number of columns in the result set.

Calling NumResultCols() sets the value of ColumnCount to the number of columns in the result set associated with StatementHandle, or to zero if the statement cannot return columns.

RETURN VALUE

[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

DIAGNOSTICS

The application must have prepared StatementHandle (by calling Prepare() or ExecDirect()) before the call to NumResultCols() (‘HY010’).

APPLICATION USAGE

The application can call NumResultCols() to retrieve the number of columns in a result set without having to obtain a separate descriptor handle. Conceptually, NumResultCols() calls GetStmtAttr() to obtain the implementation row descriptor handle, then calls GetDescField() to retrieve the COUNT field from that descriptor.

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLSMALLINT ColumnCount;
...
ReturnCode = SQLNumResultCols(StatementHandle, &ColumnCount);
```

EXAMPLE USAGE (COBOL)

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 COLUMNCOUNT PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRNumResultCols" USING STATEMENTHANDLE, COLUMNCOUNT, RETURNCODE.
```

SEE ALSO

SetDescField(), SetDescRec(), DescribeCol(), BindCol()
NAME
ParamData — Address the next dynamic parameter requiring data at execution time

SYNOPSIS
FUNCTION ParamData
  (StatementHandle INTEGER,
   Value ANY)
RETURNS (SMALLINT)

DESCRIPTION
An application calls ParamData() in conjunction with PutData() to supply dynamic arguments
that it has previously indicated will be specified at execute time.

If there are additional dynamic parameters for which execute-time data is required, ParamData() identifies to the application the next such dynamic parameter, in increasing order of parameter
number. If no additional dynamic parameter requires data, ParamData() completes the
execution of the SQL statement previously specified by ExecDirect() or Execute().

The function has the following parameters:

StatementHandle (input)
  Statement handle.

Value (output)
  If the function returns [SQL_NEED_DATA], this returns the contents of the DATA_PTR field
  of the record in the application parameter descriptor that relates to the dynamic parameter
  for which the implementation requires information. For any other return value, Value is
  undefined.

RETURN VALUE
[SQL_SUCCESS], [SQL_NEED_DATA], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO],
[SQL_NO_DATA] or [SQL_INVALID_HANDLE]. [SQL_SUCCESS_WITH_INFO] can result
when implementation-defined warnings are produced.

The return value [SQL_NEED_DATA] indicates that the implementation is ready to receive the
next execute-time dynamic argument. The Value argument returns the DATA_PTR field of the related record in the application parameter descriptor. If BindParam() was used to set
DATA_PTR in the application parameter descriptor, the Value argument is set to the value the application had passed as the ParamValue argument of BindParam().

The return values [SQL_INVALID_HANDLE] and [SQL_ERROR] (with the DIAGNOSTICS
listed below) indicate errors in the call.

The return values [SQL_SUCCESS], [SQL_SUCCESS_WITH_INFO], [SQL_NO_DATA] or
[SQL_ERROR] (with DIAGNOSTICS other than those listed below) can be produced if provision of dynamic arguments is complete and the call to ParamData() executed the original SQL statement. In this case the return value is the return value that ExecDirect() would have returned if called directly.

DIAGNOSTICS
If no additional dynamic arguments are required, ParamData() executes the original SQL
statement, and can produce any of the diagnostics listed in ExecDirect().

A previous call to ExecDirect() or Execute() must have returned [SQL_NEED_DATA], indicating
that additional dynamic arguments are required before the statement associated with
StatementHandle can be executed (‘HY010’).

The most recent function call on StatementHandle must not be another call to
ParamData() (‘HY010’).
The application cannot perform any other operation with `StatementHandle` while a data-at-execute dialogue is in progress; nor can it call `EndTran()` nor set any connection attribute that would have an impact on the treatment of the statement handle ("HY010").

**APPLICATION USAGE**

The implementation requests that the application call `ParamData()` when a previous call to `ExecDirect()` or `Execute()` returns [SQL_NEED_DATA] or a previous call to `PutData()` returns [SQL_SUCCESS]. This is always based on a prior declaration by the application that certain dynamic arguments will be passed at execute time. (For an overview of passing dynamic arguments at execute time, see Section 4.3.2 on page 48.)

If the application specifies more than one dynamic parameter whose value will be passed at execute time, it should set the DATA_PTR field of each such record of the application parameter descriptor to a unique value. The application can set this field directly by calling `SetDescField()` or `SetDescRec()`, or by using the `ParamValue` argument on a call to `BindParam()`. Each call to `ParamData()` returns this value as the `Value` argument, indicating the dynamic parameter for which the implementation seeks data on this occasion. The application may use the DATA_PTR field as a means of passing arbitrary information to the routine that supplies dynamic arguments at execute time.

In programming languages where the application cannot use DATA_PTR, the application cannot use the `Value` parameter to obtain meaningful identification of the dynamic argument needed on this occasion. However, the application can still supply data at execute time by relying on the rule that `ParamData()` requests dynamic arguments in increasing order of parameter number.

The application can call `Cancel()` to prematurely terminate the data-at-execute dialogue.

**EXAMPLE USAGE (C)**

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLPOINTER Value;
...
ReturnCode = SQLParamData(StatementHandle, &Value);
```

**EXAMPLE USAGE (COBOL)**

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 VALUE PICTURE type.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRParamData" USING STATEMENTHANDLE, VALUE, RETURNCODE.
```

**SEE ALSO**

`BindParam()`, `PutData()`, `SetDescField()`, `SetDescRec()`
NAME
Prepare — Prepare an SQL statement for execution

SYNOPSIS
FUNCTION Prepare
   (StatementHandle INTEGER,
    StatementText CHAR,
    TextLength INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The Prepare() function has the following parameters:

StatementHandle (input)
Statement handle.

StatementText (input)
SQL text string, using question-mark (?) characters as dynamic parameter markers.

TextLength (input)
Length in octets of StatementText.

Calling Prepare() prepares the SQL statement text in StatementText for execution. The application can subsequently refer to the SQL statement using StatementHandle.

Prepare() populates the implementation row descriptor with information describing the columns of the result set.

Dynamic Parameters
It is implementation-defined which of the following occurs during a call to Prepare():

- The implementation sets the fields of the implementation parameter descriptor to describe the number, data type and type attributes of dynamic parameters in the SQL statement (this corresponds to the DESCRIBE INPUT feature of embedded SQL). If there are no dynamic parameters, the descriptor's COUNT field is set to 0.

- The implementation does not modify the implementation parameter descriptor.

Section 4.3.1 on page 47 describes the cases in which the application must gain access to the implementation parameter descriptor.

For a cursor-specification, Prepare() generates a cursor name unless one already exists.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

DIAGNOSTICS
The statement text must be a valid SQL statement other than COMMIT or ROLLBACK (‘42000’).

The current user must have any privileges required to execute the SQL statement (‘42000’).*

* It is implementation-defined whether the call to Prepare() reports violations of the rule(s) in this paragraph. If it does not, then the subsequent call to Execute() reports such violations.
There must not be an open cursor on StatementHandle ('24000').

For a dynamic positioned DELETE or a dynamic positioned UPDATE statement, the cursor referenced by the statement must be open and must be defined on a separate statement handle under the same connection handle ('34000').

The following diagnostics, defined in the X/Open SQL specification, can occur based on the value of StatementText:

- **Cardinality violation**
  - '21S01' — Insert value does not match column list.
  - '21S02' — Degree of derived table does not match column list.
- **Syntax error or access violation†**
  - '42000' — Base table or view already exists.
  - '42S01' — Base table or view not found.
  - '42S11' — Index already exists.
  - '42S12' — Index not found.
  - '42S21' — Column already exists.
  - '42S22' — Column not found.

**EXAMPLE USAGE (C)**

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLCHAR *StatementText;
SQLINTEGER TextLength;
...
ReturnCode = SQLPrepare(StatementHandle,
                         StatementText, TextLength);
```

**EXAMPLE USAGE (COBOL)**

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 STATEMENTTEXT PICTURE type.
01 TEXTLENGTH PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRPrepare" USING STATEMENTHANDLE,
     STATEMENTTEXT, TEXTLENGTH, RETURNCODE.
```

**SEE ALSO**

SetCursorName(), Execute(), ExecDirect()

† The codes of the form '42Snn' are 'S00nn' in the X/Open SQL specification.
NAME
PutData — Send part or all of a dynamic argument at execute time

SYNOPSIS
FUNCTION PutData
(StatementHandle INTEGER,
 Data ANY,
 StrLen_or_Ind INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
An application calls PutData(), after calling ParamData(), to supply part or all of a dynamic argument at execute time. The function has the following parameters:

StatementHandle (input)
Statement handle.

Data (input)
All or part of a dynamic argument to be used in execution of a statement on StatementHandle.

StrLen_or_Ind (input)
A variable whose value is interpreted as follows:

• If a null value is to be used as the parameter, StrLen_or_Ind must contain the value SQL_NULL_DATA.

• If the ValueType specified when binding the parameter is SQL_CHAR:
  — If the data in Data contains a null-terminated string, StrLen_or_Ind must either contain the length in octets of the string in Data or contain the value SQL_NTS.
  — If the data in Data is not null-terminated, StrLen_or_Ind must contain the length in octets of the string in Data.

• Otherwise, StrLen_or_Ind is ignored.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE]. [SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

DIAGNOSTICS
The last call to ExecDirect(), Execute() or ParamData() on StatementHandle must have returned [SQL_NEED_DATA], indicating that the implementation needs the application to provide dynamic arguments (‘HY010’).

Previous consecutive calls to PutData() must not have occurred since the call that reported [SQL_NEED_DATA] unless the application data type of the dynamic parameter is character string (‘HY019’).

The following diagnostics relate to interpreting, converting, and assigning the dynamic argument. It is implementation-defined whether they are reported by PutData(), by the next call to ParamData(), or by the call to ParamData() that ultimately executes the SQL statement:

• The length in octets of the string in Data must not exceed the maximum length of the dynamic parameter (‘22001’).

• The value of Data must be compatible with the data type of the column (‘07001’).
• A date/time dynamic argument specified as text must contain only valid characters ('2007').
• Other data conversion diagnostics specified in Section 5.4.10 on page 76 may occur.

If the application calls `PutData()` more than once since the call that returned [SQL_NEED_DATA], none of those calls may specify a null value ('HY020').

APPLICATION USAGE

After one or more calls to `PutData()`, the application calls `ParamData()` on the same `StatementHandle`. This declares that the transfer of the given dynamic argument is complete and informs the application whether there are other dynamic parameters for which the implementation requires values.

If the application calls `PutData()` more than once before the call to `ParamData()` that completes the transfer of a dynamic argument, the implementation concatenates the data passed as the `Data` argument on the sequential calls to `PutData()` to obtain the dynamic argument.

For an overview of passing dynamic arguments at execute time, see Section 4.3.2 on page 48.

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLPOINTER Data;
SQLINTEGER StrLen_or_Ind;
...
ReturnCode = SQLPutData(StatementHandle, Data, StrLen_or_Ind);
```

EXAMPLE USAGE (COBOL)

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 DATA              PICTURE type.
01 STRLEN-OR-IND    PICTURE S9(9) BINARY.
01 RETURNCODE        PICTURE S9(4) BINARY.
...
CALL "SQLRPutData" USING STATEMENTHANDLE, DATA, STRLEN-OR-IND, RETURNCODE.
```

SEE ALSO

`BindParam()`, `ParamData()`, `SetDescField()`, `SetDescRec()`
NAME
getRow — Get the number of rows affected by an SQL statement

SYNOPSIS

```c
FUNCTION RowCount
    (StatementHandle INTEGER,
     RowCount INTEGER)
RETURNS (SMALLINT)
```

DESCRIPTION

The `RowCount()` function has the following parameters:

- **StatementHandle [input]**
  - Statement handle.

- **RowCount [output]**
  - The count of rows affected.

Calling `RowCount()` sets `RowCount` to the row count (see Section 5.3.2 on page 72) resulting from the most recent call to `ExecDirect()` or `Execute()`.

RETURN VALUE


DIAGNOSTICS

The `RowCount()` function cannot be called prior to calling `ExecDirect()` or `Execute()` for `StatementHandle` (HY010).

APPLICATION USAGE

An application cannot use both `RowCount()` and `GetDiagField()`/`GetDiagRec()` to obtain status information for a statement handle. An application should fetch all status information for the statement handle before calling `RowCount()`, since `RowCount()` always resets the diagnostic area for the statement handle.

An application can call `RowCount()` even if it has called other CLI functions since the call to `ExecDirect()` or `Execute()`.

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLINTEGER RowCount;
...
ReturnCode = SQLRowCount(StatementHandle, &RowCount);
```

EXAMPLE USAGE (COBOL)

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 ROWCOUNT PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRRowCount" USING STATEMENTHANDLE, ROWCOUNT, RETURNCODE.
```

SEE ALSO

- `ExecDirect()`, `Execute()`, `GetDiagField()`
NAME
SetConnectAttr — Set a connection attribute

SYNOPSIS
FUNCTION SetConnectAttr
(ConnectionHandle INTEGER,
 Attribute INTEGER,
 Value ANY,
 StringLength INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The application can call SetConnectAttr() to set an attribute for ConnectionHandle in order to affect the subsequent processing of CLI functions associated with ConnectionHandle. The function has the following parameters:

ConnectionHandle (input)
Connection handle.

Attribute (input)
The attribute whose value is to be set. It must be a settable attribute. For a list of valid attributes, see Connection Attributes on page 29.

Value (input)
The desired value for Attribute. The application must provide an argument of an appropriate type for Attribute.

StringLength (input)
Length in octets of Value, if the attribute value is a character string; otherwise ignored.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].
[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

DIAGNOSTICS
Attribute must be one of the defined attributes or an implementation-defined value (‘HY092’).

Value must be one of the defined values for the specified Attribute (‘HY024’).

APPLICATION USAGE
The application can call SetConnectAttr() at any time between the allocation and the freeing of ConnectionHandle. The only other function that can change the values of any connection attributes is Connect(). Allocating a connection handle sets the values of some attributes to default values.

The application can set any connection attribute more than once. Each subsequent call overrides the value specified by any previous calls for the same attribute and connection handle. The operative value of the connection attribute is the attribute’s value at the time the application calls an affected CLI function.

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDBC ConnectionHandle;
SQLINTEGER Attribute;
SQLPOINTER Value;
SQLINTEGER StringLength;
...  
ReturnCode = SQLSetConnectAttr(ConnectionHandle,
   Attribute, Value, StringLength);

EXAMPLE USAGE (COBOL)
   01 CONNECTIONHANDLE   PICTURE S9(9) BINARY.
   01 ATTRIBUTE           PICTURE S9(9) BINARY.
   01 VALUE               PICTURE type.
   01 STRINGLENGTH        PICTURE S9(9) BINARY.
   01 RETURNCODE          PICTURE S9(4) BINARY.
...
   CALL "SQLRSetConnectAttr" USING CONNECTIONHANDLE,
      ATTRIBUTE, VALUE, STRINGLENGTH, RETURNCODE.

SEE ALSO
   AllocHandle(), GetConnectAttr(), SetConnectOption()
NAME

SetConnectOption — Set a connection attribute (deprecated)

SYNOPSIS

FUNCTION SetConnectOption
  (ConnectionHandle INTEGER,
   Option SMALLINT,
   Value ANY)
RETURNS (SMALLINT)

DESCRIPTION

The application can call `SetConnectOption()` to set an attribute for `ConnectionHandle` in order to affect the subsequent processing of CLI functions associated with `ConnectionHandle`. The function has the following parameters:

`ConnectionHandle` (input)
  Connection handle.

`Option` (input)
  The desired connection attribute.

`Value` (input)
  The appropriate value for the attribute specified by `Option`. Depending on the value of `Option`, this points either to a 32-bit integer value or to a character string.

RETURN VALUE

[SQL_SUCCESS], [SQL_ERROR], [SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

DIAGNOSTICS

`Option` must be one of the defined attributes or an implementation-defined value ('HY092').

`Value` must be one of the defined values for the specified `Attribute` ('HY024').

APPLICATION USAGE

The application can call `SetConnectOption()` at any time between the allocation and the freeing of `ConnectionHandle`. The only other function that may change the values of a connection attribute is `Connect()`, since it is possible that an attribute set prior to `Connect()`, or left over from a previous connection, may no longer be valid for the new connection. Allocating a connection handle sets the values of some attributes to default values.

The application can set any connection attribute more than once. Each subsequent call overrides the value specified by any previous calls for the same attribute and connection handle. The operative value of the connection attribute is the attribute's value at the time the application calls an affected CLI function.

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDBC ConnectionHandle;
SQLSMALLINT Option;
SQLPOINTER Value;
...
ReturnCode = SQLSetConnectOption(ConnectionHandle,
   Option, Value);
```
SetConnectOption()

EXAMPLE USAGE (COBOL)

01 CONNECTIONHANDLE PICTURE S9(9) BINARY.
01 OPTION PICTURE S9(4) BINARY.
01 VALUE PICTURE type.
01 RETURNCODE PICTURE S9(4) BINARY.

CALL "SQLRSelConnectOption" USING CONNECTIONHANDLE, SPECIAL OPTIONS, VALUE, RETURNCODE.

SEE ALSO

AllocHandle(), SetConnectAttr(), GetConnectOption()
NAME
SetCursorName — Set the name of a cursor

SYNOPSIS
FUNCTION SetCursorName
StatementHandle INTEGER,
CursorName CHAR,
NameLength SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION
The application can call SetCursorName() to assign a name to a cursor. This is an optional step; the implementation implicitly gives each cursor a name (starting with ‘SQLCUR’ or ‘SQL_CUR’) when it creates the cursor.

The function has the following parameters:
StatementHandle (input)
Statement handle.

CursorName (input)
Cursor name. The implementation trims any leading and trailing spaces and, unless the result begins and ends with double quotes (in which case it is a delimited identifier; see Delimited Identifiers on page 21), converts lower-case letters to upper case.

NameLength (input)
Length in octets of CursorName.

Calling SetCursorName() associates the name CursorName with StatementHandle.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

DIAGNOSTICS
StatementHandle must be in the allocated or prepared state (see Table B-2 on page 242) (‘HY010’).

The conceptual result of evaluating CursorName as an identifier (see DESCRIPTION above) must satisfy the following tests:
• It must be a valid cursor-name as defined by the X/Open SQL specification (‘34000’).
• It must not start with ‘SQLCUR’ or ‘SQL_CUR’ (in order to avoid inadvertent duplication of names generated by the implementation) (‘34000’).
• If it exceeds [SQL_MAX_ID_LENGTH] characters in length, then the implementation uses only the first [SQL_MAX_ID_LENGTH] characters (‘01004’).
• No cursor with the same name may be already defined within the connection associated with StatementHandle (‘34000’).

APPLICATION USAGE
The cursor name that the application specifies is used for any subsequent cursors prepared with StatementHandle (until the application specifies a different cursor name by calling SetCursorName() again on StatementHandle).

If a name has not yet been set for a statement handle when a cursor is prepared, the implementation generates a unique cursor name. As long as that statement remains in the
SetCursorName()  

In a prepared state, the application may call SetCursorName() to override the generated name.  

For efficient processing, applications should not include any leading or trailing spaces in CursorName; if CursorName contains a delimited identifier, applications should position the first double quote as the first character in CursorName.

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLCHAR *CursorName;
SQLSMALLINT NameLength;
...
ReturnCode = SQLSetCursorName(StatementHandle,
                                 CursorName, NameLength);
```

EXAMPLE USAGE (COBOL)

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 CURSORNAME PICTURE X(18).
01 NAMELENGTH PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRSetCursorName" USING STATEMENTHANDLE,
       CURSORNAME, NAMELENGTH, RETURNCODE.
```

SEE ALSO

Prepare(), GetCursorName(), Execute(), ExecDirect()
NAME

SetDescField — Set a descriptor field (extensible)

SYNOPSIS

FUNCTION SetDescField
(DescriptorHandle INTEGER,
 RecNumber SMALLINT,
 FieldIdentifier SMALLINT,
 Value ANY,
 BufferLength INTEGER)
RETURNS (SMALLINT)

DESCRIPTION

The SetDescField() function sets the value of one field of a specified descriptor record associated with DescriptorHandle.

The application can call the function to set a field of a descriptor record beyond the records enumerated by the descriptor’s COUNT field. In this case, the function implicitly increases the value of the COUNT field to RecNumber.

The function has the following parameters:

DescriptorHandle (input)
Descriptor handle.

RecNumber (input)
The record number for which the specified field in the descriptor is to be set. The first record is number 1.

FieldIdentifier (input)
The identifier of the field to be set. This can be one of the following:

SQL_DESC_COUNT (type: SMALLINT)
Ignore RecNumber. Set the number of records in the descriptor.

SQL_DESC_TYPE (type: SMALLINT)
Set the TYPE field of RecNumber.

SQL_DESC_LENGTH (type: INTEGER)
Set the LENGTH field of RecNumber. If Type specifies a character-string data type, this is the maximum number of characters to store in Data.

SQL_DESC_OCTET_LENGTH (type: INTEGER)
Set the OCTET_LENGTH field of RecNumber. If Type specifies a character-string data type, this is the maximum number of octets to store in Data, including the null terminator if applicable.

SQL_DESC_OCTET_LENGTH_PTR (type: SQLPOINTER)
Set the OCTET_LENGTH_PTR field of RecNumber. Value must point to an application variable whose type is INTEGER. For an application row descriptor, this is the variable that will receive the length in octets of character-string output data (even if it does not entirely fit in the output buffer). For an application parameter descriptor, this is the variable that will describe the length in octets of a character-string dynamic argument.

SQL_DESC_Precision (type: SMALLINT)
Set the PRECISION field of RecNumber.

SQL_DESC_SCALE (type: SMALLINT)
Set the SCALE field of RecNumber.
SetDescField()

**SQL_DESC_DATETIME_INTERVAL_CODE** (type: SMALLINT)
Set the DATETIME_INTERVAL_CODE field of RecNumber.

**SQL_DESC_INDICATOR_PTR** (type: SQLPOINTER)
Set the INDICATOR_PTR field of RecNumber. Value must point to an application variable whose value is SMALLINT.

**SQL_DESC_DATA_PTR** (type: SQLPOINTER)
Set the DATA_PTR field of RecNumber. If Value points to an application variable, the descriptor record becomes bound to that variable. If Value is a null pointer, the descriptor record becomes unbound.

**Value** (input)
The desired value for FieldIdentifier. The application must provide an argument of an appropriate type for FieldIdentifier.

**BufferLength** (input)
Length in octets of Value.

If FieldIdentifier does not denote a string, then the implementation ignores BufferLength.

**RETURN VALUE**
[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE]. The [SQL_SUCCESS_WITH_INFO] outcome occurs if implementation-defined warnings are produced.

**DIAGNOSTICS**
For all values of FieldIdentifier except SQL_DESC_COUNT, RecNumber must be 1 or greater ('07009').

FieldIdentifier must be one of the defined values or an implementation-defined value ('HY091'). Not all fields retrievable through GetDescField() are settable by SetDescField().

DescriptorHandle must not identify an implementation row descriptor ('HY016').

If FieldIdentifier is SQL_DESC_DATA_PTR, the descriptor's consistency is checked; see Section 5.4.9 on page 75.

**APPLICATION USAGE**
When SetDescField() specifies a value for any field other than COUNT and the deferred fields DATA_PTR, INDICATOR_PTR and OCTET_LENGTH_PTR, the record becomes unbound. When SetDescField() specifies a value for the TYPE field, some other fields for the record receive default values as indicated below and the remaining fields are set to undefined values:

<table>
<thead>
<tr>
<th>Application Sets TYPE to:</th>
<th>Other Fields Implicitly Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR or SQL_VARCHAR</td>
<td>LENGTH is set to 1.</td>
</tr>
<tr>
<td>SQL_DATETIME</td>
<td>PRECISION is set to 0.</td>
</tr>
<tr>
<td>SQL_DECIMAL or SQL_NUMERIC</td>
<td>SCALE is set to 0; PRECISION is set to the implementation-defined default precision for the respective data type.</td>
</tr>
<tr>
<td>SQL_FLOAT</td>
<td>PRECISION is set to the implementation-defined default precision for FLOAT.</td>
</tr>
</tbody>
</table>

For a descriptor whose TYPE field is SQL_DATETIME, when SetDescField() specifies a value for the DATETIME_INTERVAL_CODE field, the PRECISION field for the record receives a default value as indicated below and sets the remaining fields to undefined values:

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EX</td>
</tr>
</tbody>
</table>

X/Open CAE Specification
When an application uses SetDescField() instead of SetDescRec() to set fields of a descriptor record, it must first declare the data type. Then, if the values implicitly set (as described above) are unacceptable, it can call SetDescField() to explicitly specify applicable data type attributes.

**EXAMPLE USAGE (C)**

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDESC DescriptorHandle;
SQLSMALLINT RecNumber;
SQLSMALLINT FieldIdentifier;
SQLPOINTER Value;
SQLINTEGER BufferLength;
...
ReturnCode = SQLSetDescField(DescriptorHandle, RecNumber, 
FieldIdentifier, Value, BufferLength);
```

**EXAMPLE USAGE (COBOL)**

```cobol
01 DESCRIPTORHANDLE PICTURE S9(9) BINARY.
01 RECNUMBER PICTURE S9(4) BINARY.
01 FIELDIDENTIFIER PICTURE S9(4) BINARY.
01 VALUE PICTURE type.
01 BUFFERLENGTH PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRSetDescField" USING STATEMENTHANDLE, RECNUMBER, 
FIELDIDENTIFIER, VALUE, BUFFERLENGTH, RETURNCODE.
```

**SEE ALSO**

SetDescRec()
SetDescRec( )

NAME
SetDescRec — Set a descriptor record (concise)

SYNOPSIS

FUNCTION SetDescRec
(DescriptorHandle INTEGER,
 RecNumber SMALLINT,
 Type SMALLINT,
 SubType SMALLINT,
 Length INTEGER,
 Precision SMALLINT,
 Scale SMALLINT,
 Data ANY,
 StringLength INTEGER,
 Indicator SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION
The SetDescRec() function initialises a descriptor record associated with DescriptorHandle.

The application can call the function to set a field of a descriptor record beyond the records
enumerated by the descriptor's COUNT field. In this case, the function implicitly increases the
value of the COUNT field to RecNumber.

The function has the following parameters:
DescriptorHandle (input)
 Descriptor handle.
RecNumber (input)
 The record number from which the description in the descriptor is to be set. The first record
is number 1.
Type (input)
 The TYPE field for the record.
SubType (input)
 The DATETIME_INTERVAL_CODE field, for records whose TYPE is SQL_DATETIME.
Length (input)
 The OCTET_LENGTH field for the record.
Precision (input)
 The PRECISION field for the record.
Scale (input)
 The SCALE field for the record.
Data (deferred)
 The DATA_PTR field for the record.
StringLength (deferred)
 The OCTET_LENGTH_PTR field for the record.
Indicator (deferred)
 The INDICATOR_PTR field for the record.

If Type specifies a character-string data type, SetDescRec() sets LENGTH to indicate the number
of characters the character string can contain. This is the largest number of characters that can
be represented in Length octets, after reserving space for an optional null terminator; except that
the resulting value is limited to the largest valid length for a CHARACTER VARYING data type.

**RETURN VALUE**

- [SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

The [SQL_SUCCESS_WITH_INFO] outcome occurs if implementation-defined warnings are produced.

**DIAGNOSTICS**

- **RecNumber** must be 1 or greater (’07009’).
- **DescriptorHandle** must not identify an implementation row descriptor (’HY016’).

The descriptor’s consistency is checked; see Section 5.4.9 on page 75.

**APPLICATION USAGE**

In languages that allow the use of pointers, the application can provide a null pointer for any of **Data**, **Indicator** and **StringLength**, in order to set the respective field of the descriptor record to a null pointer. For example, using **Data** to set the DATA_PTR field to a null pointer unbinds the record.

The application can set other fields of the descriptor record, by using calls to **SetDescField()**.

**EXAMPLE USAGE (C)**

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHDESC DescriptorHandle;
SQLSMALLINT RecNumber;
SQLSMALLINT Type;
SQLSMALLINT SubType;
SQLINTEGER Length;
SQLSMALLINT Precision;
SQLSMALLINT Scale;
SQLPOINTER Data;
SQLINTEGER StringLength;
SQLSMALLINT Indicator;
...
ReturnCode = SQLSetDescRec(DescriptorHandle, RecNumber,
    Type, SubType, Length, Precision, Scale, Data,
    &StringLength, &Indicator);
```

**EXAMPLE USAGE (COBOL)**

```cobol
01 DESCRIPTORHANDLE PICTURE S9(9) BINARY.
01 RECNUMBER PICTURE S9(4) BINARY.
01 TYPE PICTURE S9(4) BINARY.
01 SUBTYPE PICTURE S9(4) BINARY.
01 LENGTH PICTURE S9(9) BINARY.
01 PRECISION PICTURE S9(4) BINARY.
01 SCALE PICTURE S9(4) BINARY.
01 DATA PICTURE type.
01 STRINGLENGTH PICTURE S9(9) BINARY.
01 INDICATOR PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
```
CALL "SQLRSetDescRec" USING DESCRIPTORHANDLE, RECNUMBER,
    TYPE, SUBTYPE, LENGTH, PRECISION, SCALE, DATA,
    STRINGLENGTH, INDICATOR, RETURNCODE.

SEE ALSO
    SetDescField()
NAME
SetEnvAttr — Set an environment attribute

SYNOPSIS
FUNCTION SetEnvAttr
( EnvironmentHandle INTEGER,
  Attribute INTEGER,
  Value ANY,
  StringLength INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The application can call SetEnvAttr() to set an attribute for EnvironmentHandle in order to affect
the subsequent processing of CLI functions for EnvironmentHandle.

The function has the following parameters:
EnvironmentHandle (input)
Environment handle.

Attribute (input)
The attribute whose value is to be set. It must be a settable attribute. For a list of valid
attributes, see Environment Attribute on page 28.

Value (input)
The desired value for Attribute. The application must provide an argument of an
appropriate type for Attribute.

StringLength (input)
Length in octets of Value, if the attribute value is a character string; otherwise, unused.

If Attribute does not denote a string, then the implementation ignores StringLength.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or
[SQL_INVALID_HANDLE].

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are
produced.

DIAGNOSTICS
Applications cannot set environment attributes while connections are allocated on
EnvironmentHandle ("HY011").

Attribute must be one of the defined attributes or an implementation-defined value ("HY092").

Value must be one of the defined values for the specified Attribute ("HY024").

APPLICATION USAGE
No other CLI function changes the value of any environment attribute except that AllocHandle()
sets some attributes to default values, as described above, for the environment handle it
allocates. SetEnvAttr() can be called more than once for any environment attribute; each
subsequent call overrides the value specified by any previous calls for the same attribute and
EnvironmentHandle. The operative value of the environment attribute is the attribute's value at
the time the application calls an affected CLI function.

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHENV EnvironmentHandle;

Data Management: SQL Call Level Interface (CLI)
SQLINTEGER Attribute;
SQLPOINTER Value;
SQLINTEGER StringLength;
...

ReturnCode = SQLSetEnvAttr(EnvironmentHandle,
   Attribute, Value, StringLength);

EXAMPLE USAGE (COBOL)
01 ENVIRONMENTHANDLE PICTURE S9(9) BINARY.
01 ATTRIBUTE PICTURE S9(9) BINARY.
01 VALUE PICTURE type.
01 STRINGLENGTH PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...

CALL "SQLRSetEnvAttr" USING ENVIRONMENTHANDLE,
   ATTRIBUTE, VALUE, STRINGLENGTH, RETURNCODE.

SEE ALSO
AllocHandle(), GetEnvAttr()
NAME
SetParam — Bind a dynamic parameter (concise)

SYNOPSIS

```
FUNCTION SetParam
    (StatementHandle INTEGER,
     ParameterNumber SMALLINT,
     ValueType SMALLINT,
     ParameterType SMALLINT,
     LengthPrecision INTEGER,
     ParameterScale SMALLINT,
     ParameterValue ANY,
     StrLen_or_Ind INTEGER)
    RETURNS (SMALLINT)
```

DESCRIPTION
Refer to `BindParam()`.
NAME
SetStmtAttr — Set a statement attribute

SYNOPSIS
FUNCTION SetStmtAttr
(StatementHandle INTEGER,
 Attribute INTEGER,
 Value ANY,
 StringLength INTEGER)
RETURNS (SMALLINT)

DESCRIPTION
The application can call SetStmtAttr() to set an attribute for StatementHandle in order to affect subsequent CLI function calls associated with StatementHandle.

The function has the following parameters:

StatementHandle (input)
Statement handle.

Attribute (input)
The attribute whose value is to be set. It must be a settable attribute. For a list of valid attributes, see Statement Attributes on page 30.

Value (input)
The desired value for Attribute. The application must provide an argument of an appropriate type for Attribute.

StringLength (input)
Length in octets of Value, if the attribute value is a character string; otherwise, unused.

If Attribute does not denote a string, then the implementation ignores StringLength.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

DIAGNOSTICS
Attribute must be one of the defined attributes or an implementation-defined value ('HY092'). (If Attribute is an attribute defined in this document but marked optional, and if the implementation does not support that option, the diagnostic is 'HYC00'.)

Value must be one of the defined values for the specified Attribute ('HY024'). For example, the values for the read-only options described in GetStmtAttr() cannot be used.

If Attribute is SQL_ATTR_CURSOR_SENSITIVITY or SQL_ATTR_CURSOR_SCROLLABLE, there must not be a prepared statement or an open cursor on StatementHandle ('HY011').

If Attribute is SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC, and if Value is a descriptor handle obtained by a call to GetStmtAttr(), then the values of Attribute and StatementHandle must be the same as they were on the call to GetStmtAttr() ('HY017').

APPLICATION USAGE
SetStmtAttr() can be called at any time between the allocation and the freeing of StatementHandle. No other CLI function changes the value of any statement attribute except that AllocHandle() may set some attributes to default values for the statement handle it allocates.

SetStmtAttr() can be called more than once for any statement attribute; each subsequent call
SetStmtAttr() overrides the value specified by any previous calls for the same attribute and statement handle. Setting attributes for a statement handle is independent of associating text of an SQL statement with that handle. The operative value of the statement attribute is the attribute's value at the time the application calls the relevant CLI function (Fetch(), FetchScroll(), Prepare(), Execute() or ExecDirect(), depending on the attribute in question).

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLINTEGER Attribute;
SQLPOINTER Value;
SQLINTEGER StringLength;
...
ReturnCode = SQLSetStmtAttr(StatementHandle,
    Attribute, Value, StringLength);
```

EXAMPLE USAGE (COBOL)

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 ATTRIBUTE PICTURE S9(9) BINARY.
01 VALUE PICTURE type.
01 STRINGLENGTH PICTURE S9(9) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRSetStmtAttr" USING STATEMENTHANDLE,
    ATTRIBUTE, VALUE, STRINGLENGTH, RETURNCODE.
```

SEE ALSO

AllocHandle(), GetStmtAttr(), Prepare()
NAME
SetStmtOption — Set a statement attribute (deprecated)

SYNOPSIS
FUNCTION SetStmtOption
(StatementHandle INTEGER,
 Option SMALLINT,
 Value ANY)
RETURNS (SMALLINT)

DESCRIPTION
The application can call SetStmtOption() to set an attribute for StatementHandle in order to affect
the subsequent processing of CLI functions associated with StatementHandle. The function has
the following parameters:

StatementHandle (input)
Statement handle.

Option (input)
The attribute whose value is to be set. It must be a settable attribute. For a list of valid
attributes, see Statement Attributes on page 30.

Value (input)
The appropriate value for the attribute specified by Option. Depending on the value of
Option, this points either to a 32-bit integer value or to a character string.

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or
[SQL_INVALID_HANDLE].

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are
produced.

DIAGNOSTICS
Option must be one of the defined attributes or an implementation-defined value (‘HY092’). (If
Option is an attribute defined in this document but marked optional, and if the implementation
does not support that option, the diagnostic is ‘HYC00’.)

Value must be one of the defined values for the specified Option (‘HY024’).

If Option is SQL_ATTR_CURSOR_SENSITIVITY or SQL_ATTR_CURSOR_SCROLLABLE, there
must not be a prepared statement or an open cursor on StatementHandle (‘HY011’).

If Option is SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC, and if Value is a
descrriptor handle obtained by a call to GetStmtOption(), then the values of Option and
StatementHandle must be the same as they were on the call to GetStmtOption() (‘HY017’).

APPLICATION USAGE
The application can call SetStmtOption() at any time between the allocation and the freeing of
StatementHandle. Allocating a statement handle sets the values of some attributes to default
values.

The application can set any statement attribute more than once. Each subsequent call overrides
the value specified by any previous calls for the same attribute and statement handle. The
operative value of the statement attribute is the attribute's value at the time the application calls
an affected CLI function.

EXAMPLE USAGE (C)
#include <sqlcli.h>
SQLRETURN   ReturnCode;
SQLHSTMT    StatementHandle;
SQLSMALLINT Option;
SQLPOINTER  Value;
...
ReturnCode = SQLSetStmtOption(StatementHandle, Option, Value);

EXAMPLE USAGE (COBOL)
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 OPTION       PICTURE S9(4) BINARY.
01 VALUE        PICTURE type.
01 RETURNCODE   PICTURE S9(4) BINARY.
...
CALL "SQLRSetStmtOption" USING STATEMENTHANDLE, 
    OPTION, VALUE, RETURNCODE.

SEE ALSO
  AllocHandle(), SetStmtAttr(), GetStmtOption()
NAME

SpecialColumns — Get a unique row identifier for a table

SYNOPSIS

FUNCTION SpecialColumns

(StatementHandle INTEGER,
 IdentifierType SMALLINT,
 CatalogName CHAR,
 NameLength1 SMALLINT,
 SchemaName CHAR,
 NameLength2 SMALLINT,
 TableName CHAR,
 NameLength3 SMALLINT,
 Scope SMALLINT,
 Nullable SMALLINT)

RETURNS (SMALLINT)

DESCRIPTION

The SpecialColumns() function retrieves a result set that uniquely identifies a row of the specified table. The SpecialColumns() function can obtain information on pseudo-columns — that is, columns that an implementation may automatically create in addition to those the application specifies, such as a row identifier.

The function has the following parameters:

StatementHandle (input)
Statement handle.

IdentifierType (input)
Reserved for future definition by X/Open in order to support additional types of special columns. Must be set to SQL_ROW_IDENTIFIER.

CatalogName (input)
Buffer containing catalog name of table. If the SQL_ATTR_METADATA_ID attribute of StatementHandle is SQL_TRUE, then this is interpreted as an identifier (see Treatment as Identifier on page 54).

NameLength1 (input)
Length in octets of CatalogName.

SchemaName (input)
Buffer containing schema name of table. If the SQL_ATTR_METADATA_ID attribute of StatementHandle is SQL_TRUE, then this is interpreted as an identifier (see Treatment as Identifier on page 54).

NameLength2 (input)
Length in octets of SchemaName.

TableName (input)
Buffer containing name of table. If the SQL_ATTR_METADATA_ID attribute of StatementHandle is SQL_TRUE, then this is interpreted as an identifier (see Treatment as Identifier on page 54).

NameLength3 (input)
Length in octets of TableName.

Scope (input)
The application sets this argument to specify the period of time for which it requires that
the name returned in COLUMN_NAME be valid. Valid values for Scope are:

SQL_SCOPE_CURROW
SQL_SCOPE_TRANSACTION
SQL_SCOPE_SESSION

Nullable (input)
Set to SQL_NO_NULLS to restrict the result set of SpecialColumns() to describe only those columns of TableName that are not nullable; or to SQL_NULLABLE to not thus restrict the result set.

The SpecialColumns() functions returns a result set in which each row represents one column in TableName, including implementation-defined pseudo-columns that may exist in addition to the columns the application specified in the CREATE TABLE statement.

- If there is more than one column or set of columns that uniquely identifies a row of the table, then SpecialColumns() selects a row identifier using implementation-defined criteria.
- If there are no qualifying columns, then SpecialColumns() returns an empty result set.
- If TableName does not exist, then the result set is empty.
- If TableName is a view, then it is undefined whether the result set is empty.

The result set has at least these columns in the following order, and its rows are sorted by SCOPE:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOPE</td>
<td>SMALLINT</td>
<td>The duration for which the name in COLUMN_NAME is guaranteed to point to the same row. Valid values are the same as for the Scope argument. The rows in the result set are constrained by the Scope argument such that the value of SCOPE always indicates a duration at least as long as that indicated by Scope.</td>
</tr>
<tr>
<td>COLUMN_NAME</td>
<td>VARCHAR(128) NOT NULL</td>
<td>Name of the row identifier or name of the column that is or is part of the table’s primary key.</td>
</tr>
<tr>
<td>DATA_TYPE</td>
<td>SMALLINT</td>
<td>Identifies the type of the column and contains one of the Integer Data Type identifier values listed in Table 4-4 on page 58, except that, for the date/time data types, the column contains one of the Special Integer Data Type Identifiers listed in Table 4-5 on page 58.</td>
</tr>
<tr>
<td>TYPE_NAME</td>
<td>VARCHAR(128) NOT NULL</td>
<td>The textual name of the column’s data type; this can be an implementation-defined name for DATA_TYPE.</td>
</tr>
</tbody>
</table>
### SpecialColumns()

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLUMN_SIZE</td>
<td>INTEGER</td>
<td>If DATA_TYPE is SQL_CHAR or SQL_VARCHAR, then this column contains the maximum length in characters of the column. For date/time data types, this is the total number of characters required to display the value when converted to character. For numeric data types, this is the total number of digits.</td>
</tr>
<tr>
<td>BUFFER_LENGTH</td>
<td>INTEGER</td>
<td>The size in octets of the buffer that would be required to fetch the column if SQL_DEFAULT were specified (see Table 4-9 on page 61). This length excludes any null terminator. For exact numeric data types, this length accounts for the decimal point and the sign.</td>
</tr>
<tr>
<td>DECIMAL_DIGITS</td>
<td>SMALLINT</td>
<td>Defines the total number of significant digits to the right of the decimal point. For the date/time subtypes TIME and TIMESTAMP, this is the number of digits in the fractional seconds component. For the INTEGER and SMALLINT data types, it is 0. For the CHARACTER, CHARACTER VARYING, FLOAT, REAL and DOUBLE PRECISION data types, it is null.</td>
</tr>
<tr>
<td>PSEUDO_COLUMN</td>
<td>SMALLINT</td>
<td>Indicates whether or not the column is a pseudo-column, such as a row identifier, which would not be returned in Columns(). May be one of the following values: SQL_PC_NOT_PSEUDO The column is not a pseudo-column. SQL_PC_PSEUDO The column is a pseudo-column. SQL_PC_UNKNOWN It is unknown whether the column is a pseudo-column.</td>
</tr>
</tbody>
</table>

#### RETURN VALUE

[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

#### DIAGNOSTICS

There must not be an open cursor on StatementHandle(‘24000’).

If the implementation does not support qualification by catalog name, then CatalogName must be a null pointer or a zero-length string (‘HYC00’).

Scope must be SQL_SCOPE_CURROW, SQL_SCOPE_TRANSACTION or SQL_SCOPE_SESSION (‘HY092’).
IdentifierType must be SQL_ROW_IDENTIFIER ('HY092').

Nullable must be SQL_NO_NULLS or SQL_NULLABLE ('HY092').

**EXAMPLE USAGE (C)**
```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLSMALLINT IdentifierType;
SQLCHAR *CatalogName;
SQLSMALLINT NameLength1;
SQLCHAR *SchemaName;
SQLSMALLINT NameLength2;
SQLCHAR *TableName;
SQLSMALLINT NameLength3;
SQLSMALLINT Scope;
SQLSMALLINT Nullable;
...
ReturnCode = SQLSpecialColumns(StatementHandle, IdentifierType,
                             CatalogName, NameLength1, SchemaName, NameLength2, TableName,
                             NameLength3, Scope, Nullable);
```

**EXAMPLE USAGE (COBOL)**
```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 IDENTIFIERTYPE PICTURE S9(4) BINARY.
01 CATALOGNAME PICTURE X(128).
01 NAMELENGTH1 PICTURE S9(4) BINARY.
01 SCHEMANAME PICTURE X(128).
01 NAMELENGTH2 PICTURE S9(4) BINARY.
01 TABLENAME PICTURE X(128).
01 NAMELENGTH3 PICTURE S9(4) BINARY.
01 SCOPE PICTURE S9(4) BINARY.
01 NULLABLE PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRSpecialColumns" USING STATEMENTHANDLE, IDENTIFIERTYPE,
    CATALOGNAME, NAMELENGTH1, SCHEMANAME, NAMELENGTH2, TABLENAME,
    NAMELENGTH3, SCOPE, NULLABLE, RETURNCODE.
```

**RELATION TO STANDARDS**
This function is not present in the ISO CLI Draft International Standard.

**SEE ALSO**
- Statistics(), Tables()
NAME

Statistics — Get index information for a base table

SYNOPSIS

FUNCTION Statistics
  (StatementHandle INTEGER,
   CatalogName CHAR,
   NameLength1 SMALLINT,
   SchemaName CHAR,
   NameLength2 SMALLINT,
   TableName CHAR,
   NameLength3 SMALLINT,
   Unique SMALLINT,
   Reserved SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION

The Statistics() function retrieves information on indexes for a given base table. The function has the following parameters:

StatementHandle (input)
  Statement handle.

CatalogName (input)
  Buffer containing catalog name of table. If the SQL_ATTR_METADATA_ID attribute of StatementHandle is SQL_TRUE, then this is interpreted as an identifier (see Treatment as Identifier on page 54).

NameLength1 (input)
  Length in octets of CatalogName.

SchemaName (input)
  Buffer containing schema name of table. If the SQL_ATTR_METADATA_ID attribute of StatementHandle is SQL_TRUE, then this is interpreted as an identifier (see Treatment as Identifier on page 54).

NameLength2 (input)
  Length in octets of SchemaName.

TableName (input)
  Buffer containing name of table. If the SQL_ATTR_METADATA_ID attribute of StatementHandle is SQL_TRUE, then this is interpreted as an identifier (see Treatment as Identifier on page 54).

NameLength3 (input)
  Length in octets of TableName.

Unique (input)
  Set to SQL_INDEX_UNIQUE to direct Statistics() to return only unique indexes, or set to SQL_INDEX_ALL to obtain all indexes.

Reserved (input)
  Reserved for future definition by X/Open. Must be set to 0.

The Statistics() function returns index information in the form of a result set on the statement handle where each index column is represented by one row of the result set.

• If TableName does not exist, then the result set is empty.
- If `TableName` is a view, then it is undefined whether the result set is empty.

The result set has at least these columns in the following order, and its rows are sorted by `NON_UNIQUE`, `TYPE`, `INDEX_QUALIFIER`, `INDEX_NAME` and `ORDINAL_POSITION`, in that order:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_CAT</td>
<td>VARCHAR(128)</td>
<td>The name of the catalog containing <code>TABLE_SCHEMA</code>. See <strong>APPLICATION USAGE</strong> below.</td>
</tr>
<tr>
<td>TABLE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>The name of the schema containing <code>TABLE_NAME</code>.</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>VARCHAR(128)</td>
<td>The name of the table.</td>
</tr>
<tr>
<td>NON_UNIQUE</td>
<td>SMALLINT</td>
<td>Contains the value SQL_TRUE if the index values can be non-unique, or SQL_FALSE if the index values must be unique.</td>
</tr>
<tr>
<td>INDEX_QUALIFIER</td>
<td>VARCHAR(128)</td>
<td>The string that would be used to qualify the index name in the DROP INDEX statement. Appending a dot and the INDEX_NAME (see below) produces a full specification of the index. If the implementation does not support index qualifiers, this column is null.</td>
</tr>
<tr>
<td>INDEX_NAME</td>
<td>VARCHAR(128)</td>
<td>The name of the index.</td>
</tr>
<tr>
<td>TYPE</td>
<td>SMALLINT</td>
<td>The type of index returned: SQL_INDEX_CLUSTERED, SQL_INDEX_HASHED, or SQL_INDEX_OTHER. The meaning of these values is implementation-defined. Other implementation-defined values may be returned; in this case, the meanings of all other columns of the result set may be different from that specified here.</td>
</tr>
<tr>
<td>ORDINAL_POSITION</td>
<td>SMALLINT</td>
<td>Ordinal position of the column within the index. The first column is number 1.</td>
</tr>
<tr>
<td>COLUMN_NAME</td>
<td>VARCHAR(128)</td>
<td>The name of the column in the index.</td>
</tr>
<tr>
<td>ASC_OR_DESC</td>
<td>CHAR(1)</td>
<td>Contains the value 'A' if the order of the referenced column is ascending and 'D' if the order of the referenced column is descending.</td>
</tr>
</tbody>
</table>
RETURN VALUE

[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].

[SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

DIAGNOSTICS

There must not be an open cursor on StatementHandle ('24000').

If the implementation does not support qualification by catalog name, then the value of CatalogName must be a null pointer or a zero-length string ('HYC00').

The value of Unique must be SQL_INDEX_UNIQUE or SQL_INDEX_ALL; and the value of Reserved must be 0 ('HY092').

APPLICATION USAGE

The TABLE_CAT column may be null or may contain an empty string for several reasons discussed in Section 2.4.3 on page 22. If TABLE_CAT contains an empty string, the application should take care to not append '.' when forming a fully-qualified object name for use in an SQL statement.

EXAMPLE USAGE (C)

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLCHAR *CatalogName;
SQLSMALLINT NameLength1;
SQLCHAR *SchemaName;
SQLSMALLINT NameLength2;
SQLCHAR *TableName;
SQLSMALLINT NameLength3;
SQLSMALLINT Unique;
SQLSMALLINT Reserved;
...
ReturnCode = SQLStatistics(StatementHandle, CatalogName,
  NameLength1, SchemaName, NameLength2, TableName,
  NameLength3, Unique, Reserved);
```

EXAMPLE USAGE (COBOL)

```cobol
01 STATEMENTHANDLE PICTURE S9(9) BINARY.
01 CATALOGNAME PICTURE X(128).
01 NAMELENGTH1 PICTURE S9(4) BINARY.
01 SCHEMANAME PICTURE X(128).
01 NAMELENGTH2 PICTURE S9(4) BINARY.
01 TABLENAME PICTURE X(128).
01 NAMELENGTH3 PICTURE S9(4) BINARY.
01 UNIQUE PICTURE S9(4) BINARY.
01 RESERVED PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRStatistics" USING STATEMENTHANDLE, CATALOGNAME,
  NAMELENGTH1, SCHEMANAME, NAMELENGTH2, TABLENAME,
  NAMELENGTH3, UNIQUE, RESERVED, RETURNCODE.
```
RELATION TO STANDARDS
This function is not present in the ISO CLI Draft International Standard.

SEE ALSO
SpecialColumns(), Tables()
NAME
Tables — Get table information

SYNOPSIS

FUNCTION Tables
(StatementHandle INTEGER,
CatalogName CHAR,
NameLength1 SMALLINT,
SchemaName CHAR,
NameLength2 SMALLINT,
TableName CHAR,
NameLength3 SMALLINT,
TableType CHAR,
NameLength4 SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION

The Tables() function retrieves information on tables available through the connection on which StatementHandle was allocated. The function has the following parameters:

StatementHandle (input)
Statement handle.

CatalogName (input)
Buffer that may qualify the result set by catalog name. If the SQL_ATTR_METADATA_ID attribute of StatementHandle is SQL_TRUE, then this is interpreted as an identifier (see Treatment as Identifier on page 54); otherwise, it is a pattern-value (see Qualification by Pattern-values on page 55).

NameLength1 (input)
Length in octets of CatalogName.

SchemaName (input)
Buffer that may qualify the result set by schema name. If the SQL_ATTR_METADATA_ID attribute of StatementHandle is SQL_TRUE, then this is interpreted as an identifier (see Treatment as Identifier on page 54); otherwise, it is a pattern-value (see Qualification by Pattern-values on page 55).

NameLength2 (input)
Length in octets of SchemaName.

TableName (input)
Buffer that may qualify the result set by table name. If the SQL_ATTR_METADATA_ID attribute of StatementHandle is SQL_TRUE, then this is interpreted as an identifier (see Treatment as Identifier on page 54); otherwise, it is a pattern-value (see Qualification by Pattern-values on page 55).

NameLength3 (input)
Length in octets of TableName.

TableType (input)
Buffer that may contain a value list (see Qualification by Value List on page 55) to qualify the result set by table type. Valid table types include BASE TABLE, VIEW and implementation-defined table types.

NameLength4 (input)
Length in octets of TableType.
Table information is returned in the form of a result set on the statement handle where each table is represented by one row of the result set. It is implementation-defined whether or not a user has SELECT privileges to any of these tables. The result set returned by `Tables()` has at least these columns in the following order:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_CAT</td>
<td>VARCHAR(128)</td>
<td>The name of the catalog containing TABLE_SCHEMA. See APPLICATION USAGE below.</td>
</tr>
<tr>
<td>TABLE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>The name of the schema containing TABLE_NAME.</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>VARCHAR(128)</td>
<td>The name of the table or view.</td>
</tr>
<tr>
<td>TABLE_TYPE</td>
<td>VARCHAR(254)</td>
<td>Identifies the type of the table or view. It can have the values 'BASE TABLE' or 'VIEW' or other implementation-defined values.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR(254)</td>
<td>May contain descriptive information about the table.</td>
</tr>
</tbody>
</table>

**RETURN VALUE**

- [SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE].
- [SQL_SUCCESS_WITH_INFO] can result when implementation-defined warnings are produced.

**DIAGNOSTICS**

There must not be an open cursor on `StatementHandle ('24000')`.

If the implementation does not support qualification by catalog name, then the value of `CatalogName` must be a null pointer or a zero-length string ('HYC00').

**APPLICATION USAGE**

In languages that allow the use of pointers, the application can provide a null pointer for any of `CatalogName`, `SchemaName`, `TableName` and `TableType`, in order to inhibit qualification of the result set on the respective basis.

The `TABLE_CAT` column may be null or may contain an empty string for several reasons discussed in Section 2.4.3 on page 22. If `TABLE_CAT` contains an empty string, the application should take care to not append '.' when forming a fully-qualified object name for use in an SQL statement.

**EXAMPLE USAGE (C)**

```c
#include <sqlcli.h>
SQLRETURN ReturnCode;
SQLHSTMT StatementHandle;
SQLCHAR *CatalogName;
SQLSMALLINT NameLength1;
SQLCHAR *SchemaName;
SQLSMALLINT NameLength2;
SQLCHAR *TableName;
SQLSMALLINT NameLength3;
SQLCHAR *TableType;
```
SQLSMALLINT NameLength4;

...;

ReturnCode = SQLTables(StatementHandle,
    CatalogName, NameLength1,
    SchemaName, NameLength2,
    TableName, NameLength3,
    TableType, NameLength4);

EXAMPLE USAGE (COBOL)

  01 STATEMENTHANDLE PICTURE S9(9) BINARY.
  01 CATALOGNAME PICTURE X(128).
  01 NAMELENGTH1 PICTURE S9(4) BINARY.
  01 SCHEMANAME PICTURE X(128).
  01 NAMELENGTH2 PICTURE S9(4) BINARY.
  01 TABLENAME PICTURE X(128).
  01 NAMELENGTH3 PICTURE S9(4) BINARY.
  01 TABLETYPE PICTURE X(128).
  01 NAMELENGTH4 PICTURE S9(4) BINARY.
  01 RETURNCODE PICTURE S9(4) BINARY.

   ...;

   CALL "SQLRTables" USING STATEMENTHANDLE, CATALOGNAME, NAMELENGTH1,
      SCHEMANAME, NAMELENGTH2, TABLENAME, NAMELENGTH3, TABLETYPE,
      NAMELENGTH4, RETURNCODE.

RELATION TO STANDARDS

This function is not present in the ISO CLI Draft International Standard.
NAME
Transact — Commit or roll back a transaction (deprecated)

SYNOPSIS
FUNCTION Transact
( EnvironmentHandle INTEGER,
  ConnectionHandle INTEGER,
  CompletionType SMALLINT)
RETURNS (SMALLINT)

DESCRIPTION
The Transact() function completes (commits or rolls back) a transaction, enacting or undoing any changes to the database performed on any connection with the specified handle since the transaction began (see Section 4.9 on page 63).

The function has the following parameters:
EnvironmentHandle (input)
  Environment handle.
ConnectionHandle (input)
  Connection handle.
CompletionType (input)
  The desired completion outcome. This may be one of the following:
  SQL_COMMIT
    Work done is to be committed.
  SQL_ROLLBACK
    Work done is to be rolled back.

To complete a transaction on a single connection, the application passes a valid connection handle as ConnectionHandle. To complete a transaction that spans multiple connections, the application sets ConnectionHandle to SQL_NULL_HDBC and passes a valid environment handle as EnvironmentHandle.

Successful completion of a transaction closes any open cursors.

It is implementation-defined whether a transaction can span connections. On an implementation where transactions can span connections, the following aspects of multiple-connection transactions are implementation-defined:

• the sequence in which the connections have their transactions completed
• the effect that the success or failure of commitment over one connection has on the success or failure over any other.

If the work at some connections is committed and the work at others is rolled back, administrative action is typically necessary to assess or restore the integrity of the database.

The effect of calling Transact() is implementation-defined if a valid connection handle is specified as ConnectionHandle (see Effects across Connections on page 63).

RETURN VALUE
[SQL_SUCCESS], [SQL_ERROR], [SQL_SUCCESS_WITH_INFO] or [SQL_INVALID_HANDLE]. The [SQL_SUCCESS_WITH_INFO] outcome occurs if implementation-defined warnings are produced.
**Diagnostics**

The `CompletionType` argument must be one of the values listed above ("HY012").

The function may fail because the connection fails ("08007"). In this case, the application might not be able to determine whether the requested transaction outcome took effect before the failure.

The server can decide to roll back the transaction even though the call to `Transact()` specified SQL_COMMIT. (The SQLSTATE class is ‘40’ and the subclass is one of those specified in Appendix A.)

**Errors for Global Transaction Control**

The `Transact()` function may fail for any of the following reasons:

- The implementation is not able to guarantee that all work in the global transaction can be completed atomically. Under implementation-defined criteria, either the transaction is still active ("25S02") or all work in the transaction active in the specified handle is rolled back ("25S03").

- One or more of the connections in the specified handle fails to complete the transaction with the outcome the application specified ("25S01"). The transaction state is unknown.

The application must not complete by calling `Transact()` a global transaction it started by calling a function from an explicit transaction demarcation API, such as the X/Open Distributed Transaction Processing (DTP) function `tx_begin()` ("2D000").

**Application Usage**

X/Open Distributed Transaction Processing (DTP) documents describe methods of coordinating databases so that changes to any recoverable resources are committed or rolled back atomically. For example, the emerging Transaction Demarcation API specifies functions by which an application declares the start, commitment or rollback of a global transaction. An application may use these functions in place of the implicit start of transactions described above, and in place of `Transact()`, in order to coordinate SQL work with non-SQL work.

**Example Usage (C)**

```c
#include <sqlcl.h>
SQLRETURN ReturnCode;
SQLHENV EnvironmentHandle;
SQLHDBC ConnectionHandle;
SQLSMALLINT CompletionType;
...
ReturnCode = SQLTransact(EnvironmentHandle, ConnectionHandle, CompletionType);
```

**Example Usage (COBOL)**

```cobol
01 ENVIRONMENTHANDLE PICTURE S9(9) BINARY.
01 CONNECTIONHANDLE PICTURE S9(9) BINARY.
01 COMPLETIONTYPE PICTURE S9(4) BINARY.
01 RETURNCODE PICTURE S9(4) BINARY.
...
CALL "SQLRTransact" USING ENVIRONMENTHANDLE, CONNECTIONHANDLE, COMPLETIONTYPE, RETURNCODE.
```

**Relation to Standards**

This function is not present in the ISO CLI Draft International Standard, but the preferred equivalent, `EndTran()`, is. However, the ability to specify a connection handle represents an
extension to the ISO CLI Draft International Standard, which only permits `EndTran()` to be applied to the entire environment.

**SEE ALSO**

`EndTran()`
This appendix lists the Sqlstate diagnostic codes that CLI functions can return. Notes on the various columns are as follows:

**Description**

The wordings in the descriptions are not normative or visible to the application. They are generally based on the ISO SQL standard but sometimes have been adapted to use X/Open terminology. A change in wording should not be construed as defining a different diagnostic from the ISO SQL standard.

**Reference**

The Reference column is meant to be a complete list of CLI functions that return a given code, for the benefit of application writers; except that for codes returned by many CLI functions, the column gives a section reference within this document instead.

Many of the diagnostics that reference Prepare() can instead be raised by Execute() in some implementations; see Prepare() on page 194 for details.

The following codes occur in the Reference column:

[E] The code can be returned by any function that executes an SQL statement: ExecDirect(), Execute() and ParamData().

[F] The code can be returned by any function that fetches a row of a result set: Fetch() and FetchScroll().
<table>
<thead>
<tr>
<th>Sqlstate</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
</table>
| '00000' | **Success**  
Not used in CLI; function returns [SQL_SUCCESS]. | Section 5.1 on page 67 |
| '01002' | **Success with warning**  
— Disconnect error. | Section 5.4.10 on page 76, Disconnect() |
| '01004' | — String data, right truncation. | Section 5.4.10 on page 76, ColAttribute(), DescribeCol(), [E], [F], GetConnectAttr(), GetCursorName(), GetData(), GetDescField(), GetDescRec(), GetEnvAttr(), GetInfo(), GetStmtAttr() |
| '01006' | — Privilege not revoked.  
— Privilege not granted. | [E] |
| '02000' | **No data**  
Not used in CLI; function returns [SQL_NO_DATA]. | Section 5.1 on page 67 |
| '07001' | **Dynamic SQL error**  
— Parameter descriptor does not match parameter(s) in quantity or type. | [E], PutData() |
| '07002' | — COUNT field incorrect. | [F] |
| '07005' | — Prepared statement not a cursor-specification | ColAttribute(), DescribeCol() |
| '07006' | — Restricted data type attribute violation. | Section 5.4.10 on page 76, BindParam(), [E], [F], GetData() |
| '07008' | — Invalid descriptor count. | [E], [F], GetData() |
| '07009' | — Invalid descriptor index. | BindCol(), BindParam(), DescribeCol(), GetData(), GetDescField(), GetDescRec(), SetDescField(), SetDescRec() |
| '08001' | **Connection exception**  
— Client unable to establish connection. | Connect() |
<p>| '08002' | — Connection name in use. | Connect() |
| '08003' | — Connection does not exist. | Section 5.4.1 on page 73, AllocHandle(), FreeStmt(), GetConnectAttr() |
| '08004' | — Server rejected the connection. | Connect() |
| '08006' | — Connection failure. | Section 5.4.1 on page 73 |
| '08007' | — Transaction state unknown. | Section 5.4.1 on page 73, EndTran(), Transact() |</p>
<table>
<thead>
<tr>
<th>Sqlstate</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0A001'</td>
<td>Feature not supported — Multiple-server transaction.</td>
<td>Section 5.4.2 on page 73, Section 5.4.4 on page 74, Connect()</td>
</tr>
<tr>
<td>'21S01'</td>
<td>Cardinality violation — Insert value list does not match column list.</td>
<td>ExecDirect(), Prepare()</td>
</tr>
<tr>
<td>'21S02'</td>
<td>Cardinalsity violation — Degree of derived table does not match column list.</td>
<td>ExecDirect(), Prepare()</td>
</tr>
<tr>
<td>'22001'</td>
<td>Data exception — String data, right truncation.</td>
<td>Section 5.4.10 on page 76</td>
</tr>
<tr>
<td>'22002'</td>
<td>— Indicator variable required but not supplied.</td>
<td>[F], GetData()</td>
</tr>
<tr>
<td>'22003'</td>
<td>— Numeric value out of range.</td>
<td>Section 5.4.10 on page 76</td>
</tr>
<tr>
<td>'22005'</td>
<td>— Error in length assignment</td>
<td>ExecDirect(), Execute()</td>
</tr>
<tr>
<td>'22007'</td>
<td>— Invalid date/time format.</td>
<td>PutData()</td>
</tr>
<tr>
<td>'22012'</td>
<td>— Division by zero.</td>
<td>[E], [F]</td>
</tr>
<tr>
<td>'22018'</td>
<td>— Invalid character value.</td>
<td>PutData()</td>
</tr>
<tr>
<td>'23000'</td>
<td>Integrity constraint violation</td>
<td>[E]</td>
</tr>
<tr>
<td>'24000'</td>
<td>Invalid cursor state</td>
<td>Columns(), [E], [F], GetTypeInfo(), Prepare(), SpecialColumns(),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Statistics(), Tables()</td>
</tr>
<tr>
<td>'25000'</td>
<td>Invalid transaction state</td>
<td>Disconnect(), Section 5.4.4 on page 74</td>
</tr>
<tr>
<td>'25S01'</td>
<td>— Transaction state unknown.</td>
<td>EndTran(), Transact()</td>
</tr>
<tr>
<td>'25S02'</td>
<td>— Transaction is still active.</td>
<td>EndTran(), Transact()</td>
</tr>
<tr>
<td>'25S03'</td>
<td>— Transaction is rolled back.</td>
<td>EndTran(), Transact()</td>
</tr>
<tr>
<td>'28000'</td>
<td>Invalid authorisation specification</td>
<td>Connect()</td>
</tr>
<tr>
<td>'2D000'</td>
<td>Invalid transaction termination</td>
<td>EndTran(), Transact()</td>
</tr>
<tr>
<td>'34000'</td>
<td>Invalid cursor name</td>
<td>ExecDirect(), Prepare(),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SetCursorName()</td>
</tr>
<tr>
<td>'40000'</td>
<td>Transaction rollback</td>
<td>Section 5.4.4 on page 74, [F],</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EndTran(), Transact()</td>
</tr>
<tr>
<td>'40001'</td>
<td>— Serialisation failure.</td>
<td>[F]</td>
</tr>
<tr>
<td>'40003'</td>
<td>— Statement completion unknown.</td>
<td>Section 5.4.1 on page 73</td>
</tr>
<tr>
<td>'40500'</td>
<td>— Implementation-defined.</td>
<td>Section 5.4.4 on page 74</td>
</tr>
<tr>
<td>'409ZZ'</td>
<td></td>
<td>Section 5.4.4 on page 74</td>
</tr>
<tr>
<td>'40I00'</td>
<td></td>
<td>Section 5.4.4 on page 74</td>
</tr>
<tr>
<td>'40ZZZ'</td>
<td></td>
<td>Section 5.4.4 on page 74</td>
</tr>
<tr>
<td>Sqlstate</td>
<td>Description</td>
<td>Reference</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>'42000'</td>
<td>Syntax error or access violation</td>
<td>ExecDirect(), Prepare()</td>
</tr>
<tr>
<td>'42S01'</td>
<td>Base table or view already exists.*</td>
<td>ExecDirect(), Prepare()</td>
</tr>
<tr>
<td>'42S02'</td>
<td>Base table or view not found.*</td>
<td>ExecDirect(), Prepare()</td>
</tr>
<tr>
<td>'42S11'</td>
<td>Index already exists.*</td>
<td>ExecDirect(), Prepare()</td>
</tr>
<tr>
<td>'42S12'</td>
<td>Index not found.*</td>
<td>ExecDirect(), Prepare()</td>
</tr>
<tr>
<td>'42S21'</td>
<td>Column already exists.*</td>
<td>ExecDirect(), Prepare()</td>
</tr>
<tr>
<td>'42S22'</td>
<td>Column not found.*</td>
<td>ExecDirect(), Prepare()</td>
</tr>
</tbody>
</table>

**CLI-specific error**

<table>
<thead>
<tr>
<th>Sqlstate</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>'HY001'</td>
<td>Memory allocation error.</td>
<td>Section 5.4.5 on page 74</td>
</tr>
<tr>
<td>'HY003'</td>
<td>Invalid data type in application descriptor.</td>
<td>BindCol(), BindParam(), GetData()</td>
</tr>
<tr>
<td>'HY004'</td>
<td>Invalid data type (in other context).</td>
<td>BindParam(), GetTypeInfo()</td>
</tr>
<tr>
<td>'HY007'</td>
<td>Associated statement is not prepared.</td>
<td>GetDescField(), GetDescRec()</td>
</tr>
<tr>
<td>'HY008'</td>
<td>Operation cancelled.</td>
<td>Section 5.4.6 on page 75</td>
</tr>
<tr>
<td>'HY009'</td>
<td>Invalid use of null pointer.</td>
<td>Section 5.4.8 on page 75</td>
</tr>
<tr>
<td>'HY010'</td>
<td>Function sequence error.</td>
<td>Appendix B</td>
</tr>
<tr>
<td>'HY011'</td>
<td>Attribute cannot be set now.</td>
<td>Section 5.4.3 on page 73, SetEnvAttr(), SetStmtAttr(), SetStmtOption()</td>
</tr>
<tr>
<td>'HY012'</td>
<td>Invalid transaction operation code.</td>
<td>EndTran(), Transact()</td>
</tr>
<tr>
<td>'HY013'</td>
<td>Memory management error.</td>
<td>Section 5.4.5 on page 74</td>
</tr>
<tr>
<td>'HY014'</td>
<td>Limit on number of handles exceeded.</td>
<td>Alloc*()</td>
</tr>
<tr>
<td>'HY015'</td>
<td>No cursor name available.</td>
<td>GetCursorName()</td>
</tr>
<tr>
<td>'HY016'</td>
<td>Cannot modify an implementation row descriptor.</td>
<td>CopyDesc(), SetDescField(), SetDescRec()</td>
</tr>
<tr>
<td>'HY017'</td>
<td>Invalid use of an automatically-allocated descriptor handle.</td>
<td>FreeHandle(), SetStmtAttr(), SetStmtOption()</td>
</tr>
<tr>
<td>'HY018'</td>
<td>Server declines cancel request.</td>
<td>Cancel()</td>
</tr>
<tr>
<td>'HY019'</td>
<td>Non-character data sent in pieces.</td>
<td>PutData()</td>
</tr>
<tr>
<td>'HY020'</td>
<td>Attempt to concatenate a null value.</td>
<td>PutData()</td>
</tr>
<tr>
<td>'HY021'</td>
<td>Inconsistent descriptor information.</td>
<td>Section 5.4.9 on page 75</td>
</tr>
</tbody>
</table>

* Implementations may raise '42000' instead of this code if, for example, the application is not entitled to obtain this metadata. The codes '42Snn' are 'S00nn' in the X/Open SQL specification.
<table>
<thead>
<tr>
<th>Sqlstate</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘HY023’</td>
<td>CLI-specific error (continued)</td>
<td><code>BindCol</code>, <code>BindParam</code></td>
</tr>
<tr>
<td>‘HY024’</td>
<td>— Descriptor invalid on indirect reference.</td>
<td><code>SetConnectAttr</code>, <code>SetConnectOption</code>, <code>SetEnvAttr</code>, <code>SetStmtAttr</code>, <code>SetStmtOption</code></td>
</tr>
<tr>
<td>‘HY090’</td>
<td>— Invalid attribute value.</td>
<td><code>Section 5.4.7 on page 75, Connect, GetData</code></td>
</tr>
<tr>
<td>‘HY091’</td>
<td>— Invalid string length or buffer length.</td>
<td><code>ColAttribute</code>, <code>GetDescField</code>, <code>SetDescField</code></td>
</tr>
<tr>
<td>‘HY092’</td>
<td>— Invalid attribute/option identifier.</td>
<td><code>AllocHandle</code>, <code>EndTran</code>, <code>FreeHandle</code>, <code>FreeStmt</code>, <code>Get*Attr</code>, <code>Get*Option</code>, <code>Set*Attr</code>, <code>Set*Option</code>, <code>SpecialColumns</code>, <code>Statistics</code></td>
</tr>
<tr>
<td>‘HY096’</td>
<td>— Invalid information type.</td>
<td><code>GetInfo</code></td>
</tr>
<tr>
<td>‘HY103’</td>
<td>— Invalid retrieval code.</td>
<td><code>DataSources</code></td>
</tr>
<tr>
<td>‘HY104’</td>
<td>— Invalid <code>LengthPrecision</code> value.</td>
<td><code>BindParam</code></td>
</tr>
<tr>
<td>‘HY106’</td>
<td>— Invalid fetch orientation.</td>
<td><code>FetchScroll</code></td>
</tr>
<tr>
<td>‘HY500’</td>
<td>— Implementation-defined.</td>
<td><code>All</code></td>
</tr>
<tr>
<td>‘HY9ZZ’</td>
<td>— Implementation-defined.</td>
<td><code>Section 5.4.10 on page 76, Columns, GetStmtAttr, GetStmtOption, SetStmtAttr, SetStmtOption, SpecialColumns, Statistics, Tables</code></td>
</tr>
<tr>
<td>‘HYC00’</td>
<td>— Optional feature not implemented.</td>
<td><code>All</code></td>
</tr>
<tr>
<td>‘HY100’</td>
<td>— Implementation-defined.</td>
<td><code>All</code></td>
</tr>
<tr>
<td>‘HYZZZ’</td>
<td>— Implementation-defined.</td>
<td><code>All</code></td>
</tr>
</tbody>
</table>
The two tables in this appendix show the effect of each CLI function on the states of the various CLI handles. Table B-1 on page 241 describes valid state transitions for environment handles and connection handles. Table B-2 on page 242 describes legal state transitions for statement handles.

**Interpretation of the Tables**

An entry under a particular state in the table asserts that it is not a sequencing error to call the CLI function from that state. (Calling the CLI function may produce some other error, as described on the appropriate reference manual page.) If the call is successful, the state table entry shows the resulting state.

**Invalid State Errors**

A blank entry asserts that it is a sequencing error to call the CLI function in that state. The function sets SQLSTATE to ‘HY010’ *(Function sequence error)*; unless a more explicit error code applies, such as ‘24000’ *(Invalid cursor state)*. These cases are described on the reference manual pages and the SQLSTATE value appears in Table B-2 on page 242.

A state table error could be caused by passing to a CLI function an invalid handle — a null or unallocated handle or a handle of the wrong type. In these cases, the function returns [SQL_INVALID_HANDLE]. Corresponding state table columns (describing attempted operations on an unallocated handle) have the legend INV_H.

`Cancel()` is not included in the tables because it does not cause any state transition. Its only effect is to cause the function active at the time `Cancel()` was called to return an error if the operation was cancelled. See `Cancel()` on page 105 for details.

**Notation**

The tables describe input to the CLI function in parentheses, even though that may not be the exact syntax used. The tables denote output from the routine, including return status, using an arrow (→) followed by the specific output.

A general state table entry (one that does not show inputs or outputs) describes all remaining cases of calls to that routine. These general entries assume the routine returns success. Calls that return failure do not make state transitions, except where described by specific state table entries.

The boldfaced headings of some state table columns, such as `prepared`, are referenced elsewhere in this specification; but the wording of these column headings is not normative.
B.1 Environment State Transitions

A CLI environment can be in one of only two states: allocated and unallocated.

In the unallocated state, the only valid function on the environment is `AllocHandle()` (which changes the environment's state to allocated).\(^{25}\)

In the allocated state, the application can call `FreeHandle()`, `GetEnvAttr()` and `SetEnvAttr()` on the environment. None of these changes the state of the environment except that calling `FreeHandle()` changes its state to unallocated.

In the allocated state, the application can also allocate connections, as described in Section B.2 on page 241.

\(^{25}\) As described on the reference manual page, certain calls to `AllocHandle()` return a restricted handle that the application can use only to obtain diagnostic information. The restricted handle is not a separate state of the environment handle, since invalid uses of the restricted handle return [SQL_INVALID_HANDLE] rather than the function sequence error 'HY010'.

Each connection handle can be in one of the following states:

- C₀ Unallocated.
- C₁ Allocated.
- C₂ Allocated and connected to a database.

The initial state is C₀.

### Table B-1 State Table for Connection Handles

<table>
<thead>
<tr>
<th>Function</th>
<th>C₀</th>
<th>C₁</th>
<th>C₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocHandle (Connection)</td>
<td>C₁</td>
<td>INV_H</td>
<td>C₁ [1]</td>
</tr>
<tr>
<td>GetConnectAttr(),</td>
<td>INV_H</td>
<td>C₁ [1]</td>
<td>C₂</td>
</tr>
<tr>
<td>GetConnectOption(),</td>
<td>INV_H</td>
<td>C₁ [1]</td>
<td>C₂</td>
</tr>
<tr>
<td>GetInfo(),</td>
<td>INV_H</td>
<td>C₁ [1]</td>
<td>C₂</td>
</tr>
<tr>
<td>SetConnectAttr(),</td>
<td>INV_H</td>
<td>C₁ [1]</td>
<td>C₂</td>
</tr>
<tr>
<td>SetConnectOption()</td>
<td>INV_H</td>
<td>C₁ [1]</td>
<td>C₂</td>
</tr>
<tr>
<td>DataSources(),</td>
<td>INV_H</td>
<td>C₁ [1]</td>
<td>C₂</td>
</tr>
<tr>
<td>GetEnvAttr(),</td>
<td>INV_H</td>
<td>C₁ [1]</td>
<td>C₂</td>
</tr>
<tr>
<td>GetFunctions()</td>
<td>INV_H</td>
<td>C₁ [1]</td>
<td>C₂</td>
</tr>
<tr>
<td>Connect()</td>
<td>INV_H</td>
<td>C₁ [1]</td>
<td>C₂</td>
</tr>
<tr>
<td>AllocHandle (Descriptor),</td>
<td>INV_H</td>
<td>'08003’</td>
<td>C₂</td>
</tr>
<tr>
<td>AllocHandle (Statement)</td>
<td>INV_H</td>
<td>'08003’</td>
<td>C₂</td>
</tr>
<tr>
<td>Disconnect()</td>
<td>INV_H</td>
<td>'08003’</td>
<td>C₁</td>
</tr>
<tr>
<td>FreeHandle (Connection)</td>
<td>INV_H</td>
<td>C₀</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

[1] If the operation (getting or setting the specified connection attribute, or getting the specified GetInfo() item) requires an existing connection, then a call from this connection state raises '08003'.

When a connection is in state C₂, the application can allocate statement handles whose states are governed by Table B-2 on page 242, and can allocate descriptor handles whose states are discussed in Section B.4 on page 245.
### B.3 Statement Transitions

A statement handle (of type SQLHSTMT) can be in one of the following states:

- **S₀** Not allocated.
- **S₁** Allocated.
- **S₂** Prepared (whether or not the dynamic parameters are set and columns are bound).
- **S₃** Executed, or cursor open but not positioned on a row.
- **S₄** Cursor positioned on a row.

**S₅** through **S₇** refer to the data-at-execute dialogue and jump to the state table in Section B.3.1 on page 244.

All statement handles are initially in state **S₀**.

The numbers in [ ] refer to the notes following the table.

<table>
<thead>
<tr>
<th>Function</th>
<th>not allocated</th>
<th>allocated</th>
<th>prepared</th>
<th>executed</th>
<th>cursor positioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocHandle(Statement) [1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetCursorName()</td>
<td>INV_H</td>
<td>S₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetCursorName()</td>
<td>INV_H</td>
<td>S₁</td>
<td>S₂</td>
<td>S₃</td>
<td>S₄</td>
</tr>
<tr>
<td>BindCol(), BindParam()</td>
<td>INV_H</td>
<td>S₁</td>
<td>S₂</td>
<td>S₃</td>
<td>S₄</td>
</tr>
<tr>
<td>GetStmtAttr(), GetStmtOption(),</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetParam(), SetStmtAttr(),</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetStmtOption()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ColAttribute(), DescribeCol(),</td>
<td>INV_H</td>
<td>S₂</td>
<td>S₃</td>
<td></td>
<td>'24000'</td>
</tr>
<tr>
<td>NumResultCols()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RowCount()</td>
<td></td>
<td></td>
<td></td>
<td>S₃</td>
<td>S₄</td>
</tr>
<tr>
<td>Fetch(), FetchScroll()</td>
<td>INV_H</td>
<td>S₄</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetData()</td>
<td>INV_H</td>
<td></td>
<td></td>
<td>S₄</td>
<td></td>
</tr>
<tr>
<td>CloseCursor()</td>
<td>INV_H</td>
<td>S₁</td>
<td>S₂</td>
<td>S₂[3]</td>
<td></td>
</tr>
<tr>
<td>FreeHandle(Statement)</td>
<td>INV_H</td>
<td>S₀</td>
<td>S₀</td>
<td>S₀</td>
<td>S₀</td>
</tr>
<tr>
<td>EndTran(), Transact() [6]</td>
<td>S₀</td>
<td>S₁</td>
<td>S₁</td>
<td>S₁</td>
<td></td>
</tr>
<tr>
<td>Disconnect() [6]</td>
<td>S₀</td>
<td>S₀</td>
<td>'25000'</td>
<td>'25000'</td>
<td>'25000'</td>
</tr>
</tbody>
</table>

| Table B-2  | State Table for Statement Handles |

X/Open CAE Specification
Notes:

[1] The connection handle that `AllocHandle()` references must be in state $C_2'$; see Table B-1 on page 241.

[2] For WHERE CURRENT OF `cursor`, the separate statement handle that was used to open `cursor` must be in state $S_3$ or $S_4$ for `Prepare()`. After `Prepare()`, that statement handle remains in the same state.

[3] In state $S_3'$, a statement may be reprepared, and a CLI function that returns a result set [4] can be called, only if there are no open cursors (that is, only if any `cursor-specification` or CLI function that returns a result set [4] on the statement handle has been followed by a `CloseCursor()`).

[4] Result-set functions include the metadata functions described in Section 4.6 on page 53 and `GetTypeInfo()`.

[5] For WHERE CURRENT OF `cursor`, the separate statement handle that was used to open `cursor` must be in state $S_4'$. After `Execute()` or `ExecDirect()`, that statement handle remains in state $S_4'$.

[6] These functions do not explicitly specify a statement handle. The entry for `EndTran()` and `Transact()` show state transitions for all statement handles allocated in the scope (environment or connection) specified in the call. The entry for `Disconnect()` (which specifies a connection) shows state transitions for all statement handles allocated on that connection.
B.3.1 Data-at-Execute Dialogue

An application may set an application parameter descriptor to declare that it will pass the actual data for one or more dynamic parameters at execute time. When an application calls `ExecDirect()` and `Execute()` and there is at least one dynamic parameter that needs data, the data-at-execute dialogue begins. (See Section 4.3.2 on page 48 for an overview.) Table B-2 on page 242 illustrates these cases by showing the return value → [SQL_NEED_DATA] and the resulting state S5.

The following states are involved in the data-at-execute dialogue:

S5 The application is due to call `ParamData()` to determine the identity of the first dynamic parameter for which data is needed.

S6 The application is due to call `PutData()` to supply the first part (or all of) a dynamic argument.

S7 The application has called `PutData()` at least once for the current dynamic parameter.

The initial state in this table is S5.

The numbers in [] refer to the notes following the table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Statement Handle States</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S5</td>
</tr>
<tr>
<td><code>ParamData()</code> → [SQL_NEED_DATA]</td>
<td><code>ParamData()</code> providing parameter needing data</td>
</tr>
<tr>
<td><code>ParamData()</code></td>
<td></td>
</tr>
<tr>
<td><code>PutData()</code></td>
<td>provide first piece</td>
</tr>
<tr>
<td><code>Cancel()</code></td>
<td>other calls to <code>PutData()</code></td>
</tr>
<tr>
<td><code>Disconnect()</code></td>
<td></td>
</tr>
</tbody>
</table>

Table B-3 State Table for Statement Handles (Data-at-Execute Dialogue)

Notes:

[7] Not applicable. When called in S5 (that is, just after `ExecDirect()` or `Execute()` returned [SQL_NEED_DATA]), `ParamData()` always returns [SQL_NEED_DATA].

[8] This line illustrates the use of `Cancel()`, typically by the same application, to cancel the data-at-execute dialogue. The statement handle state reverts to the state from which it entered this table: The resulting state is S1 (allocated) if it was `ExecDirect()` that originally returned [SQL_NEED_DATA], or S2 (prepared) if it was `Execute()`.

[9] This function does not explicitly specify a statement handle. This entry in the table illustrates that it is a state error to disconnect a connection on which there is any statement handle involved in the data-at-execute dialogue.
B.4 Descriptor State Transitions

A descriptor can be in one of only two states: allocated and unallocated.

In the unallocated state, the only valid function on the descriptor is `AllocHandle()` (which changes the descriptor's state to allocated).

In the allocated state, the application can call `CopyDesc()`, `FreeHandle()`, `GetDescField()`, `GetDescRec()`, `SetDescField()` and `SetDescRec()` on the descriptor. None of these changes the state of the descriptor except that calling `FreeHandle()` changes its state to unallocated.

Calling `Disconnect()` frees (changes to the unallocated state) all descriptor handles allocated on the connection.
This appendix contains the source of a C-language header file and a COBOL header file.

### C.1 C-language File <sqlcli.h>

```c
/* sqlcli.h Header File for SQL CLI.
 * The actual header file must contain at least the information
 * specified here, except that the comments may vary.
 */
/* API declaration data types */
typedef unsigned char SQLCHAR;
typedef long SQLINTEGER;
typedef short SQLSMALLINT;
typedef double SQLDOUBLE;
typedef double SQLFLOAT;
typedef float SQLREAL;
typedef void * SQLPOINTER;
typedef unsigned char SQLDATE;
typedef unsigned char SQLTIME;
typedef unsigned char SQLTIMESTAMP;
typedef unsigned char SQLVARCHAR;
typedef unsigned char SQLDECIMAL;
typedef unsigned char SQLNUMERIC;
/* function return type */
typedef SQLSMALLINT SQLRETURN;
/* generic data structures */
typedef SQLINTEGER SQLHENV; /* environment handle */
typedef SQLINTEGER SQLHDBC; /* connection handle */
typedef SQLINTEGER SQLHSTMT; /* statement handle */
typedef SQLINTEGER SQLHDESC; /* descriptor handle */
/* special length/indicator values */
 /* The value of SQL_NULL_DATA, despite being a #define, should
 * not be changed; it was chosen for compatibility with the
 * ISO SQL definition of the value returned to indicate a null value.
 */
#define SQL_NULL_DATA -1
#define SQL_DATA_AT_EXEC -2
/* return values from functions */
 /* These codes, despite being #defines, should not be changed;
 * they were chosen for compatibility with the ISO SQL definition
 * of SQLCODE. */
#define SQL_SUCCESS 0
#define SQL_SUCCESS_WITH_INFO 1
#define SQL_NO_DATA 100
#define SQL_ERROR -1
#define SQL_INVALID_HANDLE -2
#define SQL_NEED_DATA 99
```

Data Management: SQL Call Level Interface (CLI) 247
/* test for SQL_SUCCESS or SQL_SUCCESS_WITH_INFO */
define SQL_SUCCEEDED(rc) (((rc)&(~1))==0)

/* flags for null-terminated string */
define SQL_NTS -3
#define SQL_NTSL -3L

/* maximum message length */
#define SQL_MAX_MESSAGE_LENGTH 512

/* maximum identifier length */
define SQL_MAX_ID_LENGTH 18

/* date/time length constants */
define SQL_DATE_LEN 10
#define SQL_TIME_LEN 8 /* add P+1 if precision is nonzero */
define SQL_TIMESTAMP_LEN 19 /* add P+1 if precision is nonzero */

/* handle type identifiers */
define SQL_HANDLE_ENV 1
#define SQL_HANDLE_DBC 2
#define SQL_HANDLE_STMT 3
#define SQL_HANDLE_DESC 4

/* environment attribute */
define SQL_ATTR_OUTPUT_NTS 10001

/* connection attribute */
define SQL_ATTR_AUTO_IPD 10001

/* connection and schema attributes */
define SQL_ATTR_METADATA_ID 10014

/* statement attributes */
define SQL_ATTR_APP_ROW_DESC 10010
#define SQL_ATTR_APP_PARAM_DESC 10011
#define SQL_ATTR_IMP_ROW_DESC 10012
#define SQL_ATTR_IMP_PARAM_DESC 10013
#define SQL_ATTR_CURSOR_SCROLLABLE -1
#define SQL_ATTR_CURSOR_SENSITIVITY -2

/* identifiers of fields in the SQL descriptor */
define SQL_DESC_COUNT 1001
#define SQL_DESC_TYPE 1002
#define SQL_DESC_LENGTH 1003
#define SQL_DESC_OCTET_LENGTH_PTR 1004
#define SQL_DESC_PRECISION 1005
#define SQL_DESC_SCALE 1006
#define SQL_DESC_DATETIME_INTERVAL_CODE 1007
#define SQL_DESC_NULLABLE 1008
#define SQL_DESC_INDICATOR_PTR 1009
#define SQL_DESC_DATA_PTR 1010
#define SQL_DESC_NAME 1011
#define SQL_DESC_UNNAMED 1012
#define SQL_DESC_OCTET_LENGTH 1013
#define SQL_DESC_ALLOC_TYPE 1099

/* identifiers of fields in the diagnostics area */
define SQL_DIAG_RETURNCODE 1
#define SQL_DIAG_NUMBER 2
#define SQL_DIAG_ROW_COUNT 3
#define SQL_DIAG_SQLSTATE 4
#define SQL_DIAG_NATIVE 5
#define SQL_DIAG_MESSAGE_TEXT 6
#define SQL_DIAG_DYNAMIC_FUNCTION 7
#define SQL_DIAG_CLASS_ORIGIN 8
#define SQL_DIAG_SUBCLASS_ORIGIN 9
#define SQL_DIAG_CONNECTION_NAME 10
#define SQL_DIAG_SERVER_NAME 11
#define SQL_DIAG_DYNAMIC_FUNCTION_CODE 12

/* dynamic function codes
 * These codes, despite being #defines, should not be changed;
 * they were chosen for compatibility with the ISO SQL definition
 * of dynamic function codes.
 */
#define SQL_DIAG_ALTER_TABLE 4
#define SQL_DIAG_CREATE_INDEX -1
#define SQL_DIAG_CREATE_TABLE 77
#define SQL_DIAG_CREATE_VIEW 84
#define SQL_DIAG_DELETE_WHERE 19
#define SQL_DIAG_DROP_INDEX -2
#define SQL_DIAG_DROP_TABLE 32
#define SQL_DIAG_DROP_VIEW 36
#define SQL_DIAG_DYNAMIC_DELETE_CURSOR 38
#define SQL_DIAG_DYNAMIC_UPDATE_CURSOR 81
#define SQL_DIAG_GRANT 48
#define SQL_DIAG_INSERT 50
#define SQL_DIAG_REVOKE 59
#define SQL_DIAG_SELECT_CURSOR 85
#define SQL_DIAG_UNKNOWN_STATEMENT 0
#define SQL_DIAG_UPDATE_WHERE 82

/* SQL data type codes
 * These codes, despite being #defines, should not be changed;
 * they were chosen for compatibility with the ISO SQL definition
 * of data type codes.
 */
#define SQL_CHAR 1
#define SQL_NUMERIC 2
#define SQL_DECIMAL 3
#define SQL_INTEGER 4
#define SQL_SMALLINT 5
#define SQL_FLOAT 6
#define SQL_REAL 7
#define SQL_DOUBLE 8
#define SQL_DATETIME 9
#define SQL_VARCHAR 12

/* One-parameter shortcuts for date/time data types */
#define SQL_TYPE_DATE 91
#define SQL_TYPE_TIME 92
#define SQL_TYPE_TIMESTAMP 93

/* Statement attribute values for cursor sensitivity */
#define SQL_UNSPECIFIED 0
#define SQL_INSENSITIVE 1
#define SQL_SENSITIVE 2

/* GetTypeInfo() request for all data types */
#define SQL_ALL_TYPES 0

/* Default conversion code for BindCol(), BindParam() and GetData() */
#define SQL_DEFAULT 99

Data Management: SQL Call Level Interface (CLI) 249
/* GetData() code indicating that the application row descriptor
 * specifies the data type */
#define SQL_ARD_TYPE -99

/* SQL date/time type subcodes
 * These codes, despite being #defines, should not be changed;
 * they were chosen for compatibility with the ISO SQL definition
 * of date/time type subcodes. */
#define SQL_CODE_DATE 1
#define SQL_CODE_TIME 2
#define SQL_CODE_TIMESTAMP 3

/* CLI option values */
#define SQL_FALSE 0
#define SQL_TRUE 1

/* values of NULLABLE field in descriptor
 * These codes, despite being #defines, should not be changed;
 * they were chosen for compatibility with the ISO SQL definition
 * of the NULLABLE descriptor field. */
#define SQL_NO_NULLS 0
#define SQL_NULLABLE 1

/* Value returned by GetTypeInfo() to denote that it is
 * not known whether or not a data type supports null values. */
#define SQL_NULLABLE_UNKNOWN 2

/* Values returned by GetTypeInfo() to show WHERE clause
 * supported */
#define SQL_PRED_NONE 0
#define SQL_PRED_CHAR 1
#define SQL_PRED_BASIC 2

/* values of UNNAMED field in descriptor
 * These codes, despite being #defines, should not be changed;
 * they were chosen for compatibility with the ISO SQL definition
 * of the UNNAMED descriptor field. */
#define SQL_NAMED 0
#define SQL_UNNAMED 1

/* values of ALLOC_TYPE field in descriptor */
#define SQL_DESC_ALLOC_AUTO 1
#define SQL_DESC_ALLOC_USER 2

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

/* Codes used for FetchOrientation in FetchScroll(), and in DataSources() */
#define SQL_FETCH_NEXT 1
#define SQL_FETCH_FIRST 2

/* Other codes used for FetchOrientation in FetchScroll() */
#define SQL_FETCH_LAST 3
#define SQL_FETCH_PRIOR 4
#define SQL_FETCH_ABSOLUTE 5
#define SQL_FETCH_RELATIVE 6

/* EndTran() options */
#define SQL_COMMIT 0
#define SQL_ROLLBACK 1

/* null handles returned by AllocHandle() */
#define SQL_NULL_HENV 0
#define SQL_NULL_HDBC 0
#define SQL_NULL_HSTMT 0
#define SQL_NULL_HDESC 0

/* null handle used in place of parent handle when allocating HENV */
#define SQL_NULL_HANDLE 0L

/* Values that may appear in the result set of SpecialColumns() */
#define SQL_SCOPE_CURROW 0
#define SQL_SCOPE_TRANSACTION 1
#define SQL_SCOPE_SESSION 2
#define SQL_PC_UNKNOWN 0
#define SQL_PC_NON_PSEUDO 1
#define SQL_PC_PSEUDO 2

/* Reserved value for the IdentifierType argument of SpecialColumns() */
#define SQL_ROW_IDENTIFIER 1

/* Reserved values for UNIQUE argument of Statistics() */
#define SQL_INDEX_UNIQUE 0
#define SQL_INDEX_ALL 1

/* Values that may appear in the result set of Statistics() */
#define SQL_INDEX_CLUSTERED 1
#define SQL_INDEX_HASHED 2
#define SQL_INDEX_OTHER 3

/* GetFunctions() values to identify CLI functions */
#define SQL_API_SQLALLOCCONNECT 1
#define SQL_API_SQLALLOCENV 2
#define SQL_API_SQLALLOCHANDLE 1001
#define SQL_API_SQLALLOCHANDLE 3
#define SQL_API_SQLBINDCOL 4
#define SQL_API_SQLBINDPARAM 22
#define SQL_API_SQLCANCEL 5
#define SQL_API_SQLCLOSECURSOR 1003
#define SQL_API_SQLDATASOURCES 57
#define SQL_API_SQLDESCRIBECOL 8
#define SQL_API_SQLDISCONNECT 9
#define SQL_API_SQLERROR 10
#define SQL_API_SQLEXECDIRECT 11
#define SQL_API_SQLEXECUTE 12
#define SQL_API_SQLFETCH 13
#define SQL_API_SQLFETCHSCROLL 1021
#define SQL_API_SQLFREECONNECT 14
#define SQL_API_SQLFREEENV 15
#define SQL_API_SQLFREEHANDLE 1006
#define SQL_API_SQLGETCONNECTATTR 1007

#define SQL_API_SQLGETCONNECTOPTION 42
#define SQL_API_SQLGETCURSORNAME 17
#define SQL_API_SQLGETDATA 43
#define SQL_API_SQLGETDESCFIELD 1008
#define SQL_API_SQLGETDESCREC 1009
#define SQL_API_SQLGETDIAGFIELD 1010
#define SQL_API_SQLGETDIAGREC 1011
#define SQL_API_SQLGETENVATTR 1012
#define SQL_API_SQLGETFUNCTIONS 44
#define SQL_API_SQLGETINFO 45
#define SQL_API_SQLGETSTMTATTR 1014
#define SQL_API_SQLGETSTMTOPTION 46
#define SQL_API_SQLGETTYPEINFO 47
#define SQL_API_SQLLANGUAGES 2001
#define SQL_API_SQLNUMRESULTCOLS 18
#define SQL_API_SQLPARAMDATA 48
#define SQL_API_SQLPREPARE 19
#define SQL_API_SQLPUTDATA 49
#define SQL_API_SQLRELEASEENV 1015
#define SQL_API_SQLROWCOUNT 20
#define SQL_API_SQLSERVERINFO 2002
#define SQL_API_SQLSETCONNECTATTR 1016
#define SQL_API_SQLSETCONNECTOPTION 50
#define SQL_API_SQLSETCURSORNAME 21
#define SQL_API_SQLSETDESCFIELD 1017
#define SQL_API_SQLSETDESCREC 1018
#define SQL_API_SQLSETENVATTR 1019
#define SQL_API_SQLSETPARAM 22
#define SQL_API_SQLSETSTMTATTR 1020
#define SQL_API_SQLSETSTMTOPTION 51
#define SQL_API_SQLSPECIALCOLUMNS 52
#define SQL_API_SQLSTATISTICS 53
#define SQL_API_SQLTABLES 54
#define SQL_API_SQLTRANSACT 23

/* Information requested by GetInfo() */
#define SQL_MAX_DRIVER_CONNECTIONS 0
#define SQL_MAX_CONCURRENT_ACTIVITIES 1
#define SQL_DATA_SOURCE_NAME 2
#define SQL_FETCH_DIRECTION 8
#define SQL_SERVER_NAME 13
#define SQL_SEARCH_PATTERN_ESCAPE 14
#define SQL_DBMS_NAME 17
#define SQL_DBMS_VER 18
#define SQL_ACCESSIBLE_TABLES 19
#define SQL_CURSOR_COMMIT_BEHAVIOR 23
#define SQL_DATA_SOURCE_READ_ONLY 25
#define SQL_DEFAULT_TXN_ISOLATION 26
#define SQL_IDENTIFIER_CASE 28
#define SQL_IDENTIFIER_QUOTE_CHAR 29
#define SQL_MAX_COLUMN_NAME_LEN 30
#define SQL_MAX_CURSOR_NAME_LEN 31
#define SQL_MAX_SCHEMA_NAME_LEN 32
#define SQL_MAX_CATALOG_NAME_LEN 34
#define SQL_MAX_TABLE_NAME_LEN 35
#define SQL_SCROLL_CONCURRENCY 43
#define SQL_TXN_CAPABLE 46
#define SQL_USER_NAME 47
#define SQL_TXN_ISOLATION_OPTION 72
#define SQL_INTEGRITY 73
#define SQL_GETDATA_EXTENSIONS 81
#define SQL_NULL_COLLATION 85
#define SQL_ALTER_TABLE 86
#define SQL_ORDER_BY_COLUMNS_IN_SELECT 90
#define SQL_SPECIAL_CHARACTERS 94
#define SQL_MAX_COLUMNS_IN_GROUP_BY 97
#define SQL_MAX_COLUMNS_IN_INDEX 98
#define SQL_MAX_COLUMNS_IN_ORDER_BY 99
#define SQL_MAX_COLUMNS_IN_SELECT 100
#define SQL_MAX_COLUMNS_IN_TABLE 101
#define SQL_MAX_INDEX_SIZE 102
#define SQL_MAX_ROW_SIZE 104
#define SQL_MAX_STATEMENT_LEN 105
#define SQL_MAX_TABLES_IN_SELECT 106
#define SQL_MAX_USER_NAME_LEN 107
#define SQL_OJ_CAPABILITIES 115
#define SQL_XOPEN_CLI_YEAR 10000
#define SQL_CURSOR_SENSITIVITY 10001
#define SQL_DESCRIBE_PARAMETER 10002
#define SQL_CATALOG_NAME 10003
#define SQL_COLLATION_SEQ 10004
#define SQL_MAX_IDENTIFIER_LEN 10005

/* SQL_ALTER_TABLE bitmasks */
#define SQL_AT_ADD_COLUMN 0x00000001L
#define SQL_AT_DROP_COLUMN 0x00000002L
#define SQL_AT_ALTER_COLUMN 0x00000004L
#define SQL_AT_ADD_CONSTRAINT 0x00000008L
#define SQL_AT_DROP_CONSTRAINT 0x00000010L

/* SQL_CURSOR_COMMIT_BEHAVIOR values */
#define SQL_CB_DELETE 0
#define SQL_CB_CLOSE 1
#define SQL_CB_PRESERVE 2

/* SQL_FETCH_DIRECTION bitmasks */
#define SQL_FD_FETCH_NEXT 0x00000001L
#define SQL_FD_FETCH_FIRST 0x00000002L
#define SQL_FD_FETCH_LAST 0x00000004L
#define SQL_FD_FETCH_PRIOR 0x00000008L
#define SQL_FD_FETCH_ABSOLUTE 0x00000010L
#define SQL_FD_FETCH_RELATIVE 0x00000020L

/* SQL_GETDATA_EXTENSIONS bitmasks */
#define SQL_GD_ANY_COLUMN 0x00000001L

/* SQL_IDENTIFIER_CASE values */
#define SQL_IC_UPPER 1
#define SQL_IC_LOWER 2
#define SQL_IC_SENSITIVE 3
#define SQL_IC_MIXED 4

/* SQL_OJ_CAPABILITIES bitmasks */
#define SQL_OJ_LEFT 0x00000001L
#define SQL_OJ_RIGHT 0x00000002L
#define SQL_OJ_FULL 0x00000004L
#define SQL_OJ_NESTED 0x00000008L
#define SQL_OJ_NOT_ORDERED 0x00000010L
#define SQL_OJ_INNER 0x00000020L
#define SQL_OJ_ALL_COMPARISON_OPS 0x00000040L
/* SQL_SCROLL_CONCURRENCY bitmasks */
#define SQL_SCCO_READ_ONLY 0x00000001L
#define SQL_SCCO_LOCK 0x00000002L
#define SQL_SCCO_OPT_ROWVER 0x00000004L
#define SQL_SCCO_OPT_VALUES 0x00000008L
/* SQL_TXN_CAPABLE values */
#define SQL_TC_NONE 0
#define SQL_TC_DML 1
#define SQL_TC_ALL 2
#define SQL_TC_DDL_COMMIT 3
#define SQL_TC_DDL_IGNORE 4
/* SQL_TXN_ISOLATION_OPTION bitmasks */
#define SQL_TXN_READ_UNCOMMITTED 0x00000001L
#define SQL_TXN_READ_COMMITTED 0x00000002L
#define SQL_TXN_REPEATABLE_READ 0x00000004L
#define SQL_TXN_SERIALIZABLE 0x00000008L

SQLRETURN SQLAllocConnect(SQLHENV EnvironmentHandle,
                            SQLHDBC *ConnectionHandle);

SQLRETURN SQLAllocEnv(SQLHENV *EnvironmentHandle);

SQLRETURN SQLAllocHandle(SQLSMALLINT HandleType,
                          SQLINTEGER InputHandle, SQLINTEGER *OutputHandle);

SQLRETURN SQLAllocStmt(SQLHDBC ConnectionHandle,
                        SQLHSTMT *StatementHandle);

SQLRETURN SQLBindCol(SQLHSTMT StatementHandle, SQLSMALLINT ColumnNumber,
                      SQLSMALLINT TargetType, SQLPOINTER TargetValue,
                      SQLINTEGER BufferLength, SQLINTEGER *StrLen_or_Ind);

SQLRETURN SQLBindParam(SQLHSTMT StatementHandle,
                       SQLSMALLINT ParameterNumber, SQLSMALLINT ValueType,
                       SQLSMALLINT ParameterType, SQLINTEGER LengthPrecision,
                       SQLSMALLINT ParameterScale, SQLPOINTER ParameterValue,
                       SQLINTEGER *StrLen_or_Ind);

SQLRETURN SQLCancel(SQLHSTMT StatementHandle);

SQLRETURN SQLCloseCursor(SQLHSTMT StatementHandle);

SQLRETURN SQLColAttribute (SQLHSTMT StatementHandle,
                          SQLSMALLINT ColumnNumber, SQLSMALLINT FieldIdentifier,
                          SQLPOINTER CharacterAttribute, SQLSMALLINT BufferLength,
                          SQLINTEGER *StringLength, SQLPOINTER NumericAttribute);

SQLRETURN SQLColumns(SQLHSTMT StatementHandle,
                      SQLCHAR *CatalogName, SQLSMALLINT NameLength1,
                      SQLCHAR *SchemaName, SQLSMALLINT NameLength2,
                      SQLCHAR *TableName, SQLSMALLINT NameLength3,
                      SQLCHAR *ColumnName, SQLSMALLINT NameLength4);

SQLRETURN SQLConnect(SQLHDBC ConnectionHandle,
                     SQLCHAR *ServerName, SQLSMALLINT NameLength1,
                     SQLCHAR *UserName, SQLSMALLINT NameLength2,
                     SQLCHAR *Authentication, SQLSMALLINT NameLength3);

SQLRETURN SQLCopyDesc(SQLHDESC SourceDescHandle,
                      SQLHDESC TargetDescHandle);

SQLRETURN SQLDataSources(SQLHENV EnvironmentHandle,
                         SQLSMALLINT Direction, SQLCHAR *ServerName,
                         SQLSMALLINT BufferLength1, SQLSMALLINT *NameLength1,
Language-specific Header Files

C-language File <sqlcli.h>

SQLCHAR *Description, SQLSMALLINT BufferLength2,
SQLSMALLINT *NameLength2);

SQLRETURN SQLDescribeCol(SQLHstmt StatementHandle,
SQLSMALLINT ColumnNumber, SQLCHAR *ColumnName,
SQLSMALLINT BufferLength, SQLSMALLINT *NameLength,
SQLSMALLINT *DataType, SQLINTEGER *ColumnSize,
SQLSMALLINT *DecimalDigits, SQLSMALLINT *Nullable);

SQLRETURN SQLDisconnect(SQLHDBC ConnectionHandle);

SQLRETURN SQLEndTran(SQLSMALLINT HandleType, SQLINTEGER Handle,
SQLSMALLINT CompletionType);

SQLRETURN SQLError(SQLHENV EnvironmentHandle,
SQLHDBC ConnectionHandle, SQLHstmt StatementHandle,
SQLCHAR *Sqlstate, SQLINTEGER *NativeError,
SQLCHAR *MessageText, SQLSMALLINT BufferLength,
SQLSMALLINT *TextLength);

SQLRETURN SQLExecDirect(SQLHstmt StatementHandle,
SQLCHAR *StatementText, SQLINTEGER TextLength);

SQLRETURN SQLExecute(SQLHstmt StatementHandle);

SQLRETURN SQLFetch(SQLHstmt StatementHandle);

SQLRETURN SQLFetchScroll(SQLHstmt StatementHandle,
SQLSMALLINT FetchOrientation, SQLINTEGER FetchOffset);

SQLRETURN SQLFreeConnect(SQLHDBC ConnectionHandle);

SQLRETURN SQLFreeEnv(SQLHENV EnvironmentHandle);

SQLRETURN SQLFreeHandle(SQLSMALLINT HandleType, SQLINTEGER Handle);

SQLRETURN SQLFreeStmt(SQLHstmt StatementHandle,
SQLSMALLINT Option);

SQLRETURN SQLGetConnectAttr(SQLHDBC ConnectionHandle,
SQLINTEGER Attribute, SQLPOINTER Value,
SQLINTEGER BufferLength, SQLINTEGER *StringLength);

SQLRETURN SQLGetConnectOption(SQLHDBC ConnectionHandle,
SQLSMALLINT Option, SQLPOINTER Value);

SQLRETURN SQLGetCursorName(SQLHstmt StatementHandle,
SQLCHAR *CursorName, SQLSMALLINT BufferLength,
SQLSMALLINT *NameLength);

SQLRETURN SQLGetData(SQLHstmt StatementHandle,
SQLSMALLINT ColumnNumber, SQLSMALLINT TargetType,
SQLPOINTER TargetValue, SQLINTEGER BufferLength,
SQLINTEGER *StrLen_or_Ind);

SQLRETURN SQLGetDescField(SQLHdesc DescriptorHandle,
SQLSMALLINT RecNumber, SQLSMALLINT FieldIdentifier,
SQLPOINTER Value, SQLINTEGER BufferLength,
SQLINTEGER *StringLength);

SQLRETURN SQLGetDescRec(SQLHdesc DescriptorHandle,
SQLSMALLINT RecNumber, SQLCHAR *Name,
SQLSMALLINT BufferLength, SQLSMALLINT *StringLength,
SQLSMALLINT *Type, SQLSMALLINT *SubType, SQLINTEGER *Length,
SQLSMALLINT *Precision, SQLSMALLINT *Scale,
SQLSMALLINT *Nullable);

Data Management: SQL Call Level Interface (CLI)
C-language File `<sqlcli.h>`  Language-specific Header Files

```c
SQLRETURN SQLGetDiagField(SQLSMALLINT HandleType, SQLINTEGER Handle,
                           SQLSMALLINT RecNumber, SQLSMALLINT DiagIdentifier,
                           SQLPOINTER DiagInfo, SQLSMALLINT BufferLength,
                           SQLSMALLINT *StringLength);

SQLRETURN SQLGetDiagRec(SQLSMALLINT HandleType, SQLINTEGER Handle,
                         SQLSMALLINT RecNumber, SQLCHAR *Sqlstate,
                         SQLINTEGER *NativeError, SQLCHAR *MessageText,
                         SQLSMALLINT BufferLength, SQLSMALLINT *TextLength);

SQLRETURN SQLGetEnvAttr(SQLHENV EnvironmentHandle, SQLINTEGER Attribute,
                         SQLPOINTER Value, SQLINTEGER BufferLength,
                         SQLINTEGER *StringLength);

SQLRETURN SQLGetFunctions(SQLHDBC ConnectionHandle, SQLSMALLINT FunctionId,
                          SQLSMALLINT *Supported);

SQLRETURN SQLGetInfo(SQLHDBC ConnectionHandle, SQLSMALLINT InfoType,
                      SQLPOINTER InfoValue, SQLSMALLINT BufferLength,
                      SQLSMALLINT *StringLength);

SQLRETURN SQLGetStmtAttr(SQLHSTMT StatementHandle, SQLINTEGER Attribute,
                          SQLPOINTER Value, SQLINTEGER BufferLength,
                          SQLINTEGER *StringLength);

SQLRETURN SQLGetStmtOption(SQLHSTMT StatementHandle, SQLSMALLINT Option,
                           SQLPOINTER Value);

SQLRETURN SQLGetTypeInfo(SQLSTMT StatementHandle, SQLSMALLINT DataType);

SQLRETURN SQLNumResultCols(SQLHSTMT StatementHandle, SQLSMALLINT *ColumnCount);

SQLRETURN SQLParamData(SQLSTMT StatementHandle, SQLPOINTER *Value);

SQLRETURN SQLPrepare(SQLHSTMT StatementHandle, SQLCHAR *StatementText,
                     SQLINTEGER TextLength);

SQLRETURN SQLPutData(SQLSTMT StatementHandle, SQLPOINTER Data, SQLINTEGER StrLen_or_Ind);

SQLRETURN SQLRowCount(SQLHSTMT StatementHandle, SQLINTEGER *RowCount);

SQLRETURN SQLSetConnectAttr(SQLHDBC ConnectionHandle, SQLINTEGER Attribute,
                            SQLPOINTER Value, SQLINTEGER StringLength);

SQLRETURN SQLSetConnectOption(SQLHDBC ConnectionHandle, SQLSMALLINT Option,
                              SQLPOINTER Value);

SQLRETURN SQLSetCursorName(SQLHSTMT StatementHandle, SQLCHAR *CursorName,
                           SQLSMALLINT NameLength);

SQLRETURN SQLSetDescField(SQLHDESC DescriptorHandle, SQLSMALLINT RecNumber,
                          SQLSMALLINT FieldIdentifier, SQLPOINTER Value,
                          SQLINTEGER BufferLength);

SQLRETURN SQLSetDescRec(SQLHDESC DescriptorHandle, SQLSMALLINT RecNumber,
                        SQLSMALLINT Type, SQLSMALLINT SubType, SQLINTEGER Length,
                        SQLSMALLINT Precision, SQLSMALLINT Scale,
                        SQLPOINTER Data, SQLINTEGER *StringLength,
                        SQLSMALLINT *Indicator);

SQLRETURN SQLSetEnvAttr(SQLHENV EnvironmentHandle, SQLINTEGER Attribute,
                        SQLPOINTER Value,
```
#define SQLSetParam SQLBindParam

SQLRETURN SQLSetStmtAttr(SQLHSTMT StatementHandle,
                        SQLINTEGER Attribute, SQLPOINTER Value,
                        SQLINTEGER StringLength);

SQLRETURN SQLSetStmtOption(SQLHSTMT StatementHandle,
                         SQLSMALLINT Option, SQLPOINTER Value);

SQLRETURN SQLSpecialColumns(SQLSTMT StatementHandle,
                           SQLSMALLINT IdentifierType, SQLCHAR *CatalogName,
                           SQLSMALLINT NameLength1, SQLCHAR *SchemaName,
                           SQLSMALLINT NameLength2, SQLCHAR *TableName,
                           SQLSMALLINT NameLength3, SQLSMALLINT Scope,
                           SQLSMALLINT Nullable);

SQLRETURN SQLStatistics(SQLSTMT StatementHandle,
                     SQLCHAR *CatalogName, SQLSMALLINT NameLength1,
                     SQLCHAR *SchemaName, SQLSMALLINT NameLength2,
                     SQLCHAR *TableName, SQLSMALLINT NameLength3,
                     SQLSMALLINT Unique, SQLSMALLINT Reserved);

SQLRETURN SQLTables(SQLHSTMT StatementHandle,
                  SQLCHAR *CatalogName, SQLSMALLINT NameLength1,
                  SQLCHAR *SchemaName, SQLSMALLINT NameLength2,
                  SQLCHAR *TableName, SQLSMALLINT NameLength3,
                  SQLCHAR *TableType, SQLSMALLINT NameLength4);

SQLRETURN SQLTransact(SQLHENV EnvironmentHandle,
                  SQLHDBC ConnectionHandle, SQLSMALLINT CompletionType);
C.2 COBOL File SQLCLI.CBH

COBOL applications include the typical header file SQLCLI.CBH by containing the following statement:

COPY SQLCLI.CBH

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

* SPECIAL LENGTH/INDICATOR VALUES
  01 SQL-NULL-DATA PIC S9(9) BINARY VALUE IS -1.
  01 SQL-DATA-AT-EXEC PIC S9(9) BINARY VALUE IS -2.

* RETURN VALUES FROM FUNCTIONS
  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.
  01 SQL-NEED-DATA PIC S9(4) BINARY VALUE IS 99.

* FLAGS FOR NULL-TERMINATED STRING
  01 SQL-NTS PIC S9(4) BINARY VALUE IS -3.
  01 SQL-NTSL PIC S9(9) BINARY VALUE IS -3.

* MAXIMUM MESSAGE LENGTH
  * VENDORS MAY INCREASE THIS CONSTANT, BUT ITS VALUE MUST BE AT LEAST 512.
  01 SQL-MAX-MESSAGE-LENGTH PIC S9(4) BINARY VALUE IS 512.

* MAXIMUM IDENTIFIER LENGTH
  01 SQL-MAX-ID-LENGTH PIC S9(4) BINARY VALUE IS 18.

* DATE/TIME LENGTH CONSTANTS
  * ADD P+1 FOR TIME AND TIMESTAMP IF PRECISION IS NONZERO
  01 SQL-DATE-LEN PIC S9(4) BINARY VALUE IS 10.
  01 SQL-TIME-LEN PIC S9(4) BINARY VALUE IS 8.
  01 SQL-TIMESTAMP-LEN PIC S9(4) BINARY VALUE IS 19.

* ENVIRONMENT ATTRIBUTE
  01 SQL-ATTR-OUTPUT-NTS PIC S9(9) BINARY VALUE IS 10001.
  01 SQL-OPT-OUTPUT-NTS PIC S9(4) BINARY VALUE IS 10001.

* CONNECTION ATTRIBUTE
  01 SQL-ATTR-AUTO-IPD PIC S9(9) BINARY VALUE IS 10001.
  01 SQL-OPT-AUTO-IPD PIC S9(4) BINARY VALUE IS 10001.

* CONNECTION AND STATEMENT ATTRIBUTES
  01 SQL-ATTR-METADATA-ID PIC S9(9) BINARY VALUE IS 10014.

* HANDLE TYPE IDENTIFIERS
  01 SQL-HANDLE-ENV PIC S9(4) BINARY VALUE IS 1.
  01 SQL-HANDLE-DBC PIC S9(4) BINARY VALUE IS 2.
  01 SQL-HANDLE-STMT PIC S9(4) BINARY VALUE IS 3.
  01 SQL-HANDLE-DESC PIC S9(4) BINARY VALUE IS 4.

* STATEMENT ATTRIBUTES
  01 SQL-ATTR-APP-ROW-DESC PIC S9(9) BINARY VALUE IS 10010.
  01 SQL-ATTR-APP-PARAM-DESC PIC S9(9) BINARY VALUE IS 10011.
  01 SQL-ATTR-IMP-ROW-DESC PIC S9(9) BINARY VALUE IS 10012.
  01 SQL-ATTR-IMP-PARAM-DESC PIC S9(9) BINARY VALUE IS 10013.
  01 SQL-ATTR-CURSOR-SCROLLABLE PIC S9(9) BINARY VALUE IS -1.
  01 SQL-ATTR-CURSOR-SENSITIVITY PIC S9(9) BINARY VALUE IS -2.
  01 SQL-OPT-APP-ROW-DESC PIC S9(4) BINARY VALUE IS 10010.
01 SQL-OPT-APP-PARAM-DESC PIC S9(4) BINARY VALUE IS 10011.
01 SQL-OPT-IMP-ROW-DESC PIC S9(4) BINARY VALUE IS 10012.
01 SQL-OPT-IMP-PARAM-DESC PIC S9(4) BINARY VALUE IS 10013.
01 SQL-OPT-CURSOR-SCROLLABLE PIC S9(4) BINARY VALUE IS -1.
01 SQL-OPT-CURSOR-SENSITIVITY PIC S9(4) BINARY VALUE IS -2.

* IDENTIFIERS OF FIELDS IN THE SQL DESCRIPTOR
01 SQL-DESC-COUNT PIC S9(4) BINARY VALUE IS 1001.
01 SQL-DESC-TYPE PIC S9(4) BINARY VALUE IS 1002.
01 SQL-DESC-LENGTH PIC S9(4) BINARY VALUE IS 1003.
01 SQL-DESC-OCTET-LENGTH-PTR PIC S9(4) BINARY VALUE IS 1004.
01 SQL-DESC-PRECISION PIC S9(4) BINARY VALUE IS 1005.
01 SQL-DESC-SCALE PIC S9(4) BINARY VALUE IS 1006.
01 SQL-DESC-DATETIME-INTERVAL-CODE PIC S9(4) BINARY VALUE IS 1007.
01 SQL-DESC-NULLABLE PIC S9(4) BINARY VALUE IS 1008.
01 SQL-DESC-INDICATOR-PTR PIC S9(4) BINARY VALUE IS 1009.
01 SQL-DESC-DATA-PTR PIC S9(4) BINARY VALUE IS 1010.
01 SQL-DESC-NAME PIC S9(4) BINARY VALUE IS 1011.
01 SQL-DESC-UNNAMED PIC S9(4) BINARY VALUE IS 1012.
01 SQL-DESC-OCTET-LENGTH PIC S9(4) BINARY VALUE IS 1013.
01 SQL-DESC-ALLOC-TYPE PIC S9(4) BINARY VALUE IS 1099.

* IDENTIFIERS OF FIELDS IN THE DIAGNOSTICS AREA
01 SQL-DIAG-RETURNCODE PIC S9(4) BINARY VALUE IS 1.
01 SQL-DIAG-NUMBER PIC S9(4) BINARY VALUE IS 2.
01 SQL-DIAG-ROW-COUNT PIC S9(4) BINARY VALUE IS 3.
01 SQL-DIAG-SQLSTATE PIC S9(4) BINARY VALUE IS 4.
01 SQL-DIAG-NATIVE PIC S9(4) BINARY VALUE IS 5.
01 SQL-DIAG-MESSAGE-TEXT PIC S9(4) BINARY VALUE IS 6.
01 SQL-DIAG-CLASS-ORIGIN PIC S9(4) BINARY VALUE IS 8.
01 SQL-DIAG-SUBCLASS-ORIGIN PIC S9(4) BINARY VALUE IS 9.
01 SQL-DIAG-CONNECTION-NAME PIC S9(4) BINARY VALUE IS 10.
01 SQL-DIAG-SERVER-NAME PIC S9(4) BINARY VALUE IS 11.
01 SQL-DIAG-DYNAMIC-FUNCTION-CODE PIC S9(4) BINARY VALUE IS 12.

* DYNAMIC FUNCTION CODES
01 SQL-DIAG-ALTER-TABLE PIC S9(9) BINARY VALUE IS 4.
01 SQL-DIAG-CREATE-INDEX PIC S9(9) BINARY VALUE IS -1.
01 SQL-DIAG-CREATE-TABLE PIC S9(9) BINARY VALUE IS 77.
01 SQL-DIAG-CREATE-VIEW PIC S9(9) BINARY VALUE IS 84.
01 SQL-DIAG-DELETE-WHERE PIC S9(9) BINARY VALUE IS 19.
01 SQL-DIAG-DELETE-INDEX PIC S9(9) BINARY VALUE IS -2.
01 SQL-DIAG-DROP-TABLE PIC S9(9) BINARY VALUE IS 32.
01 SQL-DIAG-DROP-DPOINT PIC S9(9) BINARY VALUE IS 36.
01 SQL-DIAG-DYNAMIC-UPDATE-CURSOR PIC S9(9) BINARY VALUE IS 38.
01 SQL-DIAG-DYNAMIC-UPDATE-CURSOR PIC S9(9) BINARY VALUE IS 81.
01 SQL-DIAG-DYNAMIC-DELETE-CURSOR PIC S9(9) BINARY VALUE IS 38.
01 SQL-DIAG-DYNAMIC-UPDATE-CURSOR PIC S9(9) BINARY VALUE IS 81.
01 SQL-DIAG-ALTER-TABLE PIC S9(9) BINARY VALUE IS 4.
01 SQL-DIAG-DELETE-INDEX PIC S9(9) BINARY VALUE IS -1.
01 SQL-DIAG-DELETE-INDEX PIC S9(9) BINARY VALUE IS -2.
01 SQL-DIAG-DROP-TABLE PIC S9(9) BINARY VALUE IS 32.
01 SQL-DIAG-DYNAMIC-DELETE-CURSOR PIC S9(9) BINARY VALUE IS 38.
01 SQL-DIAG-DYNAMIC-UPDATE-CURSOR PIC S9(9) BINARY VALUE IS 81.
01 SQL-DIAG-DYNAMIC-DELETE-CURSOR PIC S9(9) BINARY VALUE IS 38.
01 SQL-DIAG-DYNAMIC-UPDATE-CURSOR PIC S9(9) BINARY VALUE IS 81.
01 SQL-DIAG-DYNAMIC-DELETE-CURSOR PIC S9(9) BINARY VALUE IS 38.
01 SQL-DIAG-DYNAMIC-UPDATE-CURSOR PIC S9(9) BINARY VALUE IS 81.
01 SQL-DIAG-ALTER-TABLE PIC S9(9) BINARY VALUE IS 4.
01 SQL-DIAG-CREATE-INDEX PIC S9(9) BINARY VALUE IS -1.
01 SQL-DIAG-CREATE-TABLE PIC S9(9) BINARY VALUE IS 77.
01 SQL-DIAG-CREATE-VIEW PIC S9(9) BINARY VALUE IS 84.
01 SQL-DIAG-DELETE-WHERE PIC S9(9) BINARY VALUE IS 19.
01 SQL-DIAG-DELETE-INDEX PIC S9(9) BINARY VALUE IS -2.
01 SQL-DIAG-DROP-TABLE PIC S9(9) BINARY VALUE IS 32.
01 SQL-DIAG-DYNAMIC-UPDATE-CURSOR PIC S9(9) BINARY VALUE IS 81.
01 SQL-DIAG-UNKNOWN-STATEMENT PIC S9(9) BINARY VALUE IS 0.
01 SQL-DIAG-UPDATE-WHERE PIC S9(9) BINARY VALUE IS 82.

* SQL DATA TYPE CODES
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-DATETIME PIC S9(4) BINARY VALUE IS 9.
01 SQL-VARCHAR PIC S9(4) BINARY VALUE IS 12.

* ONE-PARAMETER SHORTCUTS FOR DATE/TIME DATA TYPES
01 SQL-TYPE-DATE PIC S9(4) BINARY VALUE IS 91.
01 SQL-TYPE-TIME PIC S9(4) BINARY VALUE IS 92.
01 SQL-TYPE-TIMESTAMP PIC S9(4) BINARY VALUE IS 93.

* STATEMENT ATTRIBUTE VALUES FOR CURSOR SENSITIVITY
01 SQL-UNSPECIFIED PIC S9(9) BINARY VALUE IS 0.
01 SQL-INSENSITIVE PIC S9(9) BINARY VALUE IS 1.
01 SQL-SENSITIVE PIC S9(9) BINARY VALUE IS 2.

* SQLRGETTYPEINFO() REQUEST FOR ALL DATA TYPES
01 SQL-ALL-TYPES PIC S9(4) BINARY VALUE IS 0.

* DEFAULT CONVERSION CODE FOR SQLBINDCOL() AND SQLBINDPARAM()
01 SQL-DEFAULT PIC S9(4) BINARY VALUE IS 99.

* SQLGETDATA() CODE INDICATING THAT APPLICATION ROW DESCRIPTOR
* SPECIFIES THE DATA TYPE
01 SQL-ARD-TYPE PIC S9(4) BINARY VALUE IS -99.

* DATE/TIME TYPE SUBCODES
01 SQL-CODE-DATE PIC S9(4) BINARY VALUE IS 1.
01 SQL-CODE-TIME PIC S9(4) BINARY VALUE IS 2.
01 SQL-CODE-TIMESTAMP PIC S9(4) BINARY VALUE IS 3.

* CLI OPTION VALUES
01 SQL-FALSE PIC S9(4) BINARY VALUE IS 0.
01 SQL-FALSEL PIC S9(9) BINARY VALUE IS 0.
01 SQL-TRUE PIC S9(4) BINARY VALUE IS 1.
01 SQL-TRUEN PIC S9(9) BINARY VALUE IS 1.

* VALUES OF NULLABLE FIELD IN DESCRIPTOR
01 SQL-NO-NULLS PIC S9(4) BINARY VALUE IS 0.
01 SQL-NULLABLE PIC S9(4) BINARY VALUE IS 1.

*VALUE RETURNED BY SQLRGETTYPEINFO() TO DENOTE THAT IT IS
*NOT KNOWN WHETHER OR NOT A DATA TYPE SUPPORTS NULL VALUES.
01 SQL-NULLABLE-UNKNOWN PIC S9(4) BINARY VALUE IS 2.

*VALUES RETURNED BY SQLRGETTYPEINFO() TO SHOW SUPPORT IN
*WHERE CLAUSE
01 SQL-PRED-NONE PIC S9(4) BINARY VALUE IS 0.
01 SQL-PRED-CHAR PIC S9(4) BINARY VALUE IS 1.
01 SQL-PRED-BASIC PIC S9(4) BINARY VALUE IS 2.

* VALUES OF UNNAMED FIELD IN DESCRIPTOR
01 SQL-NAMED PIC S9(4) BINARY VALUE IS 0.
01 SQL-UNNAMED PIC S9(4) BINARY VALUE IS 1.

* VALUES OF ALLOC-TYPE FIELD IN DESCRIPTOR
01 SQL-DESC-ALLOC-AUTO PIC S9(4) BINARY VALUE IS 1.
01 SQL-DESC-ALLOC-USER PIC S9(4) BINARY VALUE IS 2.

* SQLRFREESTMT 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-PARMS PIC S9(4) BINARY VALUE IS 3.

* CODES USED FOR FETCHORIENTATION IN SQLRFETCHSCROLL(),
* AND IN SQLRDATASOURCES()
01 SQL-FETCH-NEXT PIC S9(4) BINARY VALUE IS 1.
01 SQL-FETCH-FIRST PIC S9(4) BINARY VALUE IS 2.

* OTHER CODES USED FOR FETCHORIENTATION IN SQLRFETCHSCROLL()
01 SQL-FETCH-LAST PIC S9(4) BINARY VALUE IS 3.
01 SQL-FETCH-PRIOR PIC S9(4) BINARY VALUE IS 4.
01 SQL-FETCH-ABSOLUTE PIC S9(4) BINARY VALUE IS 5.
01 SQL-FETCH-RELATIVE PIC S9(4) BINARY VALUE IS 6.

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

* NULL HANDLES RETURNED BY SQLRALLOCHANDLE()
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.
01 SQL-NULL-HDESC PIC S9(9) BINARY VALUE IS 0.

* NULL HANDLE USED IN PLACE OF PARENT HANDLE WHEN ALLOCATING HENV
01 SQL-NULL-HANDLE PIC S9(9) BINARY VALUE IS 0.

* VALUES THAT MAY APPEAR IN THE RESULT SET OF SQLRSPECIALCOLUMNS()
01 SQL-SCOPE-CURROW PIC S9(4) BINARY VALUE IS 0.
01 SQL-SCOPE-TRANSACTION PIC S9(4) BINARY VALUE IS 1.
01 SQL-SCOPE-SESSION PIC S9(4) BINARY VALUE IS 2.
01 SQL-PC-UNKNOWN PIC S9(4) BINARY VALUE IS 0.
01 SQL-PC-NON-PSEUDO PIC S9(4) BINARY VALUE IS 1.
01 SQL-PC-PSEUDO PIC S9(4) BINARY VALUE IS 2.

* RESERVED VALUE FOR THE IDENTIFIERTYPE ARGUMENT OF SQLRSPECIALCOLUMNS()
01 SQL-ROW-IDENTIFIER PIC S9(4) BINARY VALUE IS 1.

* RESERVED VALUES FOR UNIQUE ARGUMENT OF SQLRSTATISTICS()
01 SQL-INDEX-UNIQUE PIC S9(4) BINARY VALUE IS 0.
01 SQL-INDEX-ALL PIC S9(4) BINARY VALUE IS 1.

* VALUES THAT MAY APPEAR IN THE RESULT SET OF SQLRSTATISTICS()
01 SQL-INDEX-CLUSTERED PIC S9(4) BINARY VALUE IS 1.
01 SQL-INDEX-HASHED PIC S9(4) BINARY VALUE IS 2.
01 SQL-INDEX-OTHER PIC S9(4) BINARY VALUE IS 3.

* SQLRGETFUNCTIONS() VALUES TO IDENTIFY CLI FUNCTIONS
01 SQL-API-SQLALLOCCONNECT PIC S9(4) BINARY VALUE IS 1.
01 SQL-API-SQLALLOCENV PIC S9(4) BINARY VALUE IS 2.
01 SQL-API-SQLALLOCHANDLE PIC S9(4) BINARY VALUE IS 1001.
01 SQL-API-SQLALLOCGSTMT PIC S9(4) BINARY VALUE IS 3.
01 SQL-API-SQLBINDCOL PIC S9(4) BINARY VALUE IS 1004.
01 SQL-API-SQLBINDPARAM PIC S9(4) BINARY VALUE IS 4.
01 SQL-API-SQLCOLUMNS PIC S9(4) BINARY VALUE IS 57.
01 SQL-API-SQLCOPYDESC PIC S9(4) BINARY VALUE IS 6.
01 SQL-API-SQLDATASOURCES PIC S9(4) BINARY VALUE IS 7.
01 SQL-API-SQLDESCRIBECOL PIC S9(4) BINARY VALUE IS 1003.
01 SQL-API-SQLDISCONNECT PIC S9(4) BINARY VALUE IS 8.
01 SQL-API-SQLDISCONNECT PIC S9(4) BINARY VALUE IS 9.
01 SQL-API-SQLENDTRAN PIC S9(4) BINARY VALUE IS 1004.
01 SQL-API-SQLERROR PIC S9(4) BINARY VALUE IS 10.
01 SQL-API-SQLEXECDIRECT PIC S9(4) BINARY VALUE IS 11.
01 SQL-API-SQLEXECUTE PIC S9(4) BINARY VALUE IS 12.
01 SQL-API-SQLFETCH PIC S9(4) BINARY VALUE IS 13.
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01 SQL-API-SQLFETCHSCROLL</td>
<td>PIC S9(4) BINARY VALUE IS 1021.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLFREECONNECT</td>
<td>PIC S9(4) BINARY VALUE IS 14.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLFREEENV</td>
<td>PIC S9(4) BINARY VALUE IS 15.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLFREEHANDLE</td>
<td>PIC S9(4) BINARY VALUE IS 1006.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLFREESTMT</td>
<td>PIC S9(4) BINARY VALUE IS 16.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLGETCONNECTATTR</td>
<td>PIC S9(4) BINARY VALUE IS 42.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLGETCONNECTOPTION</td>
<td>PIC S9(4) BINARY VALUE IS 17.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLGETCURSORNAME</td>
<td>PIC S9(4) BINARY VALUE IS 19.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLGETDATA</td>
<td>PIC S9(4) BINARY VALUE IS 11.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLGETDESCFIELD</td>
<td>PIC S9(4) BINARY VALUE IS 1009.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLGETDESCREC</td>
<td>PIC S9(4) BINARY VALUE IS 1010.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLGETDIAGFIELD</td>
<td>PIC S9(4) BINARY VALUE IS 1011.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLGETDIAGREC</td>
<td>PIC S9(4) BINARY VALUE IS 1012.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLGETENVATTR</td>
<td>PIC S9(4) BINARY VALUE IS 1014.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLGETFUNCTIONS</td>
<td>PIC S9(4) BINARY VALUE IS 44.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLGETINFO</td>
<td>PIC S9(4) BINARY VALUE IS 45.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLGETSTMTATTR</td>
<td>PIC S9(4) BINARY VALUE IS 1015.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLGETSTMTOPTION</td>
<td>PIC S9(4) BINARY VALUE IS 46.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLGETTYPEINFO</td>
<td>PIC S9(4) BINARY VALUE IS 47.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLLANGUAGES</td>
<td>PIC S9(4) BINARY VALUE IS 2001.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLNUMRESULTCOLS</td>
<td>PIC S9(4) BINARY VALUE IS 18.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLPARAMDATA</td>
<td>PIC S9(4) BINARY VALUE IS 48.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLPREPARE</td>
<td>PIC S9(4) BINARY VALUE IS 19.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLPUTDATA</td>
<td>PIC S9(4) BINARY VALUE IS 49.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLRELEASEENV</td>
<td>PIC S9(4) BINARY VALUE IS 1015.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLRORDER</td>
<td>PIC S9(4) BINARY VALUE IS 20.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLSERVERINFO</td>
<td>PIC S9(4) BINARY VALUE IS 2002.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLSETCONNECTATTR</td>
<td>PIC S9(4) BINARY VALUE IS 1016.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLSETCONNECTOPTION</td>
<td>PIC S9(4) BINARY VALUE IS 50.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLSETCURSORNAME</td>
<td>PIC S9(4) BINARY VALUE IS 21.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLSETDESCFIELD</td>
<td>PIC S9(4) BINARY VALUE IS 1017.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLSETDESCREC</td>
<td>PIC S9(4) BINARY VALUE IS 1018.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLSETENVATTR</td>
<td>PIC S9(4) BINARY VALUE IS 1019.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLSETPARAM</td>
<td>PIC S9(4) BINARY VALUE IS 22.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLSETSTMTATTR</td>
<td>PIC S9(4) BINARY VALUE IS 1020.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLSETSTMTOPTION</td>
<td>PIC S9(4) BINARY VALUE IS 51.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLSPECIALCOLUMNS</td>
<td>PIC S9(4) BINARY VALUE IS 52.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLSTATISTICS</td>
<td>PIC S9(4) BINARY VALUE IS 53.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLTABLES</td>
<td>PIC S9(4) BINARY VALUE IS 54.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 SQL-API-SQLTRANSC</td>
<td>PIC S9(4) BINARY VALUE IS 23.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* INFORMATION REQUESTED BY SQLGETINFO()
01 SQL-MAX-CATALOG-NAME-LEN PIC S9(4) BINARY VALUE IS 34.
01 SQL-MAX-TABLE-NAME-LEN PIC S9(4) BINARY VALUE IS 35.
01 SQL-SCROLL-CONCURRENCY PIC S9(4) BINARY VALUE IS 43.
01 SQL-TXN-CAPABLE PIC S9(4) BINARY VALUE IS 46.
01 SQL-USER-NAME PIC S9(4) BINARY VALUE IS 47.
01 SQL-TXN-ISOLATION-OPTION PIC S9(4) BINARY VALUE IS 72.
01 SQL-INTEGRITY PIC S9(4) BINARY VALUE IS 73.
01 SQL-GETDATA-EXTENSIONS PIC S9(4) BINARY VALUE IS 81.
01 SQL-NULL-COLLATION PIC S9(4) BINARY VALUE IS 85.
01 SQL-ALTER-TABLE PIC S9(4) BINARY VALUE IS 86.
01 SQL-ORDER-BY-COLUMNS-IN-SELECT PIC S9(4) BINARY VALUE IS 90.
01 SQL-SPECIAL-CHARACTERS PIC S9(4) BINARY VALUE IS 94.
01 SQL-MAX-COLUMNS-IN-GROUP-BY PIC S9(4) BINARY VALUE IS 97.
01 SQL-MAX-COLUMNS-IN-INDEX PIC S9(4) BINARY VALUE IS 98.
01 SQL-MAX-COLUMNS-IN-ORDER-BY PIC S9(4) BINARY VALUE IS 99.
01 SQL-MAX-COLUMNS-IN-TABLE PIC S9(4) BINARY VALUE IS 100.
01 SQL-MAX-COLUMNS-IN-TABLE PIC S9(4) BINARY VALUE IS 101.
01 SQL-MAX-INDEX-SIZE PIC S9(4) BINARY VALUE IS 102.
01 SQL-MAX-ROW-SIZE PIC S9(4) BINARY VALUE IS 104.
01 SQL-MAX-STATEMENT-LEN PIC S9(4) BINARY VALUE IS 105.
01 SQL-MAX-TABLES-IN-SELECT PIC S9(4) BINARY VALUE IS 106.
01 SQL-MAX-TABLES-IN-SELECT PIC S9(4) BINARY VALUE IS 107.
01 SQL-OJ-CAPABILITIES PIC S9(4) BINARY VALUE IS 115.
01 SQL-XOPEN-CLI-YEAR PIC S9(4) BINARY VALUE IS 10000.
01 SQL-CURSOR-SENSITIVITY PIC S9(4) BINARY VALUE IS 10001.
01 SQL-DESCRIBE-PARAMETER PIC S9(4) BINARY VALUE IS 10002.
01 SQL-SQL-CATALOG-NAME PIC S9(4) BINARY VALUE IS 10003.
01 SQL-SQL-COLLATION-SEQ PIC S9(4) BINARY VALUE IS 10004.
01 SQL-SQL-MAX-IDENTIFIER-LEN PIC S9(4) BINARY VALUE IS 10005.

/* SQL-ALTER-TABLE values */
01 SQL-ALTER-TABLE PIC S9(9) BINARY VALUE IS 1.
01 SQL-DROP-COLUMN PIC S9(9) BINARY VALUE IS 2.
01 SQL-ALTER-COLUMN PIC S9(9) BINARY VALUE IS 4.
01 SQL-DROP-CONSTRAINT PIC S9(9) BINARY VALUE IS 8.
01 SQL-DROP-CONSTRAINT PIC S9(9) BINARY VALUE IS 16.

/* SQL-CURSOR-COMMIT-BEHAVIOR values */
01 SQL-CB-DELETE PIC S9(4) BINARY VALUE IS 0.
01 SQL-CB-CLOSE PIC S9(4) BINARY VALUE IS 1.
01 SQL-CB-PRESERVE PIC S9(4) BINARY VALUE IS 2.

/* SQL-DESCRIBE-PARAMETER values */
01 SQL-FETCH-NEXT PIC S9(9) BINARY VALUE IS 1.
01 SQL-FETCH-FIRST PIC S9(9) BINARY VALUE IS 2.
01 SQL-FETCH-LAST PIC S9(9) BINARY VALUE IS 4.
01 SQL-FETCH-PRIOR PIC S9(9) BINARY VALUE IS 8.
01 SQL-FETCH-ABSOLUTE PIC S9(9) BINARY VALUE IS 16.
01 SQL-FETCH-RELATIVE PIC S9(9) BINARY VALUE IS 32.

/* SQL-IDENTIFIER-CASE values */
01 SQL-ANY-COLUMN PIC S9(4) BINARY VALUE IS 1.
01 SQL-ANY-ORDER PIC S9(4) BINARY VALUE IS 2.

/* SQL-OJ-CAPABILITIES values */
01 SQL-OJ-LEFT PIC S9(9) BINARY VALUE IS 1.
01 SQL-OJ-RIGHT PIC S9(9) BINARY VALUE IS 2.
01 SQL-OJ-FULL PIC S9(9) BINARY VALUE IS 4.
01 SQL-OJ-NESTED PIC S9(9) BINARY VALUE IS 8.
01 SQL-OJ-NOT-ORDERED PIC S9(9) BINARY VALUE IS 16.
01 SQL-OJ-INNER PIC S9(9) BINARY VALUE IS 32.
01 SQL-OJ-ALL-COMPARISON-OPS PIC S9(9) BINARY VALUE IS 64.

/* SQL-SCROLL-CONCURRENCY values */
01 SQL-SCCO-READ-ONLY PIC S9(9) BINARY VALUE IS 1.
01 SQL-SCCO-LOCK PIC S9(9) BINARY VALUE IS 2.
01 SQL-SCCO-OPT-ROWVER PIC S9(9) BINARY VALUE IS 4.
01 SQL-SCCO-OPT-VALUES PIC S9(9) BINARY VALUE IS 8.

/* SQL-TXN-CAPABLE values */
01 SQL-TC-NONE PIC S9(4) BINARY VALUE IS 1.
01 SQL-TC-DML PIC S9(4) BINARY VALUE IS 2.
01 SQL-TC-ALL PIC S9(4) BINARY VALUE IS 3.
01 SQL-TC-DDL-COMMIT PIC S9(4) BINARY VALUE IS 4.
01 SQL-TC-DDL-IGNORE PIC S9(4) BINARY VALUE IS 5.

/* SQL-TXN-ISOLATION-OPTION values */
01 SQL-TXN-READ-UNCOMMITTED PIC S9(9) BINARY VALUE IS 1.
01 SQL-TXN-READ-COMMITTED PIC S9(9) BINARY VALUE IS 2.
01 SQL-TXN-REPEATABLE-READ PIC S9(9) BINARY VALUE IS 4.
01 SQL-TXN-SERIALIZABLE PIC S9(9) BINARY VALUE IS 8.
This appendix contains three sample functions that use CLI.

- The first example (in Section D.1) creates a table, inserts data into the table, and selects the inserted data.
- The second example (in Section D.2 on page 268) shows interactive *ad hoc* query processing.
- The third example (in Section D.3 on page 272) illustrates providing long parameter data at execute time.

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

### D.1 Create Table, Insert, Select

This example function creates a table, inserts data into the table, and selects the inserted data.

The example illustrates the execution of SQL statement text both using the `Prepare()` and `Execute()` method, and using the `ExecDirect()` method.

The example illustrates both the case where the application uses the automatically-generated descriptors and the case where the application allocates a descriptor of its own and associates this descriptor with the SQL statement.

Code comments include the equivalent statements in embedded SQL to show how embedded SQL operations correspond to CLI function calls.

```c
#include <stddef.h>
#include <sqlcli.h>
#include <string.h>
#define NAMELEN 50

int print_err(SQLSMALLINT handletype, SQLINTEGER handle);

int example1(SQLCHAR *server, SQLCHAR *uid, SQLCHAR *pwd)
{
    SQLHENV henv;
    SQLHDBC hdbc;
    SQLHDESC hdesc;
    SQLHDESC hdesc1;
    SQLHDESC hdesc2;
    SQLHSTMT hstmt;
    SQLINTEGER id;
    SQLSMALLINT idind;
    SQLCHAR name[NAMELEN+1];
    SQLINTEGER namelen;
    SQLSMALLINT nameind;

    /* EXEC SQL CONNECT TO :server USER :uid USING :pwd; */

    /* allocate an environment handle */
```
Create Table, Insert, Select

```c
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
/* allocate a connection handle */
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
/* connect to database */
if (SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS)
      != SQL_SUCCESS)
    return(print_err(SQL_HANDLE_DBC, hdbc));
/* allocate a statement handle */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

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

  /* execute the CREATE TABLE statement */
  if (SQLExecDirect(hstmt, create, SQL_NTS) != SQL_SUCCESS)
    return(print_err(SQL_HANDLE_STMT, hstmt));
}

EXEC SQL COMMIT WORK; /*
/* commit CREATE TABLE */
SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT); /*
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(SQL_HANDLE_STMT, hstmt));
  /* application parameter descriptor */
  SQLGetStmtAttr(hstmt, SQL_ATTR_APP_PARAM_DESC, &hdesc1, 0L,
                 (SQLINTEGER *)NULL);
  SQLSetDescRec(hdesc1, 1, SQL_INTEGER, 0, 0, 0,
                (SQLPOINTER)&id, (SQLINTEGER *)NULL, (SQLSMALLINT *)NULL);
  SQLSetDescRec(hdesc1, 2, SQL_CHAR, 0, NAMELEN, 0, 0,
                (SQLPOINTER)name, (SQLINTEGER *)NULL, (SQLSMALLINT *)NULL);
  /* implementation parameter descriptor */
  SQLGetStmtAttr(hstmt, SQL_ATTR_IMP_PARAM_DESC, &hdesc2, 0L,
                 (SQLINTEGER *)NULL);
  SQLSetDescRec(hdesc2, 1, SQL_INTEGER, 0, 0, 0,
                (SQLPOINTER)NULL, (SQLINTEGER *)NULL, (SQLSMALLINT *)NULL);
  SQLSetDescRec(hdesc2, 2, SQL_VARCHAR, 0, NAMELEN, 0, 0,
                (SQLPOINTER)NULL, (SQLINTEGER *)NULL, (SQLSMALLINT *)NULL);

  /* assign parameter values and execute the INSERT */
  id=500;
  (void)strcpy(name, "Babbage" );
  if (SQLExecute(hstmt) != SQL_SUCCESS)
    return(print_err(SQL_HANDLE_STMT, hstmt));
}
EXEC SQL COMMIT WORK; /*
/* commit inserts */
SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT); /*
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(SQL_HANDLE_STMT, hstmt));
}

/* EXEC SQL FETCH c1 INTO :id, :name; */
/* this time, explicitly allocate an application row descriptor */
SQLAllocHandle(SQL_HANDLE_DESC, hdbc, &hdesc);
SQLSetDescRec(hdesc, 1, SQL_INTEGER, 0, 0L, 0, 0,
              (SQLPOINTER)&id, (SQLINTEGER *)&idind);
SQLSetDescRec(hdesc, 2, SQL_CHAR, 0, NAMELEN, 0, 0,
              (SQLPOINTER)name, (SQLINTEGER *)&namelen, (SQLSMALLINT *)&nameind);
/* associate descriptor with statement handle */
SQLSetStmtAttr(hstmt, SQL_ATTR_APP_ROW_DESC, &hdesc, 0);
/* execute the fetch */
SQLFetch(hstmt);

/* EXEC SQL COMMIT WORK; */
/* commit the transaction */
SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT);

/* EXEC SQL CLOSE c1; */
SQLCloseCursor(hstmt);
/* free the statement handle */
SQLFreeHandle(SQL_HANDLE_STMT, hstmt);

/* EXEC SQL DISCONNECT; */
/* Disconnect from the database. This frees the descriptor handles. */
SQLDisconnect(hdbc);
/* free connection handle */
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
/* free environment handle */
SQLFreeHandle(SQL_HANDLE_ENV, henv);
return(0);
}
D.2 Interactive Query

The following sample function, based on the flowchart in Figure 4-2 on page 43, uses the concise CLI functions to interactively execute an SQL statement supplied as an argument. For an overview of the concise CLI functions, see Chapter 6.

In the case where the user types a cursor-specification, the function fetches and displays all rows of the result set.

This example illustrates the use of GetDiagField() to identify the type of SQL statement executed and, for SQL statements where the row count is defined on all implementations, the use of GetDiagField() to obtain the row count.

/*
 * Sample program - uses concise CLI functions to execute
 * interactively an ad-hoc statement.
 */
#include <stddef.h>
#include <sqlcli.h>
#include <string.h>
#include <stdlib.h>
#define MAXCOLS 100
#define max(a,b) (a>b?a:b)

int print_err(SQLSMALLINT handletype, SQLINTEGER handle);
int build_indicator_message(SQLCHAR *errmsg, SQLPOINTER *data,
    SQLSMALLINT collen, SQLINTEGER *outlen, SQLSMALLINT colnum);
SQLINTEGER display_length(SQLSMALLINT coltype, SQLINTEGER collen,
    SQLCHAR *colname);

example2(SQLCHAR *server, SQLCHAR *uid, SQLCHAR *pwd, SQLCHAR *sqlstr)
{
    int i;
    SQLHENV henv;
    SQLHDBC hdbc;
    SQLHSTMT hstmt;
    SQLCHAR errmsg[256];
    SQLCHAR colname[32];
    SQLSMALLINT coltype;
    SQLSMALLINT colnamelen;
    SQLSMALLINT nullable;
    SQLINTEGER collen[MAXCOLS];
    SQLSMALLINT scale;
    SQLINTEGER outlen[MAXCOLS];
    SQLCHAR *data[MAXCOLS];
    SQLSMALLINT nresultcols;
    SQLINTEGER rowcount;
    SQLINTEGER stmttype;
    SQLRETURN rc;

    /* allocate an environment handle */
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);

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

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

/* allocate a statement handle */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

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

/* see what kind of statement it was */
SQLGetDiagField(SQL_HANDLE_STMT, hstmt, 0, SQL_DIAG_DYNAMIC_FUNCTION_CODE,
(SQLPOINTER)&stmttype, 0, (SQLSMALLINT *)&NULL);

switch (stmttype) {
  /* SELECT statement */
  case SQL_SELECT_CURSOR:
    /* determine number of result columns */
    SQLNumResultCols(hstmt, &nresultcols);
    /* 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 which 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, SQL_CHAR, data[i], collen[i],
                 &outlen[i]);
    }
    /* display result rows */
    while (SQL_SUCCEEDED(rc=SQLFetch(hstmt) )) {
      errmsg[0] = '\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 */
        /* print any truncation messages */
        (void)printf("\n%s", errmsg);
      }
    }
    /* while rows to fetch */
    SQLCloseCursor(hstmt);
    break;
  /* searched DELETE, INSERT or searched UPDATE statement */
  case SQL_DELETE_WHERE:
  case SQL_INSERT:
  case SQL_UPDATE_WHERE:
    /* check rowcount */
    SQLGetDiagField(SQL_HANDLE_STMT, hstmt, 0, SQL_DIAG_ROW_COUNT,
                    (SQLPOINTER)&rowcount, 0, (SQLSMALLINT *)&NULL);
    if (SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT) == SQL_SUCCESS) {
(void) printf("Operation successful\n");
}
else {
 (void) printf("Operation failed\n");
}
(void)printf("%ld rows affected\n", rowcount);
break;

/* other statements */
case SQL_ALTER_TABLE:
case SQL_CREATE_INDEX:
case SQL_CREATE_TABLE:
case SQL_CREATE_VIEW:
case SQL_DROP_INDEX:
case SQL_DROP_TABLE:
case SQL_DROP_VIEW:
case SQL_DYNAMIC_DELETE_CURSOR:
case SQL_DYNAMIC_UPDATE_CURSOR:
case SQL_GRANT:
case SQL_REVOKE:
  if (SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT)
    == SQL_SUCCESS) {
    (void) printf("Operation successful\n");
  }
  else {
    (void) printf("Operation failed\n");
  }
  break;

/* implementation-defined statement */
default:
  (void)printf("Statement type=%ld\n", stmttype);
  break;
}

/* free data buffers */
for (i=0; i<nresultcols; i++) {
  (void)free(data[i]);
}

/* free statement handle */
SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
/* disconnect from database */
SQLDisconnect(hdbc);
/* free connection handle */
SQLFreeHandle(SQL_HANDLE_DBC, dbc);
/* free environment handle */
SQLFreeHandle(SQL_HANDLE_ENV, env);

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 maximum length in characters of char string representation of
no. as: = max(precision) + leading sign + E + 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_DATETIME:
      return(max(SQL_TIMESTAMP_LEN,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, %d\n", coltype);
      return(0);
      break;
  }
}

int build_indicator_message(SQLCHAR *errmsg, SQLPOINTER *data,
  SQLINTEGER collen, SQLINTEGER *outlen, SQLSMALLINT colnum)
{
  if (*outlen == SQL_NULL_DATA) {
    (void)strcpy((char *)data, "NULL");
    *outlen=4;
  }
  else {
    sprintf((char *)errmsg+strlen((char *)errmsg),
      "%d chars truncated, col %d\n", *outlen-collen+1, colnum);
    *outlen=255;
  }
}
D.3 Providing Long Dynamic Arguments at Execute Time

In the following example, an application prepares an SQL statement to insert data into the EMPLOYEE table. The statement contains parameters for the NAME, ID, and PHOTO columns. For each parameter, the application calls `bindParam()` to specify the C and SQL data types of the parameter. It also specifies that the data for the first and third parameters will be passed at execute time, and passes the values 1 and 3 for later retrieval by `ParamData()` . These values will identify which parameter is being processed.

The application calls `GetNextID()` to get the next available employee ID number. It then calls `Execute()` to execute the statement. The `Execute()` function returns [SQL_NEED_DATA] when it needs data for the first and third parameters. The application calls `ParamData()` to retrieve the value it stored with `bindParam()` ; it uses this value to determine which parameter to send data for. For each parameter, the application calls `InitUserData()` to initialise the data routine. It repeatedly calls `GetUserData()` and `PutData()` to get and send the parameter data. Finally, it calls `ParamData()` to indicate it has sent all the data for the parameter and to retrieve the value for the next parameter. After data has been sent for both parameters, `ParamData()` returns SQL_SUCCESS.

For the first parameter, `InitUserData()` does not do anything and `GetUserData()` calls a routine to prompt the user for the employee name. For the third parameter, `InitUserData()` calls a routine to prompt the user for the name of a file containing a bitmap photo of the employee and opens the file. `GetUserData()` retrieves the next MAX_DATA_LEN octets of photo data from the file. After it has retrieved all the photo data, it closes the photo file.

Note that some application routines are omitted for clarity.

```
/*
 * Sample program - uses ParamData and PutData to pass long data at
 * execute time.
 */
#include <sqlcli.h>
#include <stdio.h>
#include <string.h>
#ifndef NULL
#define NULL 0
#endif
#define NAME_LEN 30
#define MAX_FILE_NAME_LEN 256
#define MAX_PHOTO_LEN 32000
#define MAX_DATA_LEN 1024

int print_err(SQLSMALLINT handletype, SQLINTEGER handle);
SQLINTEGER GetNextID();
void InitUserData(SQLSMALLINT sParam, SQLPOINTER InitValue);
SQLSMALLINT GetUserData(SQLPOINTER InitValue, SQLSMALLINT sParam,
SQLCHAR *Data, SQLINTEGER *StrLen_or_Ind);

example3(SQLCHAR *server, SQLCHAR *uid, SQLCHAR* pwd)
{
    SQLHENV henv;
    SQLHDBC hdbc;
    SQLHSTMT hstmt;
    SQLRETURN rc;
    SQLINTEGER NameParamLength, IDLength = 0, PhotoParamLength, StrLen_or_Ind;
    SQLINTEGER ID;
```
Sample C Programs

Providing Long Dynamic Arguments at Execute Time

```c
SQLSMALLINT Param1=1, Param3=3;
SQLPOINTER pToken, InitValue;
SQLCHAR Data[MAX_DATA_LEN];
/* Allocate an environment handle */
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
/* Allocate a connection handle */
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
/* Connect to database */
rc = SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(print_err(SQL_HANDLE_DBC, hdbc));
/* Allocate a statement handle */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
/* Prepare the INSERT statement */
rc = SQLPrepare(hstmt,
    "INSERT INTO EMPLOYEE (NAME, ID, PHOTO) VALUES (?, ?, ?)",
    SQL_NTS);
if (rc == SQL_SUCCESS) {
    /* Bind the parameters. For parameters 1 and 3, pass the */
    /* parameter number in rgbValue instead of a buffer address. */
    SQLBindParam(hstmt, 1, SQL_CHAR, SQL_CHAR,
        NAME_LEN, 0, &Param1, &NameParamLength);
    SQLBindParam(hstmt, 2, SQL_INTEGER, SQL_INTEGER,
        sizeof(ID), 0, &ID, &IDLength);
    SQLBindParam(hstmt, 3, SQL_CHAR, SQL_CHAR,
        MAX_PHOTO_LEN, 0, &Param3, &PhotoParamLength);
    /* Set values so data for parameters 1 and 3 will be passed */
    /* at execution. */
    NameParamLength = PhotoParamLength = SQL_DATA_AT_EXEC;
    ID = GetNextID(); /* Get next available employee ID number. */
    rc = SQLExecute(hstmt);
    /* For data-at-execution parameters, call SQLParamData to get the */
    /* parameter number set by SQLBindParam. Call InitUserData. */
    /* Call GetUserData and SQLPutData repeatedly to get and put all */
    /* data for the parameter. Call SQLParamData to finish processing */
    /* this parameter and start processing the next parameter. */
    while (rc == SQL_NEED_DATA) {
        rc = SQLParamData(hstmt, &pToken);
        if (rc == SQL_NEED_DATA) {
            InitUserData(pToken, InitValue);
            while (GetUserData(InitValue, pToken, Data, &StrLen_or_Ind))
                SQLPutData(hstmt, Data, StrLen_or_Ind);
        }
    }
/* Commit the transaction. */
```
Providing Long Dynamic Arguments at Execute Time

Sample C Programs

```c
SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT);
/* Free the Statement handle. */
SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
/* Disconnect from the database */
SQLDisconnect(hdbc);
/* Free the Connection handle */
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
/* Free the Environment handle */
SQLFreeEnv(SQL_HANDLE_ENV, henv);
return(0);
}

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

void InitUserData(SQLSMALLINT sParam, SQLPOINTER InitValue)
{
    SQLCHAR szPhotoFile[MAX_FILE_NAME_LEN];

    switch sParam {
        case 3:
            /* Prompt user for bitmap file containing employee photo. */
            /* OpenPhotoFile opens the file and returns the file handle. */

            PromptPhotoFileName(szPhotoFile);
            OpenPhotoFile(szPhotoFile, (FILE *)InitValue);
            break;
    }
}

SQLSMALLINT GetUserData(SQLPOINTER InitValue,
    SQLSMALLINT sParam,
    SQLCHAR *Data,
    SQLINTEGER *StrLen_or_Ind)
{
    switch sParam {
        case 1:
            /* Prompt user for employee name. */

            PromptEmployeeName(Data);
            *StrLen_or_Ind = SQL_NTS;
            return (1);

        case 3:
            /* GetNextPhotoData returns the next piece of photo data and */
            /* the number of octets of data returned (up to MAX_DATA_LEN). */

            Done = GetNextPhotoData((FILE *)InitValue, Data,
                MAX_DATA_LEN, StrLen_or_Ind);
            if (Done) {
```
ClosePhotoFile((FILE *)InitValue);
return(1);
}
return(0);
}

// Data Management: SQL Call Level Interface (CLI)
This appendix maps CLI functions to and from the embedded SQL specified in the X/Open SQL specification.

The forms shown in this appendix are not necessarily the actual syntax; they are meant only to illustrate classes of syntax that map to one another.

CLI dynamic arguments correspond generally to host variables in embedded SQL.

### E.1 CLI to X/Open Embedded SQL

The following table indicates which of the X/Open SQL specification statements, if any, correspond to the CLI functions. The numbers in [ ] refer to the notes following the table.

<table>
<thead>
<tr>
<th>CLI Function</th>
<th>X/Open SQL Statement</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocConnect()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>AllocEnv()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>AllocHandle()</td>
<td>ALLOCATE DESCRIPTOR</td>
<td>[1]</td>
</tr>
<tr>
<td>AllocStmt()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>BindCol()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>BindParam()</td>
<td>SET DESCRIPTOR</td>
<td></td>
</tr>
<tr>
<td>Cancel()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>CloseCursor()</td>
<td>CLOSE</td>
<td></td>
</tr>
<tr>
<td>ColAttribute()</td>
<td>GET DESCRIPTOR</td>
<td></td>
</tr>
<tr>
<td>Columns()</td>
<td>SELECT...FROM COLUMNS</td>
<td>CL... has additional columns.</td>
</tr>
<tr>
<td>Connect()</td>
<td>CONNECT</td>
<td></td>
</tr>
<tr>
<td>CopyDesc()</td>
<td>DESCRIBE</td>
<td></td>
</tr>
<tr>
<td>DataSources()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>DescribeCol()</td>
<td>GET DESCRIPTOR</td>
<td></td>
</tr>
<tr>
<td>Disconnect()</td>
<td>DISCONNECT</td>
<td></td>
</tr>
<tr>
<td>EndTran()</td>
<td>COMMIT/ROLLBACK</td>
<td></td>
</tr>
<tr>
<td>Error()</td>
<td>GET DIAGNOSTICS</td>
<td></td>
</tr>
<tr>
<td>ExecDirect()</td>
<td>EXECUTE IMMEDIATE</td>
<td>[2, 3]</td>
</tr>
<tr>
<td>Execute()</td>
<td>EXECUTE</td>
<td>[2]</td>
</tr>
<tr>
<td>Fetch()</td>
<td>FETCH</td>
<td></td>
</tr>
<tr>
<td>FetchScroll()</td>
<td>FETCH</td>
<td></td>
</tr>
<tr>
<td>FreeConnect()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>FreeEnv()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>FreeHandle()</td>
<td>DEALLOCATE DESCRIPTOR</td>
<td>[1]</td>
</tr>
<tr>
<td>FreeStmt()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>GetConnectAttr()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>GetConnectOption()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>GetCursorName()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>GetData()</td>
<td>GET DESCRIPTOR ... DATA</td>
<td>Can also get a long column in pieces.</td>
</tr>
<tr>
<td>CLI Function</td>
<td>X/Open SQL Statement</td>
<td>Comments</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>GetDescField()</td>
<td>GET DESCRIPTOR</td>
<td>[4]</td>
</tr>
<tr>
<td>GetDescRec()</td>
<td>GET DESCRIPTOR</td>
<td>[4]</td>
</tr>
<tr>
<td>GetDescField()</td>
<td>GET DIAGNOSTICS</td>
<td></td>
</tr>
<tr>
<td>GetDiagRec()</td>
<td>GET DIAGNOSTICS</td>
<td></td>
</tr>
<tr>
<td>GetEnvAttr()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>GetFunctions()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>GetInfo()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>GetStmtAttr()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>GetStmtOption()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>GetTypeInfo()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>NumResultCols()</td>
<td>GET DESCRIPTOR ... COUNT</td>
<td></td>
</tr>
<tr>
<td>ParamData()</td>
<td>PREPARE</td>
<td></td>
</tr>
<tr>
<td>Prepare()</td>
<td>PREPARE</td>
<td></td>
</tr>
<tr>
<td>PutData()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>RowCount()</td>
<td>GET DIAGNOSTICS</td>
<td></td>
</tr>
<tr>
<td>SetConnectAttr()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>SetConnectOption()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>SetCursorName()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>SetEnvAttr()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>SetDescField()</td>
<td>SET DESCRIPTOR</td>
<td>[4]</td>
</tr>
<tr>
<td>SetDescRec()</td>
<td>SET DESCRIPTOR</td>
<td>[4]</td>
</tr>
<tr>
<td>SetEnvAttr()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>SetParam()</td>
<td>SET DESCRIPTOR</td>
<td></td>
</tr>
<tr>
<td>SetStmtAttr()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>SpecialColumns()</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>Statistics()</td>
<td>SELECT...FROM INDEXES</td>
<td></td>
</tr>
<tr>
<td>Tables()</td>
<td>SELECT...FROM TABLES</td>
<td></td>
</tr>
<tr>
<td>Transact()</td>
<td>COMMIT/ROLLBACK</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

[1] CLI data structures other than descriptors have no analog in embedded SQL. Embedded SQL client data structures other than descriptors are allocated and freed implicitly.

[2] If the SQL text being prepared or executed is a *cursor-specification*, this function maps to DECLARE CURSOR followed by OPEN.

[3] If the text being prepared contains dynamic parameter markers and it is not a *cursor-specification*, this function maps to PREPARE followed by EXECUTE.

[4] In CLI, descriptors are also used to specify dynamic arguments.
### E.2 X/Open Embedded SQL to CLI

This table describes which CLI functions, if any, correspond to the embedded SQL statements in the X/Open SQL specification. The numbers in [ ] refer to the notes following the table.

<table>
<thead>
<tr>
<th>X/Open SQL Statement</th>
<th>CLI Function</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cursor Declarations</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECLARE CURSOR</td>
<td>none</td>
<td>[1]</td>
</tr>
<tr>
<td>Dynamic DECLARE CURSOR</td>
<td>none</td>
<td>[1]</td>
</tr>
<tr>
<td><strong>Data Definition Statements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CREATE INDEX</td>
<td>none</td>
<td>[2]</td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>none</td>
<td>[2]</td>
</tr>
<tr>
<td>CREATE VIEW</td>
<td>none</td>
<td>[2]</td>
</tr>
<tr>
<td>DROP INDEX</td>
<td>none</td>
<td>[2]</td>
</tr>
<tr>
<td>DROP TABLE</td>
<td>none</td>
<td>[2]</td>
</tr>
<tr>
<td>DROP VIEW</td>
<td>none</td>
<td>[2]</td>
</tr>
<tr>
<td>GRANT</td>
<td>none</td>
<td>[2]</td>
</tr>
<tr>
<td>REVOKE</td>
<td>none</td>
<td>[2]</td>
</tr>
<tr>
<td><strong>Data Manipulation Statements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOSE</td>
<td>CloseCursor()</td>
<td>[1,3]</td>
</tr>
<tr>
<td>Positioned DELETE</td>
<td>none</td>
<td>[2]</td>
</tr>
<tr>
<td>Searched DELETE</td>
<td>none</td>
<td>[2]</td>
</tr>
<tr>
<td>FETCH</td>
<td>Fetch()</td>
<td>[2]</td>
</tr>
<tr>
<td>INSERT</td>
<td>none</td>
<td>[2]</td>
</tr>
<tr>
<td>OPEN</td>
<td>none</td>
<td>[1]</td>
</tr>
<tr>
<td>SELECT</td>
<td>not supported</td>
<td>[5]</td>
</tr>
<tr>
<td>Positioned UPDATE</td>
<td>none</td>
<td>[1,3]</td>
</tr>
<tr>
<td>Searched UPDATE</td>
<td>none</td>
<td>[2]</td>
</tr>
<tr>
<td><strong>Dynamic SQL Statements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALLOCATE DESCRIPTOR</td>
<td>AllocHandle(Descr.)</td>
<td></td>
</tr>
<tr>
<td>Dynamic CLOSE</td>
<td>CloseCursor()</td>
<td></td>
</tr>
<tr>
<td>DEALLOCATE DESCRIPTOR</td>
<td>FreeHandle(Descr.)</td>
<td></td>
</tr>
<tr>
<td>Dynamic Positioned DELETE</td>
<td>ExecDirect()</td>
<td>[2]</td>
</tr>
<tr>
<td>DESCRIBE</td>
<td>CopyDesc()</td>
<td></td>
</tr>
<tr>
<td>EXECUTE</td>
<td>Execute()</td>
<td></td>
</tr>
<tr>
<td>EXECUTE IMMEDIATE</td>
<td>ExecDirect()</td>
<td></td>
</tr>
<tr>
<td>Dynamic FETCH</td>
<td>Fetch()</td>
<td></td>
</tr>
<tr>
<td>GET DESCRIPTOR</td>
<td>GetDescField()</td>
<td>COUNT form or single field.</td>
</tr>
<tr>
<td></td>
<td>GetDescRec()</td>
<td>Entire item descriptor area.</td>
</tr>
<tr>
<td>Dynamic OPEN</td>
<td>Execute()</td>
<td></td>
</tr>
<tr>
<td>Prepare</td>
<td>Prepare()</td>
<td></td>
</tr>
<tr>
<td>SET DESCRIPTOR</td>
<td>SetDescField()</td>
<td>COUNT form or single field.</td>
</tr>
<tr>
<td></td>
<td>SetDescRec()</td>
<td>Entire item descriptor area.</td>
</tr>
<tr>
<td>Dynamic Positioned UPDATE</td>
<td>ExecDirect()</td>
<td>[2]</td>
</tr>
<tr>
<td>X/Open SQL Statement</td>
<td>CLI Function</td>
<td>Comments</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td><strong>Transaction Control Statements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td>EndTran(SQL_COMMIT)</td>
<td></td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>EndTran(SQL_ROLLBACK)</td>
<td></td>
</tr>
<tr>
<td><strong>Connection Statements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONNECT</td>
<td>Connect()</td>
<td></td>
</tr>
<tr>
<td>DISCONNECT</td>
<td>Disconnect()</td>
<td></td>
</tr>
<tr>
<td>SET CONNECTION</td>
<td>none</td>
<td>CLI does not support DISCONNECT ALL.</td>
</tr>
<tr>
<td>[4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Diagnostics Statement</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GET DIAGNOSTICS</td>
<td>GetDiagField()</td>
<td>Gets any single diagnostic field; extensible.</td>
</tr>
<tr>
<td></td>
<td>GetDiagRec()</td>
<td>Gets SQLSTATE, native code and message text.</td>
</tr>
</tbody>
</table>

**Notes:**

[1] The CLI implementation implicitly declares and opens a cursor if the application passes a cursor-specification to Execute() or ExecDirect(). To set a name for the cursor, use SetCursorName(). To retrieve a cursor name, use GetCursorName().

[2] CLI does not provide direct equivalents for these SQL statements, but they can be prepared by Prepare() and then executed by Execute(), or executed directly using ExecDirect().

[3] The application obtains the implementation-generated cursor name by first calling GetCursorName().

[4] The CLI implementation tracks which connection is current and switches to a different connection if the application uses a handle specifying or implying a different connection.

[5] The Columns() and Tables() functions are equivalent to a cursor-specification on a system view.
This Glossary is intended to assist understanding and is not a substantive part of this specification.

**active connection**
A current or dormant connection on which any SQL statement has been successfully executed during the current transaction.

**application**
The end-user program written to operate on a database. This term is used to distinguish that program from the implementation (the CLI product) when discussing the calling interface.

**application row buffer**
An area provided by the application into which the implementation returns database data.

**application parameter buffer**
An area provided by the application into which the application places dynamic arguments.

**application parameter descriptor**
A descriptor that describes the elements of the application parameter buffer.

**application row descriptor**
A descriptor that describes the elements of the application row buffer.

**argument**
A value the application passes into a function when invoking that function.

**authentication string**
A string that is used to validate the user identifier.

**base table**
A table that is not a view. See also *View*.

**binding**
The act of associating a parameter, variable or other placeholder with an actual value.

**bound parameter**
A dynamic parameter whose value has been specified by association with a bound descriptor record. See Section 3.5.6 on page 37.

**bound column**
A column reference that is described by a descriptor record in the associated bound descriptor. See Section 3.5.6 on page 37.

**by reference**
An argument that will reflect changes made within the body of that function outside the scope of that function.

**by value**
An argument that will not reflect changes made within the body of the function outside the scope of that function.

**byte**
An octet. This specification avoids the term *byte*. It uses *octet* to refer to an amount of storage, and *character* to refer to a component of character-string data independent of the amount of storage.
character
A component of character-string data, which requires one or more octets of storage. See Section 2.3.4 on page 18.

client
The component of a database application that requests data for processing.

column attribute
An item of information about a column, such as the column length.

current connection
The connection on which a function operates.

cursor
A movable pointer into a result set.

cursor-specification
Throughout this specification, cursor-specification refers to the entire syntax of the cursor-specification (SELECT statement) defined in the X/Open SQL specification. This does not include the SELECT...INTO syntax of the dynamic FETCH statement of embedded SQL.

data-at-execute parameter
A bound dynamic parameter whose fields are set to indicate that the application will provide the dynamic argument when the statement is executed (see Section 4.3.2 on page 48).

database handle
A handle that identifies the context of a connection to a server. All data that is pertinent to the server is associated with this handle.

deadlock
A condition under which an activity may not proceed because it is dependent on exclusive resources that are owned by some other activity, which in turn is dependent on exclusive resources in use by the original activity.

deferred field
A field of a descriptor that the application specifies at one time and for which the implementation retrieves or supplies the value at a later time.

delimited identifier
A double-quoted string that identifies an object within a database. All text within the double quotes is assumed to be part of the identifier and is interpreted literally.

deprecated feature
A term used in discussing X/Open-compliance; see the full discussion of compliance in Section 1.5 on page 8.

descriptor
A data structure inside the implementation that contains information about one or more columns or dynamic arguments.

dormant connection
A connection that the application has established previously and has not disconnected, but is not using in the current database activity.

dynamic SQL
An execution model in which the actual SQL statement is not known until run time.
**dynamic parameter**
A position in an SQL statement that represents a literal value the application must define before executing the statement.

**dynamic argument**
The value that is bound to a dynamic parameter of an SQL statement.

**embedded SQL**
An execution model in which SQL statements are included as part of the compilable source code of a program.

**environment handle**
A handle that identifies the global context for database access. All data that is pertinent to all objects in the environment is associated with this handle.

**error condition**
A condition in the execution of a function that indicates failure to perform the requested operation. Contrast *warning condition*.

**error status**
A return code from a function that reports an *error condition*.

**escape character**
A character that disables special interpretation of the character that follows it in a pattern-value. See *Qualification by Pattern-values* on page 55.

**function**
One of the CLI functions that this specification defines, unless a different meaning is evident in context.

**handle**
A variable that refers to a data structure within the implementation. Handles hide implementation-defined data structures.

**identifier**
A text string, whose content is subject to certain rules, that identifies an object in a database, such as a cursor, table or user.

**implementation**
The X/Open-compliant CLI product. This term is used to distinguish the CLI product from the application (the end-user program) when discussing the calling interface.

**implementation-defined**
A term used in discussing X/Open-compliance; see the full discussion of compliance in Section 1.5 on page 8.

**implementation parameter buffer**
The area in the implementation that contains the dynamic arguments in a form in which they will be passed to the database.

**implementation parameter descriptor**
A descriptor that describes the elements of the implementation parameter buffer.

**implementation row buffer**
The area in the implementation that contains data read from the database.

**implementation row descriptor**
A descriptor that describes the elements of the implementation row buffer.
implicit descriptor
   A descriptor that the implementation creates when the application allocates a statement handle. Implicit descriptors are associated with exactly one statement handle and may not be re-associated with any other statement handle. An implicit descriptor is destroyed when the application frees its statement handle.

metadata
   The definitions of all active base tables, viewed tables, indexes, privileges and user names in a database.

metadata function
   One of the CLI functions that query the metadata (the structure of a database) rather than the data.

octet
   A discrete unit of memory accommodating at least 8 bits. An octet is the unit this specification uses to measure the memory requirements of a character.

open connection
   A connection that is either active or dormant.

optional feature
   A term used in discussing X/Open-compliance; see the full discussion of compliance in Section 1.5 on page 8.

parameter
   The abstract representation of the function arguments.

parameter number
   The ordinal position of a dynamic parameter.

pending transaction
   A transaction that has not been completed.

positioned update
   An UPDATE statement in SQL that operates on data identified by a cursor.

positioned delete
   A DELETE statement in SQL that operates on data identified by a cursor.

precision
   The number of bits or digits representing a number.

pseudo-column
   A column in a table other than those specified by an application using the CREATE TABLE or ALTER TABLE statements of embedded SQL. Pseudo-columns are undefined by this document. An example of a pseudo-column is a row identifier.

qualification
   An application’s optional use of a schema name and, on some implementations, a catalog name, to make unique the reference to a database object. See Section 2.4.3 on page 22.

RDA
   (Remote Database Access) A mechanism by which clients and servers communicate to carry out application database requests; specified in the X/Open RDA specification.

restricted handle
   A handle returned by AllocHandle() that the application can use only to obtain diagnostic information or to free the handle.
result set
A derived table produced as the result of SQL statement or CLI function execution. It is this
table from which rows are fetched.

scale
The number of bits or digits representing the fractional part of a number.

schema
A collection of related objects in a database.

SELECT statement
See cursor-specification.

server
The component of a database application that provides data on request.

skeleton environment
An environment referenced by a restricted handle. The only valid use of this environment is
to obtain diagnostic information.

SQL
(Structured Query Language) A standard language for codifying database queries and
updates as text, specified in the X/Open SQL specification.

statement handle
A handle that identifies the context of a single SQL statement.

status record
A record of some error or warning event maintained by the implementation. The
application can retrieve elements of the status record using GetDiagRec() or GetDiagField().

stored routine
A routine (procedure or function) that resides on the server, which the client can invoke.

system view
A view, defined in the X/Open SQL specification, that provides metadata.

transaction
An action or series of actions that are atomic; that is, such that either all actions take
permanent effect or none do. Transactions are defined more precisely, and their effects are
discussed, in the X/Open Distributed Transaction Processing Reference Model.

two-phase commit
A protocol for ensuring that a transaction is atomic (see transaction) among all sites (for
example, servers) where the actions take place. For more information, see the X/Open
Distributed Transaction Processing Reference Model.

type convertible
Data types whose values may be assigned to objects of another data type.

unbound column
A column value whose associated descriptor record is not bound.

unbound parameter
A dynamic parameter reference whose dynamic argument value has yet to be supplied by
associating it with a bound descriptor record.

undefined
A term used in discussing X/Open-compliance; see the full discussion of compliance in
Section 1.5 on page 8.
**user identifier**
The string that represents the security identity under whose auspices the connection is established.

**view**
A derived table with a name that is defined in terms of other tables. Conceptually, a base table is a table that is stored; a view is a table that is computed.

**warning condition**
A condition in the execution of a function that does not imply a failure to perform the requested operation but provides the application with additional information. Contrast error condition.

**warning status**
A return code from a function that reports a warning condition.
Index

binding columns ..............................................50
binding language ............................................6
bindings, parameter type .................................15
BindParam() ..................................................100
overview .....................................................81
blank padding ...............................................61
bound column ..............................................37, 50, 281
bound parameter ............................................281
  persistence across transactions .......................63
branding
  information on host language support ...............6
buffer size ....................................................19
buffer type
  integer values for .......................................58
by reference ...............................................281
by value ......................................................281
by-reference variant ......................................11
by-reference, by-value
  example on manual page ................................87
by-value
  null pointer as output length .......................21
by-value variant ..........................................11
byte ..........................................................281
C
  application buffer types ..............................58
  example on manual page ................................87
  examples use by-value variant .......................11
  header file ...............................................247
  sample program ..........................................265
Call Level Interface ........................................1
calling sequence .........................................73
Cancel
  not included in state tables .........................239
  result not guaranteed ..................................105
cancel statement .........................................75
Cancel() .....................................................105
  causing function to report error ....................75
  effects on status records .............................69
  overview ...................................................49
capitalisation of symbolic names ......................14
case-sensitivity ..........................................21
case-sensitivity of symbolic names ..................14
catalog
  that does not have a name .............................23
catalog name ..............................................22
catalog-name
  syntax .....................................................23
caution
  on use of SQL_DEFAULT ...............................61
changing connection
  failure from pending results .......................73
changing dynamic parameters ........................47
character ...................................................18, 282
character data
  null termination .........................................20
  choice of language .......................................6
class code in SQLSTATE ....................................69
CLI ..........................................................1
  development of ..........................................1
  client .......................................................22, 282
  client/server architecture ...........................1
CloseCursor() ..............................................107
COBOL
  application buffer type ................................59
  availability of by-value ...............................11
  example on manual page ................................87
  examples use by-reference variant ..................11
  header file ...............................................258
  hyphen instead of underscore .......................14
  naming variation .......................................14
  return value parameter ...............................67
ColAttribute() ............................................108
overview ...................................................85
column
  bound .......................................................50
  number of results .......................................50
  retrieving value in pieces ............................51
column attribute ..........................................282
column binding
  persistence across transactions .....................63
column data
  partial return by GetData() .........................156
  column name .............................................21
  using before executing statement ..................44
Columns() ...............................................111
  omitted from ISO CLI ...................................115
  overview ...................................................53
  zero-length strings valid .............................75
COLUMN_NAME
  most efficient name ....................................53
  combinations of SQL/application data types ..........60
combined input/output parameters not used ..........18
COMMIT
  not preparable ..........................................45
  commit (see also complete) .........................63
communication error ....................................73
compatibility
  of buffer data type ....................................60
  compilation of embedded SQL .........................1
  compilation of SQL text ................................44
completion
  closes open cursor .....................................63
In FreeEnv( ) .........................................................143
In FreeStmt( ) .........................................................146
In GetConnectOption() ........................................150
In GetStmtOption() ................................................185
In RowCount( ) .......................................................198
In SetConnectOption() ...........................................201
In SetParam() .........................................................213
In SetStmtOption() ................................................216
In Transact( ) .........................................................229
Deadlock ...............................................................282
deallocating handle ................................................26
decimal point truncation on either side .........................60
DECLARE CURSOR
unnecessary in CLI ..................................................38
DEFAULT server .....................................................116
default transfer .....................................................60
deferred allocation ...................................................34
deferred field ..........................................................282
deferred parameter ...................................................11
defining data structure ..............................................26
definitions of database, fetching ................................53
DELETE footnote in state table .........................................243
number of rows affected ..........................................72
delete use of cursor .....................................................38
delimited identifier ....................................................21, 282
delimiting transaction ...................................................6
demarcating transaction ..............................................128, 230
demarcation of transactions .......................................64
deprecated feature .....................................................282
Depreciated features ......................................................8
deprecated function overview .........................................85
deprecated functions .....................................................83
DESCRIBE INPUT ......................................................47
optional feature ..........................................................8
DescribeCol() ..........................................................123
overview .................................................................81
DescribeStmt( ) overview ..................................................50
descriptor .................................................................32, 282
and SQL_DEFAULT ....................................................60
consistency of fields ....................................................75
relation to handle .......................................................3
state transition ..........................................................245
when allocated ..........................................................34
descriptor handle obtaining .........................................35
obtaining as statement attribute ......................................34
destructive read in Error( ) ...........................................86
destructive write of diagnostic .........................................69
development of CLI .....................................................1
diagnostic .................................................................67
general .................................................................73
number of records .....................................................69
on manual page ..........................................................87
sequence .................................................................70
status record ............................................................69
diagnostic information ...................................................69
diagnostics area relation to handle .......................................3
diagnostics statement executed by client .........................22
direct invocation
  CLI cursor based on ..................................................3
direction of parameter ................................................18
Disconnect() causing subsequent statement to fail .............126
disconnection ............................................................73
discrepancy versus ISO resolving .........................................5
distributed transaction processing ..................................126
rollback error ............................................................74
domain of use of handle ..............................................27
dormant connection .....................................................29, 282
implicit activation .......................................................73
double quote .............................................................21
DTP .................................................................2, 65, 128, 230
rollback error ............................................................74
dynamic argument .....................................................283
mapping to host variable .............................................277
specifying value .........................................................46
dynamic parameter .....................................................283
defining value for .......................................................44
each must be defined .....................................................46
persistence across transactions .......................................63
persists when statement reused .....................................30
dynamic parameter marker .........................................46
dynamic SQL .............................................................1, 282
  CLI implemented on ...................................................4
effect all-or-nothing .....................................................63
  of transaction completion .........................................63
efficient column name ...................................................53
embedded SQL ..........................................................283
  CLI implemented on ...................................................4
  conformance ............................................................6
ending transaction .....................................................63
EndTran() ............................................................127

Index
Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>protection of global memory unnecessary</td>
<td>26</td>
</tr>
<tr>
<td>prototype function calls</td>
<td>87</td>
</tr>
<tr>
<td>pseudo-column</td>
<td>33, 84</td>
</tr>
<tr>
<td>PutData()</td>
<td>196</td>
</tr>
<tr>
<td>qualification</td>
<td>284</td>
</tr>
<tr>
<td>by catalog/schema name</td>
<td>22</td>
</tr>
<tr>
<td>of metadata query, overview</td>
<td>54</td>
</tr>
<tr>
<td>query, interactive</td>
<td>268</td>
</tr>
<tr>
<td>example program</td>
<td>268</td>
</tr>
<tr>
<td>question mark</td>
<td>46</td>
</tr>
<tr>
<td>ranking of diagnostic records</td>
<td>70</td>
</tr>
<tr>
<td>rationale for handle</td>
<td>26</td>
</tr>
<tr>
<td>RDA</td>
<td>284</td>
</tr>
<tr>
<td>record, status</td>
<td>69</td>
</tr>
<tr>
<td>reference manual page</td>
<td>87</td>
</tr>
<tr>
<td>reference, by</td>
<td>11</td>
</tr>
<tr>
<td>relation to other X/Open documents</td>
<td>4</td>
</tr>
<tr>
<td>relation to standards</td>
<td>5</td>
</tr>
<tr>
<td>relation to the X/Open SQL specification</td>
<td>3</td>
</tr>
<tr>
<td>remote database</td>
<td>22</td>
</tr>
<tr>
<td>repeated execution</td>
<td>30</td>
</tr>
<tr>
<td>faster</td>
<td>44</td>
</tr>
<tr>
<td>reserved keyword in identifier</td>
<td>21</td>
</tr>
<tr>
<td>restricted handle</td>
<td>93, 284</td>
</tr>
<tr>
<td>effect on state tables</td>
<td>240</td>
</tr>
<tr>
<td>restriction on identifier</td>
<td>21</td>
</tr>
<tr>
<td>result column</td>
<td></td>
</tr>
<tr>
<td>number of</td>
<td>50</td>
</tr>
<tr>
<td>result set</td>
<td>50, 285</td>
</tr>
<tr>
<td>implicit cursor</td>
<td>38</td>
</tr>
<tr>
<td>result set of metadata function</td>
<td>53</td>
</tr>
<tr>
<td>result value</td>
<td></td>
</tr>
<tr>
<td>recorded in statement handle</td>
<td>30</td>
</tr>
<tr>
<td>return value</td>
<td>67</td>
</tr>
<tr>
<td>in COBOL</td>
<td>67</td>
</tr>
<tr>
<td>reuse of statement handle</td>
<td>30</td>
</tr>
<tr>
<td>rollback</td>
<td></td>
</tr>
<tr>
<td>from disconnection</td>
<td>73</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td></td>
</tr>
<tr>
<td>not preparable</td>
<td>45</td>
</tr>
<tr>
<td>rollback</td>
<td></td>
</tr>
<tr>
<td>spontaneous report</td>
<td>74</td>
</tr>
<tr>
<td>rollback (see also complete)</td>
<td>63</td>
</tr>
<tr>
<td>rounding</td>
<td></td>
</tr>
<tr>
<td>X/Open SQL</td>
<td>60</td>
</tr>
<tr>
<td>row</td>
<td></td>
</tr>
<tr>
<td>cursor-spec returning multiple</td>
<td>3</td>
</tr>
<tr>
<td>describing with descriptor</td>
<td>32</td>
</tr>
<tr>
<td>row count</td>
<td>72</td>
</tr>
<tr>
<td>obtaining with RowCount()</td>
<td>86</td>
</tr>
<tr>
<td>row descriptor</td>
<td></td>
</tr>
<tr>
<td>and SQL_DEFAULT</td>
<td>60</td>
</tr>
<tr>
<td>RowCount()</td>
<td>198</td>
</tr>
<tr>
<td>overview</td>
<td>86</td>
</tr>
<tr>
<td>run-time scope of handle</td>
<td>27</td>
</tr>
<tr>
<td>sample program</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>265</td>
</tr>
<tr>
<td>SCALE</td>
<td>33</td>
</tr>
<tr>
<td>scale</td>
<td>285</td>
</tr>
<tr>
<td>implicit change to</td>
<td>60</td>
</tr>
<tr>
<td>obtaining with GetDescField()</td>
<td>81</td>
</tr>
<tr>
<td>specifying to control conversion</td>
<td>62</td>
</tr>
<tr>
<td>schema</td>
<td>22, 285</td>
</tr>
<tr>
<td>schema-name</td>
<td>22</td>
</tr>
<tr>
<td>syntax</td>
<td>23</td>
</tr>
<tr>
<td>scope of changes to global data</td>
<td>2</td>
</tr>
<tr>
<td>scope rules for handle</td>
<td>27</td>
</tr>
<tr>
<td>scrollable cursor</td>
<td></td>
</tr>
<tr>
<td>extension to embedded SQL</td>
<td>4</td>
</tr>
<tr>
<td>scrollable cursors</td>
<td></td>
</tr>
<tr>
<td>optional feature</td>
<td>8</td>
</tr>
<tr>
<td>searched DELETE</td>
<td></td>
</tr>
<tr>
<td>number of rows affected</td>
<td>72</td>
</tr>
<tr>
<td>searched UPDATE</td>
<td></td>
</tr>
<tr>
<td>number of rows affected</td>
<td>72</td>
</tr>
<tr>
<td>SELECT statement</td>
<td>285</td>
</tr>
<tr>
<td>selecting row</td>
<td></td>
</tr>
<tr>
<td>example program</td>
<td>265</td>
</tr>
<tr>
<td>semantic conventions</td>
<td>18</td>
</tr>
<tr>
<td>semantic information</td>
<td></td>
</tr>
<tr>
<td>on manual page</td>
<td>87</td>
</tr>
<tr>
<td>sensitive cursor</td>
<td>31</td>
</tr>
<tr>
<td>extension to embedded SQL</td>
<td>4</td>
</tr>
<tr>
<td>optional feature</td>
<td>8</td>
</tr>
<tr>
<td>sequence</td>
<td></td>
</tr>
<tr>
<td>general diagnostic</td>
<td>73</td>
</tr>
<tr>
<td>of completing transactions</td>
<td>127, 229</td>
</tr>
<tr>
<td>of metadata result set columns</td>
<td>53</td>
</tr>
<tr>
<td>of retrieving column data</td>
<td>51</td>
</tr>
<tr>
<td>overview of CLI use</td>
<td>42</td>
</tr>
<tr>
<td>sequence of diagnostic</td>
<td>70</td>
</tr>
<tr>
<td>sequencing error</td>
<td>239</td>
</tr>
<tr>
<td>server</td>
<td>22, 285</td>
</tr>
<tr>
<td>access via connection</td>
<td>29</td>
</tr>
<tr>
<td>benefits of CLI architecture</td>
<td>1</td>
</tr>
<tr>
<td>connection to only one</td>
<td>26</td>
</tr>
<tr>
<td>multiple</td>
<td>29</td>
</tr>
<tr>
<td>SERVER_INFO</td>
<td></td>
</tr>
<tr>
<td>use to determine naming system</td>
<td>22</td>
</tr>
<tr>
<td>set of operations (transaction)</td>
<td>63</td>
</tr>
</tbody>
</table>
Index

SetConnectAttr( ) .................................................... 199
SetConnectOption( ) ............................................. 201
SetCursorName( ) ................................................... 203
overview .......................................................... 44
SetValue( ) ........................................................... 205
SetDescField( ) ............................................................ 35
SetDescField( ) ............................................................ 208
overview .......................................................... 82
SetEnvAttr( ) ............................................................ 211
SetEnvOption( ) ............................................................ 85
SetParam( ) ............................................................ 213
SetStmtAttr( ) ............................................................ 214
SetStmtOption( ) ..................................................... 216
settable attribute ........................................... 30
short function name .......................................... 11
simplest execution method ................................ 45
single data structure active .................................. 26
single execution .................................................. 45
ExecDirect( ) ............................................................ 45
skeleton environment ......................................... 285
effect on state tables ............................................. 240
small systems ......................................................... 74
inability to access handle .................................... 74
source code .......................................................... 1
space padding, none on Fetch( ) ......................... 20
spaces around identifier ...................................... 21
SpecialColumns( ) ................................................... 218
omitted from ISO CLI .............................................. 221
overview .......................................................... 53
specifying buffer type .......................................... 58
specifying dynamic argument .............................. 46
SQL ............................................................... 285
conformance ......................................................... 6
SQL (embedded) .................................................... 1
CLI implemented on ............................................. 4
mapping to/from CLI ............................................. 277
SQL data type ......................................................... 57
integer representation ......................................... 57
SQL descriptor ...................................................... 32
relation to handle ................................................ 3
SQL prefix of cursor name .................................... 38, 152
SQL statement ......................................................... 30
handle ............................................................... 30
identifying ........................................................... 72
persistence across transactions ............................ 63
SQLCLLCBH ........................................................... 258
SQLHDBC ............................................................. 29
state table ........................................................... 241
SQLHDESC ............................................................. 32
state transition ..................................................... 245
SQLHENV ............................................................. 28
state transition ..................................................... 240
SQLHSTMT ............................................................ 30
state table ........................................................... 242
SQLSTATE code on manual page ....................... 87
overview .......................................................... 69
relation to handle ................................................. 3
SQL_ATTR_APP attributes ........................................ 30
SQL_ATTR_CURSOR_SCROLLABLE ............................... 30
SQL_ATTR_CURSOR_SENSITIVITY ............................. 31
SQL_ATTR_IMP attributes ......................................... 31
overview .......................................................... 31
SQL_ATTR_METADATA_FOLDER ................................. 54-55
SQL_ATTR_METADATA_ID ........................................ 31
SQL_CHAR
for specifying string conversion .......................... 62
SQL_DEFAULT ......................................................... 60
cautions on use ................................................... 61
SQL_DIAG_ROW_COUNT
implicit use by RowCount( ) ............................... 86
SQL_DOUBLE
problems with SQL-DEFAULT in COBOL .............. 61
SQL_ERROR ............................................................ 68
more information from Error ......................... 129
more information from GetDiagRec( ) ............... 168
SQL_INSENSITIVE ................................................ 31
SQL_INVALID_HANDLE ........................................... 68, 93
in state tables ..................................................... 239
SQL_MAX_ID_LENGTH ............................................. 21
SQL_NEED_DATA .................................................... 68
SQL_NO_DATA ......................................................... 68
in ExecDirect( ) .................................................... 134
in Execute( ) ....................................................... 136
returned by GetDiagField .................................... 69
SQL_UTC ............................................................. 19
SQL_NULL_DATA ..................................................... 19
SQL_REAL problems with SQL-DEFAULT in COBOL ........ 61
SQL_SEARCH_PATTERN_ESCAPE ............................... 56
SQL_SENSITIVE ..................................................... 31
SQL_SUCCESS ......................................................... 67
SQL_SUCCESS_WITH_INFO ....................................... 67
indicating truncation ........................................... 19-20
more information from Error ......................... 129
more information from GetDiagRec( ) ............... 168
SQL_UNSPECIFIED ................................................ 31
standard relation to ................................................ 5
state specific to connection...........................................3
state table ..................................................................239
metadata function .....................................................53
state transition...........................................................73
statement analysing .....................................................50
identifying type...........................................................72
persistence across transactions........................................63
statement attribute ..........................................................30
statement handle ..........................................................3, 30, 285
implicit activation of connection .......................................73
state table .................................................................242
use of two.......................................................................52
statement processing .....................................................43
statement text
conformance with X/Open SQL ........................................6
static SQL ............................................................................1
Statistics() ................................................................222
omitted from ISO CLI .....................................................225
overview ........................................................................53
status record .................................................................69, 285
sequence ........................................................................70
stored routine ..................................................................285
string conversion to numeric .............................................57
zero-length .....................................................................19
string argument
trimming spaces .............................................................21
string conversion .............................................................62
string termination .............................................................19
string truncation .............................................................20, 62
structure of database, unknown ......................................1
subclass code in SQLSTATE .............................................69
subcode, date/time ...........................................................57
success ............................................................................67
symbolic name
C versus COBOL ...............................................................14
of data type .................................................................16
syntactic characteristics of CLI ......................................11
system view ....................................................................53, 285
system, database .............................................................22
table of functions .............................................................40
table, state ........................................................................239
Tables() .........................................................................226
omitted from ISO CLI .....................................................228
overview ........................................................................53
zero-length strings valid .................................................75
termination of output string .............................................19
thread of control .............................................................65
three-part naming .............................................................22
trailing space ...................................................................21
Transact() ......................................................................229
omitted from ISO CLI .....................................................230
transaction .................................................................285
compliance policy ..............................................................6
demarcation API .............................................................64
effect of completion .........................................................63
identifier ........................................................................65
management .................................................................63
model .................................................................................2
spanning connections .........................................................63
state is part of connection .................................................29
transition error .............................................................73
transition, state .............................................................239
trimming spaces from identifier .....................................21
truncation ......................................................................19-20
avoiding .................................................................51, 230
error versus warning .....................................................60
X/Open SQL .................................................................60
truncation of error text .....................................................20
truncation, string .............................................................62
two-part naming .............................................................22
two-phase commit ...........................................................285
UID.................................................................................33
type obtaining with GetDescField() ................................81
TYPE
validity checked .............................................................75
type convertible .............................................................285
type of SQL statement .................................................72
types, parameter versus SQL ..........................................15
UID
in Connect ...................................................................117
unallocated handle .........................................................239
unbound column .............................................................51, 285
unbound parameter ..........................................................285
Undefined ..........................................................................8
undefined ........................................................................285
unused output parameter ..............................................18
underscore
in pattern-value ............................................................55
replacement character for COBOL ..................................14
uniqueness
of cursor name .............................................................38
of function name ...........................................................11
 unknowable transaction outcome .....................................128, 230
unknown database structure .............................................1
UNNAMED .....................................................................33
unreported data error, SQL_DEFAULT .............................61
unused parameters .........................................................18
UPDATE
footnote in state table .........................................................243
number of rows affected ...........................................72
update
  of global data .......................................................26
  use of cursor ..........................................................38
updating fetched data ..............................................52
upper-case in symbolic names ...................................14
usage overview ..........................................................42
use of data types ..........................................................15
user identification .......................................................117
user identifier ..........................................................286
user-defined-name
  as catalog-name and schema-name ..................23
valid SQL statement .......................................................6
value, by- ..............................................................11
variable-length input parameter .................19
variant
  by-value and by-reference ..................................11
varying use of data types ..........................................60
view .................................................................286
visibility of changes to global data ......................2
warning ...............................................................67
warning condition .....................................................286
warning status ..........................................................286
WHERE CURRENT OF
  footnote in state table .....................................243
whether statement returns columns ...............50
wildcard ..............................................................55
X/Open DTP ......................................................65, 128, 230
X/Open SQL .................................................................1
  authority for SQLSTATE meanings ..................70
  mapping between statements and CLI ......277
X/Open SQL specification
  relation of CLI to ..................................................3
zero buffer size .....................................................19
zero-length string ......................................................19
  as server name ....................................................116
  for null catalog name ........................................23
  implied by null pointer ......................................18