

# Ingres 10.0

---

## OpenAPI User Guide

**INGRES**

ING-10-OAU-02

This Documentation is for the end user's informational purposes only and may be subject to change or withdrawal by Ingres Corporation ("Ingres") at any time. This Documentation is the proprietary information of Ingres and is protected by the copyright laws of the United States and international treaties. It is not distributed under a GPL license. You may make printed or electronic copies of this Documentation provided that such copies are for your own internal use and all Ingres copyright notices and legends are affixed to each reproduced copy.

You may publish or distribute this document, in whole or in part, so long as the document remains unchanged and is disseminated with the applicable Ingres software. Any such publication or distribution must be in the same manner and medium as that used by Ingres, e.g., electronic download via website with the software or on a CD-ROM. Any other use, such as any dissemination of printed copies or use of this documentation, in whole or in part, in another publication, requires the prior written consent from an authorized representative of Ingres.

To the extent permitted by applicable law, INGRES PROVIDES THIS DOCUMENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IN NO EVENT WILL INGRES BE LIABLE TO THE END USER OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USER OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF INGRES IS EXPRESSLY ADVISED OF SUCH LOSS OR DAMAGE.

The manufacturer of this Documentation is Ingres Corporation.

For government users, the Documentation is delivered with "Restricted Rights" as set forth in 48 C.F.R. Section 12.212, 48 C.F.R. Sections 52.227-19(c)(1) and (2) or DFARS Section 252.227-7013 or applicable successor provisions.

Copyright © 2010 Ingres Corporation. All Rights Reserved.

Ingres, OpenROAD, and EDBC are registered trademarks of Ingres Corporation. All other trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

# Contents

---

<b>Chapter 1: Introduction</b>	<b>9</b>
Audience .....	9
Your Support Options .....	9
What Is OpenAPI? .....	10
Common Features of Application Programming Interfaces.....	10
Differences Between OpenAPI and Other Application Programming Interfaces.....	11
OpenAPI Communication .....	11
How OpenAPI Handles Backward Compatibility .....	12
OpenAPI Concepts and Processes .....	12
Parameter Blocks.....	13
How Callback and Closure Work.....	15
How Asynchronous Processing Works .....	15
How Synchronous Processing Works.....	16
Handles.....	16
How Connections are Established and Severed.....	18
Transactions .....	19
How Query Statements Work .....	21
How Data Is Retrieved.....	24
How Cursors Work .....	25
Batch Statement Execution .....	27
Database Events .....	33
SQL Syntax .....	35
Name Server Query Statement Syntax .....	35
Query Parameters.....	36
How Unformatted Data is Handled.....	37
Data Conversion.....	38
Error Handling.....	39
<b>Chapter 2: OpenAPI Function Reference</b>	<b>41</b>
Generic Parameters.....	41
Output Parameters Allocated by OpenAPI .....	43
Function Summary .....	43
OpenAPI Management.....	44
Session Management .....	44
Query Processing.....	45
Transaction Operations.....	47
Miscellaneous.....	48
OpenAPI Functions .....	48

---

IIapi_abort() Function—Abort a Connection .....	49
IIapi_autocommit() Function—Enable or Disable Autocommit Transactions.....	50
IIapi_batch() Function—Execute Batch Statements.....	51
IIapi_cancel() Function—Cancel an Outstanding Query Statement .....	54
IIapi_catchEvent() Function—Retrieve a Database Event .....	56
IIapi_close() Function—End a Query Statement or Database Event Retrieval .....	59
IIapi_commit() Function—Commit a Transaction .....	61
IIapi_connect() Function—Connect to a DBMS Server or Name Server.....	62
IIapi_convertData() Function—Convert Ingres Data Values to Compatible Types Using Default Settings .....	66
IIapi_disconnect() Function—Close a Server Connection.....	68
IIapi_formatData() Function—Convert Ingres Data Values to Compatible Types.....	69
IIapi_getColumns() Function—Return Columns from a Previously Invoked Query Statement or Database Event Retrieval.....	70
IIapi_getColumnInfo() Function—Return Additional Information about Columns.....	74
IIapi_getCopyMap() Function—Return the Data Format of Copy File and Database Table Involved in a Copy Statement .....	75
IIapi_getDescriptor() Function—Communicate Format of Return Data with IIapi_getColumns().	76
IIapi_getErrorInfo() Function—Return Additional Error or User-defined Information.....	77
IIapi_getEvent() Function—Wait for Database Events.....	79
IIapi_getQueryInfo() Function—Return Information about a Query .....	80
IIapi_initialize() Function—Initialize OpenAPI to a Specified Input Version.....	85
IIapi_modifyConnect Function—Send Connection Parameters to Server .....	87
IIapi_position() Function—Position Cursor and Initiate Row Retrieval.....	88
IIapi_prepareCommit() Function—Begin Two-phase Commit of Transaction.....	90
IIapi_putColumns() Function—Send Data to Server to Copy Data from File to Database Table ..	91
IIapi_putParms() Function—Send Query Statement Parameter Values to a Server.....	93
IIapi_query() Function—Begin Query Statement and Allocate Statement Handle .....	95
IIapi_registerXID() Function—Reserve Unique ID for Two-phase Commit Transaction .....	98
IIapi_releaseEnv() Function—Release Resources Associated with Environment Handle .....	99
IIapi_releaseXID() Function—Release Unique ID for Two-phase Commit Transaction .....	100
IIapi_rollback() Function—Roll Back a Transaction.....	101
IIapi_savePoint() Function—Mark Savepoint in a Transaction for Partial Rollback .....	102
IIapi_scroll() Function—Scroll (Position) Cursor and Initiate Row Retrieval .....	103
IIapi_setConnectParam() Function—Assign Connection Parameter and Value to a Connection ..	105
IIapi_setDescriptor() Function—Send Information About Data Format .....	113
IIapi_setEnvParam() Function—Assign an Environment Parameter and Value in Environment Handle .....	114
IIapi_terminate() Function—Terminate OpenAPI .....	123
IIapi_wait() Function—Block Application Control Until Outstanding Operation Completes or User- defined Timeout Expires .....	124
IIapi_xaCommit() Function—Commit an XA Transaction .....	125
IIapi_xaEnd() Function—End an XA Transaction Association.....	126

---

IIapi_xaPrepare() Function—Prepare an XA transaction for Two-Phase Commit.....	128
IIapi_xaRollback() Function—Rollback an XA Transaction.....	129
IIapi_xaStart() Function—Start an XA Transaction.....	130

## **Chapter 3: OpenAPI Data Types 133**

OpenAPI Generic Data Types .....	133
OpenAPI Data Types.....	134
IIAPI_DT_ID Data Type—Describe Data Type of Database Columns and Query Parameters .....	135
IIAPI_QUERYTYPE Data Type—Describe Type of Query Being Invoked .....	136
IIAPI_STATUS Data Type—Describe the Return Status of an OpenAPI Function.....	137
OpenAPI Data Structures .....	137
IIAPI_COPYMAP Data Type—Provide Information on How to Execute the SQL Copy Statement .....	138
IIAPI_DATAVALUE Data Type—Provide Value for OpenAPI Data .....	140
IIAPI_DESCRIPTOR Data Type—Provide Description for OpenAPI Data.....	141
IIAPI_FDATABDESCR Data Type—Describe Column Data in a Copy File .....	143
IIAPI_II_DIS_TRAN_ID Data Type—Identify Distributed Ingres Transaction ID .....	145
IIAPI_II_TRAN_ID Data Type—Identify Local Ingres Transaction ID .....	145
IIAPI_SVR_ERRINFO Data Type—Describe Additional Server Information Associated with Error Messages .....	146
IIAPI_TRAN_ID Data Type—Identify an OpenAPI Transaction .....	147
IIAPI_XA_DIS_TRAN_ID Data Type—Identify a Distributed XA Transaction ID .....	148
IIAPI_XA_TRAN_ID Data Type—Identify an XA Transaction ID .....	149

## **Chapter 4: Accessing a DBMS Using SQL 151**

Mapping of SQL to OpenAPI.....	151
SQL Syntax .....	158
Describe Statement .....	158
Execute Statement .....	158
Declare Statement, Open Cursor Statement .....	159
Cursor Delete Statement .....	159
Cursor Update Statement .....	159
Execute Procedure .....	160
Repeat Queries .....	160
Queries, Parameters, and Query Data Correlation .....	160
Queries and Parameters .....	161
Query Data Correlation.....	162

## **Chapter 5: Accessing the Name Server 165**

Mapping of Name Server Query Statements to OpenAPI .....	165
Name Server Query Statement Syntax.....	166

---

Name Server Query Syntax .....	167
Create Login Statement—Create a Login Definition .....	167
Destroy Login Statement—Destroy a Login Definition .....	168
Create Password Statement—Define an Installation Password .....	168
Create Connection Statements—Create a Connection Data Definition .....	169
Destroy Connection Statement—Destroy a Data Definition .....	170
Show Connection Statement—Display Connection Data Definitions .....	172
Create Attribute Statement—Create an Attribute Data Definition .....	173
Destroy Attribute Statement—Destroy an Attribute Data Definition .....	174
Display Attribute Statement—Display an Attribute Data Definition .....	175
Show Server Statement—Display Servers in the Local Installation .....	175
How to Use ~V Marker in the Name Server Query Text .....	176

## **Chapter 6: Creating an Application with OpenAPI** **177**

How You Can Create an OpenAPI Application .....	177
Header Files .....	177
Library .....	177
Environment Variables .....	177
Sample Code .....	179
How the Synchronous Sample Code Works .....	179
How the Asynchronous Sample Code Works .....	182

## **Chapter 7: Using Repeat Queries with OpenAPI** **185**

Repeat Queries .....	185
How the Repeat Query Protocol Works .....	186
Repeat Query ID .....	186
Compile-time and Runtime IDs .....	187
Query Parameters .....	187
How the ~V Mechanism Works .....	188
Repeat Query Parameters .....	188
Example: Repeat query using the ~V marker .....	189

## **Appendix A: Error Handling** **191**

Error Codes .....	191
SQLSTATE Values and Descriptions .....	194
XA Error Codes .....	198

## **Appendix B: Data Types** **201**

Ingres Data Types .....	201
-------------------------	-----

---

Data Type Descriptions ..... 204

**Index** **211**



# Chapter 1: Introduction

---

This section contains the following topics:

[Audience](#) (see page 9)

[Your Support Options](#) (see page 9)

[What Is OpenAPI?](#) (see page 10)

[How OpenAPI Handles Backward Compatibility](#) (see page 12)

[OpenAPI Concepts and Processes](#) (see page 12)

## Audience

This guide is designed for programmers who want to use Ingres<sup>®</sup> OpenAPI<sup>®</sup> to develop applications. Some chapters assume you are already familiar with:

- Ingres components and SQL programming
- The C programming language

## Your Support Options

Enterprise customers with active maintenance and support contracts have full access to Ingres Support, including telephone support and online use of our call tracking system and knowledge base, Service Desk. For Customer Support contact details, see <http://ingres.com/support/contact.php>.

If you have an active support contract and want to register for access to Service Desk (<https://servicedesk.ingres.com>), use the enrollment form at <http://www.ingres.com/users/register.php>. (Your six-digit Account Number/Site ID is required.)

If you do not have a support agreement for Ingres Corporation products and are interested in purchasing support, contact us at [sales@ingres.com](mailto:sales@ingres.com).

For more information about support options, visit <http://ingres.com/support.php>.

Free support is available from the Ingres Open Source Community. Community members may obtain assistance for Ingres Corporation products by registering with the Ingres Community Site and using the available tools. To register, go to <http://community.ingres.com/forum/> and click Register.

The Community Forums also provide Ingres Open Source Community members the opportunity to ask questions and interact with other community members and Ingres Corporation technical staff. For more information, visit <http://community.ingres.com/forum/index.php>.

## What Is OpenAPI?

Open Application Programming Interface (*OpenAPI*) is a set of C language functions that enable you to create applications for accessing Ingres and non-Ingres databases.

It provides you with an alternative to using embedded SQL, which requires a preprocessor in addition to a C compiler. With OpenAPI, these C functions are called directly with normal function call facilities.

OpenAPI simplifies the task of developing applications when multiple interfaces, protocols, and environments are involved. It does this by providing a single interface for accessing data. You can concentrate on *what* data you want your application to access, rather than *how* it will access it.

OpenAPI provides an asynchronous method of writing applications. All OpenAPI operations are asynchronous in that a function call returns control to the application before its tasks are completed. When the tasks are completed, the function signals completion by invoking a callback function specified by the application. Thus, you can write an application as fully asynchronous, event-driven code. Alternatively, you can write synchronous code by using an OpenAPI feature that enables an application to wait for each OpenAPI function to complete its tasks.

## Common Features of Application Programming Interfaces

OpenAPI is an application programming interface, similar to Microsoft Windows ODBC and the X/Open Company SQL Call Level Interface. Application programming interfaces share the following features:

- A standard set of function calls for accessing a database  
This makes an application programming interface ideally suited for a client/server environment, in which the target database may not be known when the application is built.
- No requirements for host variables or other embedded SQL concepts  
Application developers who are familiar with function calls find an application programming interface straightforward to use.
- Preprocessor independence  
SQL statements are sent to a DBMS Server as input parameters in a function call. Query results are returned to the application as output parameters from the function call.

## Differences Between OpenAPI and Other Application Programming Interfaces

OpenAPI provides comparable functionality to Microsoft Windows ODBC and the X/Open Company SQL Call Level Interface. However, there are a few important differences, such as:

- OpenAPI supports the complete function set of Ingres SQL, including FIPS 127-2 and SQL-92 Entry Level.

ODBC and SQL Call Level Interface support only X/Open SQL—an X/Open standard based on the ANSI SQL. This difference enables applications written with OpenAPI to access Ingres databases more efficiently.

- OpenAPI is tailored to the C programming language.
- OpenAPI uses a callback-driven method for asynchronous programming, whereas ODBC uses a polling method.

SQL Call Level Interface does not provide asynchronous programming.

## OpenAPI Communication

The set of C functions in OpenAPI enable an application to connect to a DBMS Server, execute SQL statements, and retrieve results. It provides support for all Ingres SQL statements. OpenAPI also lets an application connect to the Name Server and execute Name Server query statements.

OpenAPI is an interface that currently resides on top of the Ingres client/server protocol, called the General Communications Architecture (GCA). By using the GCA protocol, OpenAPI can communicate with the following:

- Ingres servers
- Ingres Star
- Ingres Enterprise Access products that provide access to multiple Ingres databases and non-Ingres databases

## Relationship of OpenAPI to Basic Ingres Architecture

## How OpenAPI Handles Backward Compatibility

OpenAPI handles backward compatibility through OpenAPI support levels.

**More information:**

IIapi\_initialize() Function—Initialize OpenAPI to a Specified Input Version (see page 85)

IIapi\_connect() Function—Connect to a DBMS Server or Name Server (see page 62)

## OpenAPI Concepts and Processes

This section introduces the concepts that you should be familiar with before using OpenAPI. It also discusses how OpenAPI functions are used to establish connections and perform server operations.

## Parameter Blocks

The *parameter block* is a C structure that is used for passing information back and forth between an application and OpenAPI. All OpenAPI functions require a parameter block.

The application creates the parameter block and passes it as an argument to the OpenAPI function.

Each parameter block contains input and output parameters:

### **Input parameters**

Contain information sent by the application and needed by OpenAPI to carry out the request.

### **Output parameters**

Are returned from OpenAPI to the application and contain the results needed by the application for status reporting or for making subsequent function calls. Output parameters can be *immediate* output or *delayed* output:

#### **Immediate output parameters**

Contain meaningful values as soon as the OpenAPI function returns.

#### **Delayed output parameters**

Contain no meaningful values until all tasks associated with the OpenAPI function are completed.

Most parameter blocks have a common substructure containing generic parameters. The generic parameters are used to handle asynchronous processing and to communicate the return status of the function.

### Example—parameter block

The following parameter block is allocated before the OpenAPI function `IIapi_query()` is invoked:

```
typedef struct _IIAPI_QUERYPARAM  
{
```

*Generic Parameters (containing input and output parameters):*

```
    IIAPI_GENPARAM    qy_genParm;
```

*Input Parameters:*

```
    II_PTR            qy_connHandle;  
    IIAPI_QUERYTYPE  qy_queryType;  
    II_CHAR           *qy_queryText;  
    II_BOOL           qy_parameters;
```

*Input and Immediate Output Parameters:*

```
    II_PTR            qy_tranHandle;
```

*Immediate Output Parameters:*

```
    II_PTR            qy_stmtHandle;
```

```
} IIAPI_QUERYPARAM;
```

The resources associated with the parameter block must not be freed until the OpenAPI function completes.

### More information:

Generic Parameters (see page 41)

## How Callback and Closure Work

When all tasks associated with an OpenAPI function are completed, OpenAPI notifies the application by means of a callback function. When the callback function is invoked, OpenAPI sends return status and other information to the application in the delayed output parameters. This “other” information is needed by the application to make subsequent function calls.

*Closure* is the means for an application to pass any information it wishes to the callback function.

The application creates the closure parameter and passes it as an input parameter to the parameter block. OpenAPI does not care about the contents of the closure parameter; it simply passes it to the callback function when the function completes.

An asynchronous OpenAPI function always requests a callback function to notify the application when the OpenAPI function tasks are completed. If the application does not provide a callback function, OpenAPI assumes that the application is polling for the function completion.

## How Asynchronous Processing Works

A client/server operation normally involves a request and a response. The client issues the request to a server to start a database operation; in response, the server reports success or failure to the client and returns any relevant data.

When an application submits a request to a remote database, several seconds may elapse before the application receives a response. Under traditional synchronous processing, the application cannot perform any functions while it is waiting for the response. However, under asynchronous processing, the application can issue a request and then perform functions that do not require knowledge about the response. For example, it can update a window in OpenROAD or update a log file. This enables the application to maximize efficiency during the period between a request and response.

When the application completes its non-database operations, it returns control to OpenAPI, using the `IIapi_wait()` function to complete the database request.

### **More information:**

How Synchronous Processing Works (see page 16)

## How Synchronous Processing Works

OpenAPI provides a function for synchronous processing when it is needed. The `IIapi_wait()` function takes control away from the application and gives it to OpenAPI until an outstanding database operation completes or until a user-defined timeout expires. This is useful when the application does not have any functions to perform while the response is being generated; it simply calls `IIapi_wait()` and waits for the response information before proceeding to the next task.

### **More information:**

How Asynchronous Processing Works (see page 15)

## Handles

The handles to these storage areas for OpenAPI object information are returned to the application. The application then uses them in subsequent calls to OpenAPI functions to identify the objects.

An API function may allocate a handle even if the requested action fails (for example, a connect request may fail with an invalid password and still return a connection handle). In these cases, the associated OpenAPI function that releases the handle must still be called to release the resources associated with the handle and the failed request.

## Types of Handles

There are several main types of handles:

### **Environment Handle**

Identifies storage for information about user configuration settings. An application requests an environment handle with the `IIapi_initialize()` function by setting the `in_version` parameter to `IIAPI_VERSION_2` (or higher). The application can then make various user configuration settings using the `IIapi_setEnvParam()` function.

The environment handle configuration settings are used when formatting data using the `IIapi_formatData()` function. The settings are also inherited by connections opened in the context of the environment by providing the environment handle as an input parameter to the `IIapi_connect()` or `IIapi_setConnectParam()` functions. Environment handle resources are released with the `IIapi_releaseEnv()` function.

### **Connection handle**

Identifies storage for information about a specific connection to a database. An application requests a connection handle with the `IIapi_connect()` or the `IIapi_setConnectParam()` functions. The application then specifies the handle whenever it issues requests within the context of the connection. When the application no longer needs the connection, it releases the handle with the `IIapi_disconnect()` function.

### **Transaction handle**

Identifies storage for information about a specific transaction. An application requests a transaction handle with the `IIapi_query()`, `IIapi_autoCommit()`, or `IIapi_xaStart()` function. The application then specifies the handle whenever it issues requests within the context of the transaction. When the application needs to end the transaction, it releases the handle with the `IIapi_commit()`, `IIapi_rollback()`, `IIapi_autoCommit()`, or `IIapi_xaEnd()` function.

### **Statement handle**

Identifies storage for information about a specific query statement. An application requests a statement handle with the `IIapi_query()` function. The application then specifies the handle whenever it issues requests within the context of the statement. When the application no longer needs the statement handle, it releases it with the `IIapi_close()` function.

### **Event handle**

Identifies storage for information about a specific database event retrieval. An application requests an event handle with the `IIapi_catchEvent()` function. The application then specifies this handle whenever it issues requests within the context of the database event registration. When the application no longer needs the event handle, it releases it with the `IIapi_close()` function.

## How Connections are Established and Severed

Before an application can request data from a database, it must establish a dialog, or connection, with a data source—either a relational DBMS Server or the Name Server. It does this by using the `IIapi_connect()` function. The connection handle is the identifier of this connection.

If the application needs to establish connection characteristics, it does so by using `IIapi_setConnectParam()` prior to `IIapi_connect()`.

All activity between the application and the server must be within the context of a connection. Normally, the application controls the duration of the connection. When an application no longer needs to communicate with a server, it severs the connection by using the `IIapi_disconnect()` function. In some error conditions, however, the server severs the connection (`IIapi_disconnect()` must still be called to release the OpenAPI resources associated with the connection). `II api_abort()` can also be used to release the resources associated with a connection, but is only intended for use in recovering from error conditions.

## Transactions

A *transaction* is one or more query statements that make up a logical unit of work. This unit of work is either executed in its entirety, or it is totally or partially rolled back.

With OpenAPI, three types of transactions can occur. They are:

### **Multi-statement transaction**

This type of transaction only affects a single database through a single connection. A transaction is started by `IIapi_query()` when the `qy_tranHandle` parameter is `NULL`. The transaction is committed using `IIapi_commit()` or rolled back using `IIapi_rollback()`.

### **Distributed transaction**

This type of transaction uses the two-phase commit mechanism to ensure that committal of a distributed transaction occurs in all participating databases through multiple connections.

Distributed transactions are identified by a transaction ID handle returned by `IIapi_registerXID()`. This handle is used to start transactions on each participating connection by being passed as the `qy_tranHandle` value when calling `IIapi_query()`. The transaction ends by calling `IIapi_prepareCommit()` for each connection followed by `IIapi_commit()` or `IIapi_rollback()`. The transaction ID handle is freed by calling `IIapi_releaseXID()`.

### **Autocommit transaction**

This type of transaction causes each individual query to be automatically committed when complete. If a cursor is opened, the commit occurs when the cursor is closed.

Autocommit transactions are started by `IIapi_autoCommit()` using the connection handle as input. An autocommit transaction ends by calling `IIapi_autoCommit()` with the autocommit transaction handle as input.

### **XA transaction**

The XA transaction specification is a standardized interface for a distributed transaction processing system. This type of transaction uses a two-phase commit mechanism to ensure that committal of a distributed transaction occurs in all participating databases through multiple connections.

XA transactions are identified by an XA transaction ID. An XA transaction branch is started and associated with a particular connection by calling `IIapi_xaStart()`. The association is dropped by calling `IIapi_xaEnd()`.

### **More information:**

How Transactions Work (see page 20)

## How Transactions Work

The following sections detail how an application begins and ends a transaction, how distributed transactions are used, and how savepoints are used.

### How an Application Begins a Transaction

An application specifies the beginning of a new transaction by calling `IIapi_query()` with a input parameter `qy_tranHandle`. If the parameter is a NULL pointer or is a transaction ID handle created by `IIapi_registerXID()`, a new transaction is begun and a transaction handle is allocated and returned in `qy_tranHandle`. If the `qy_tranHandle` input value is a transaction handle returned by a previous call to `IIapi_query()` or `IIapi_xaStart()`, the query is performed as part of the already-opened transaction. If the `qy_tranHandle` input value is an autocommit transaction handle returned by `IIapi_autoCommit()`, the query is executed and the results are immediately committed by the server.

### How an Application Ends a Transaction

At the end of a transaction, the application calls `IIapi_commit()` or `IIapi_rollback()` before starting another transaction within the connection. `IIapi_commit()` ends the transaction by committing all SQL statements upon completion, thereby guaranteeing that changes to the database are permanent. `IIapi_rollback()` ends the transaction by aborting all query statements being executed within the transaction unless a savepoint handle is specified. For XA transactions, the association between the transaction and connection is dropped by calling `IIapi_xaEnd()`.

### How Distributed Transactions Are Used

If the transaction is distributed, `IIapi_prepareCommit()` must be called for each connection participating in the transaction prior to calling `IIapi_commit()` or `IIapi_rollback()`. For distributed transactions, the resources allocated by `IIapi_registerXID()` are freed by calling `IIapi_releaseXID()` once the transaction has been fully committed or rolled back.

### How Savepoints Are Used

In a multi-statement transaction, savepoints can be defined using the `IIapi_savePoint()` function. `IIapi_savePoint()` allocates a savepoint handle to identify each savepoint, which can be used to perform a partial rollback when calling `IIapi_rollback()`. If a savepoint is specified with `IIapi_rollback()`, only the query statements executed following the savepoint are aborted and the transaction remains active. When a transaction is committed or fully rolled back, all associated savepoint handles are automatically released.

The underlying GCA protocol accepts only one transaction at a time within a connection. Once a transaction is started, the application must use the same transaction within that connection for all query statements until the transaction is committed or rolled back.

### How XA Transactions Are Used

For distributed XA transactions, `IIapi_xaPrepare()` must be called for each transaction branch after calling `Iiapi_xaEnd()`. Each branch can then be committed or rolled back by calling `IIapi_xaCommit()` or `IIapi_xaRollback()`. XA transaction branches can also be completed directly (one-phase) by omitting the call to `IIapi_xaPrepare()`.

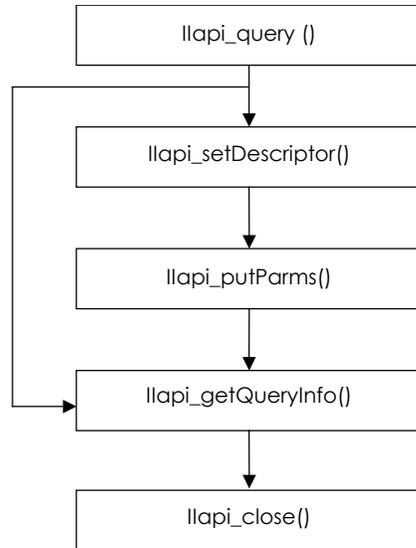
### How Query Statements Work

An application invokes query statements by calling the `IIapi_query()` function and providing the statement type and statement text in input parameters. Normally, the application obtains the results of the query by calling the `IIapi_getQueryInfo()` function and closes the statement by calling the `IIapi_close()` function.

**Note:** Most of the information in this guide pertains to connecting to and operating on a DBMS, but you can work with the Name Server also. The term *server* is used generically, and *query statement* is used generically for either an SQL statement or Name Server query statement.

Data exchange between the application and a server requires two sets of information: data descriptors and data values. Query parameters are passed in calls to the `IIapi_setDescriptor()` and `IIapi_putParms()` functions. Result data returned by query statements is retrieved by calls to the `IIapi_getDescriptor()` and `IIapi_getColumns()` functions.

### Typical Flow of Operations for SQL and Name Server Query Statement Processing



Depending on the query operation, the application may call additional OpenAPI functions. For statements that return data (the SQL select statement, Name Server show statement, database procedures with BYREF, INOUT, or OUT parameters, and row-returning database procedure), and for the SQL copy statement, the order in which an application invokes OpenAPI functions is shown in the figure that follows.

## Order of Invoking OpenAPI Functions

The left side of the diagram shows the sequence of functions for statements that return data. The right side shows the sequence of functions for copying data from a database table to the program (copy into) and copying data from the program into a database table (copy from).

## How Query Statements Are Cancelled

An application can cancel a query statement started with `IIapi_query()` before the statement is fully executed. To cancel a query statement, the application issues `IIapi_cancel()`, specifying the statement handle returned by `IIapi_query()`. `IIapi_cancel()` can be called anytime after `IIapi_query()` has returned with a status of "success." If the query has already been completed, an error is returned to the application stating so.

## How Data Is Retrieved

The following SQL statements are used to retrieve data from a DBMS Server:

### **singleton select**

Retrieves one row from the database. Used when only one result row for the select statement is desirable. If the singleton select tries to retrieve more than one row, an error occurs. This type of select does not use a cursor.

The application issues the following singleton select statement with `IIapi_query()`:

```
IIAPI_QT_SELECT_SINGLETON
```

### **select loop**

Retrieves an unlimited number of result rows. Used when the number of rows in the result set is unknown.

This style of select statement is useful for applications that need to read data to load program data sets or to generate reports. This type of select does not use a cursor.

The application issues the following select loop statement with `IIapi_query()`:

```
IIAPI_QT_SELECT
```

### **open cursor**

Opens a cursor to retrieve rows. Used when it is desirable to use a cursor.

Updatable cursors retrieve one row at a time, permitting updates or deletion of the row addressed by the cursor. Read-only cursors may retrieve more rows at a time, similar to select loops, but permit database operations between retrieval operations.

The application issues the following open cursor statement with `IIapi_query()`:

```
IIAPI_QT_OPEN
```

Regardless of the statement used for retrieving data, the application calls `IIapi_getDescriptor()` after calling `IIapi_query()` to obtain information about the format of the data being returned from the server. It then calls `IIapi_getColumns()` one or more times to retrieve the data. If no long varchar or long byte data types exist in the result set, and it is not an updatable cursor, multiple rows can be returned for each call. `IIapi_getColumns()` returns the "no more data" status once all rows have been returned. When all available data has been retrieved, the application calls `IIapi_close()` to end the process.

## How Cursors Work

Cursors enable an application to process, one at a time, the result rows returned by a select statement. The following SQL statements are used in processing data with a cursor:

### **update (cursor)**

Updates the current row.

### **delete (cursor)**

Deletes the current row.

If a cursor is opened as updatable (the default), the application can update or delete the row referenced by the cursor. If the cursor is opened as read-only, the application can read the data but cannot update or delete it.

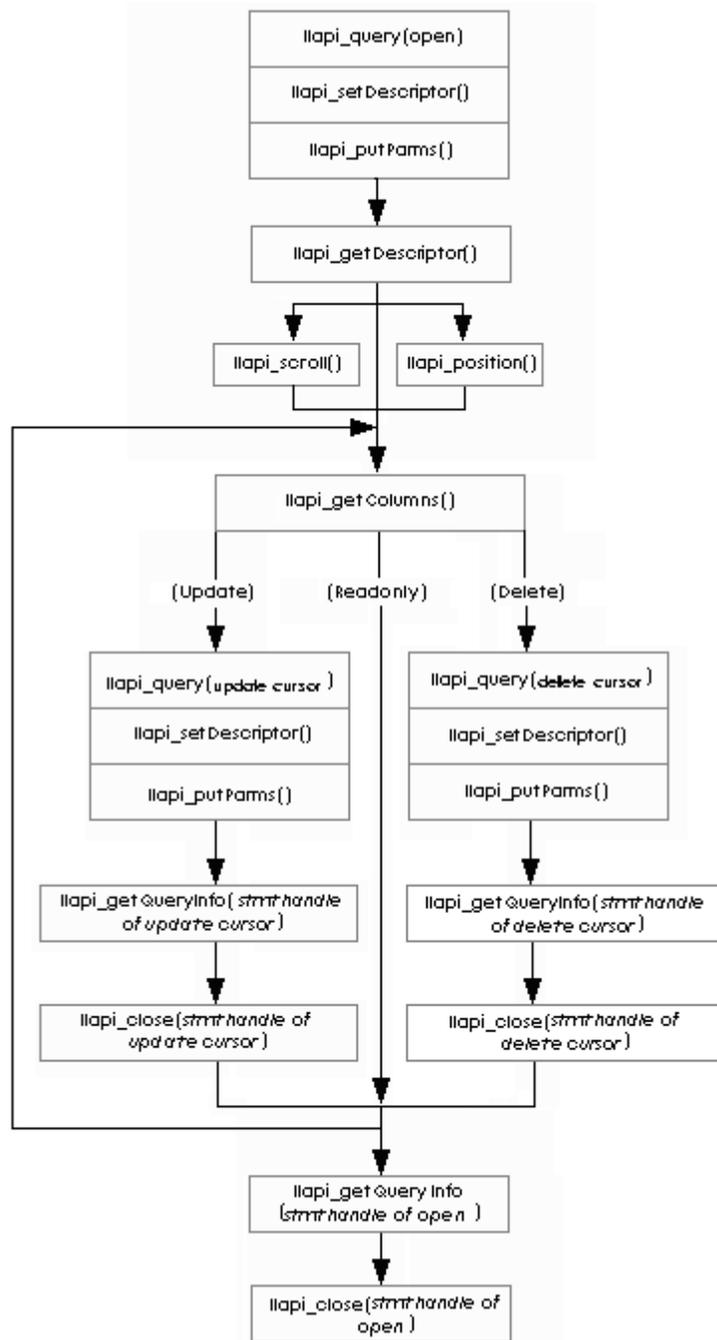
When an application calls `IIapi_query()` to open a cursor, it provides the name of the cursor as a parameter in subsequent calls to `IIapi_setDescriptor()` and `IIapi_putParms()`. *Cursor name* is the character string, unique within the application, that represents the cursor.

When the application calls `IIapi_query()` to update or delete data with a cursor, it provides the cursor ID as a parameter in subsequent calls to `IIapi_setDescriptor()` and `IIapi_putParms()`. *Cursor ID* is the statement handle returned by `IIapi_query()` when the cursor is opened.

## Scrollable Cursors

The application requests a scrollable cursor by setting the `IIAPI_QF_SCROLL` flag when opening the cursor using `IIapi_query()` with query type `IIAPI_QT_OPEN`. A scrollable cursor can be positioned prior to calling `IIapi_getColumns()` using either `IIapi_scroll()` or `IIapi_position()`. `IIapi_getColumns()` then returns rows starting with the row specified by `IIapi_scroll()` or `IIapi_position()`. If `IIapi_getColumns()` is not preceded by an explicit scroll/position request, the rows following the current cursor position are returned.

### Order of Function Calls Used to Manipulate Data with a Cursor



To use a cursor, the application:

1. Opens the cursor with `IIapi_query()`.
2. Provides cursor name and parameter descriptions and values with `IIapi_setDescriptor()` and `IIapi_putParms()`.
3. Requests a description of the data being returned from the server with `IIapi_getDescriptor()`.

The application may call `IIapi_getQueryInfo()` to obtain the status of the open cursor request.

4. Optionally positions the cursor with `IIapi_scroll()` and `IIapi_position()`.
5. Requests the data with calls to `IIapi_getColumns()` until the function returns with a status of "no more data."

The application may call `IIapi_getQueryInfo()` after each call to `IIapi_getColumns()` to obtain the status of the fetch request.

If the application is deleting or updating information with a cursor, it specifies the cursor delete or update statement for the row where the cursor is positioned. The statement handle returned from the open statement should be used as a cursor ID value for the delete or update statement.

6. Closes the SQL statement and releases the statement handle with `IIapi_close()`. This automatically closes the cursor.

## Batch Statement Execution

The OpenAPI function `IIapi_batch()` (see page 51) lets you execute a set of statements in a batch.

You can execute batch statements with or without parameters, or with multiple parameter sets. Each of these scenarios is described in the following sections.

## How You Can Maximize Performance

You can improve performance generally by following these guidelines:

- If you have large batches of inserts, we recommend that you use prepared statements. Using prepared insert statements with large batches (of more than 100) can significantly improve performance.
- If you are using prepared statements for batch, we recommend that you make your batches as large as possible. Larger batch sizes can make a significant difference with insert performance—even 2 or 3 times faster. In fact, when using prepared insert statements, the larger the batch, the better the performance. Batch sizes up to 100,000 have been noted to significantly improve performance.
- If you must use small batches (of less than 100), then you should avoid using prepared statements. The DBMS optimization works well only for large batches; for small batches, you can achieve better performance by batching non-prepared inserts.

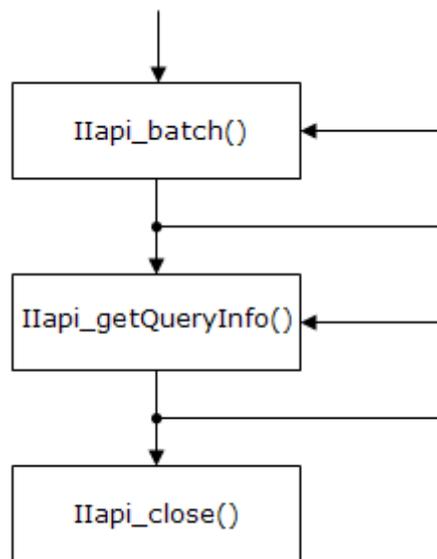
Faster inserts also can be achieved if the following conditions are met:

- Inserts must be into a base table (not a view or index).
- The table must not have any rules or integrities defined on it.
- The table must be a regular Ingres (not gateway) table.
- The inserts must be batched.
- The batched statements must be an execution of a prepared dynamic insert where the dynamic parameters exactly match the values being inserted.
- All but the first statement in the batch should be sent without query text, to achieve an additional performance boost. The prepared statement should be executed in batch as a single statement with multiple parameter sets (see page 32).

## How Batch Statements without Parameters Are Executed

A set of  $n$  statements may be executed as a batch by calling `IIapi_batch()`  $n$  times, once for each statement in the batch. For the first statement in the batch, the statement handle passed to `IIapi_batch()` is set to `NULL` and a new statement handle is returned after the call to `IIapi_batch()`. The returned statement handle is used for all subsequent calls associated with the batch.

After the last statement in the batch is submitted, the statement execution results may be retrieved by calling `IIapi_getQueryInfo()` (see page 80). In general, `IIapi_getQueryInfo()` should be called once for each query in the batch. The results for each query in the batch will be returned in the order the statements were submitted for execution:



IIapi\_getQueryInfo() returns one of the following statuses to indicate the status of the associated statement executed in the batch:

- IIAPI\_ST\_SUCCESS
- IIAPI\_ST\_MESSAGE
- IIAPI\_ST\_WARNING
- IIAPI\_ST\_ERROR

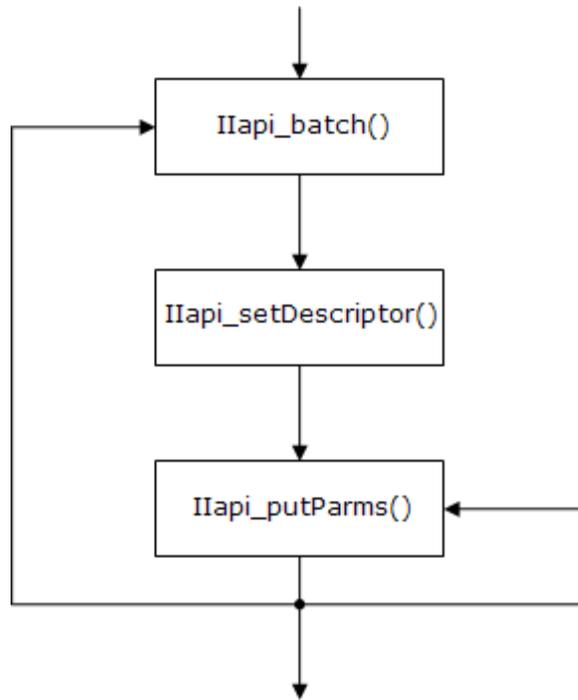
Errors caused by how the application calls IIapi\_getQueryInfo() are indicated by a status of:

- IIAPI\_ST\_FAILURE
- IIAPI\_ST\_NOT\_INITIALIZED
- IIAPI\_ST\_INVALID\_HANDLE
- IIAPI\_ST\_OUT\_OF\_MEMORY

Certain errors during batch processing may result in fewer results being returned than the number of statements contained in the batch. When this occurs, IIapi\_getQueryInfo() returns a status of IIAPI\_ST\_NO\_DATA after all the available results are returned. Even if a critical error does not occur, IIapi\_getQueryInfo() returns a status of IIAPI\_ST\_NO\_DATA after the results for all statements in the batch are returned.

## How Batch Statements with Parameters Are Executed

For statements that contain parameter markers (~V), dynamic parameter markers (?), or database procedure parameters, the parameter meta-data and values can be provided by calling `IIapi_setDescriptor()` and `IIapi_putParms()` after the call to `IIapi_batch()` for a particular statement:

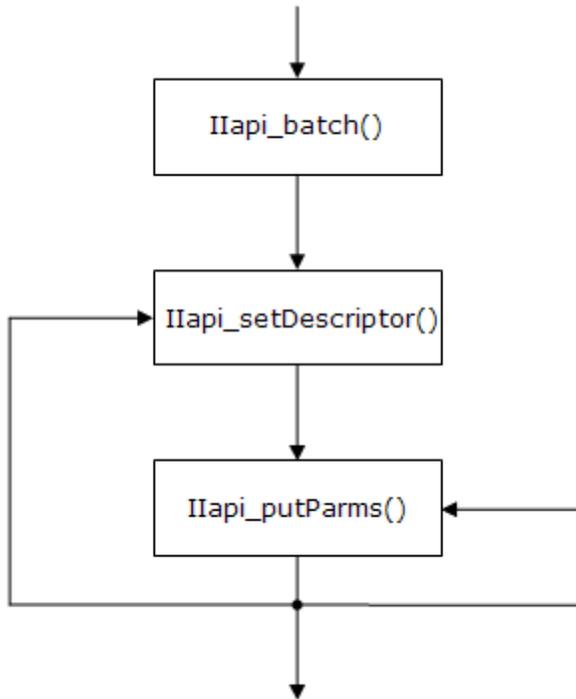


After all parameter values for a statement have been provided by calling `IIapi_putParms()`, the next batch statement may be submitted by calling `IIapi_batch()` or batch results retrieved by calling `IIapi_getQueryInfo()`.

### How Batch Statements with Multiple Parameter Sets Are Repeated

Certain statements such as insert statements, prepared statements, and database procedures, are likely to be executed multiple times in a batch with a different parameter set for each execution. Rather than calling `IIapi_batch()` with the same statement information for each parameter set, you can submit multiple parameter sets for a statement by calling `IIapi_setDescriptor()` and `IIapi_putParms()` for each parameter set after the call to `IIapi_batch()`:

Each parameter set begins with a call to `IIapi_setDescriptor()` and ends when all parameter values have been provided with `IIapi_putParms()`. The statement is executed for each parameter set provided, and execution results are returned for each execution/parameter set. Providing a statement and *n* parameter sets is the same as providing the same statement *n* times, each with a single parameter set, except that the former may be optimized to reduce or eliminate redundant operations.



After the last parameter set is provided, you can submit the next batch statement by calling `IIapi_batch()` or retrieve the batch results by calling `IIapi_getQueryInfo()`.

**Note:** The database procedure name and, optionally, the procedure owner, are provided as service parameters in the parameter set. Even though these will be the same for each repeated execution of a database procedure, these service parameters must still be included with each parameter set.

## Database Events

A *database event* is a notification to an application that a specific condition has occurred.

An application processes database events with the following SQL statements:

### **register dbevent**

Registers to receive event notifications. After register dbevent is executed, the application retrieves events with `IIapi_catchEvent()`.

### **remove dbevent**

Removes an event for which an application has previously registered.

## Order in which OpenAPI Creates, Retrieves, and Deletes Database Events

## How Database Events Are Processed

The application calls `IIapi_getQueryInfo()` and `IIapi_close()` after each call to `IIapi_query()`, shown in the illustration. Additional SQL statements are used to create, drop, and raise database events.

**Note:** For a description and usage of these statements, see the *SQL Reference Guide*.

`IIapi_catchEvent()` registers a callback function to be called when a database event notification is received. `IIapi_catchEvent()` operates similarly to the embedded SQL statement `SET_SQL(DBEVENTHANDLER = dbevent_handler)`. `IIapi_catchEvent()` cannot be used in synchronous mode; it is inherently asynchronous.

For each call to `IIapi_catchEvent()`, only one database event notification will be returned. To receive multiple database events, `IIapi_catchEvent()` must be called every time OpenAPI returns a database event notification to the application. Rather than closing and repeatedly re-allocating event handles, an event handle passed to the application callback function can be re-activated by providing it as input to `IIapi_catchEvent()`.

The server can send database event notifications with other query results. When OpenAPI receives a database event notification, OpenAPI processes the event and calls the the application callback function for any event handles matching the database event.

Database event notifications can also be sent by the server between client queries. OpenAPI provides a function, `IIapi_getEvent()`, which can be used to check for database events between queries or if the application desires only to wait for database event notification without performing other database operations.

`IIapi_getEvent()` waits for a database event notification to be sent by the server on a particular connection. No other request may be made on the connection until `IIapi_getEvent()` completes. A timeout value can be specified when calling `IIapi_getEvent()` so that the application can poll for database event notifications. `IIapi_getEvent()` operates similarly to the embedded SQL statement `GET DBEVENT`.

`IIapi_getEvent()` does not return database event notifications directly. To receive database events, the application must still issue an `IIapi_catchEvent()` request prior to calling `IIapi_getEvent()`. When received by `IIapi_getEvent()`, OpenAPI processes a database event notification and calls the application callback function for any event handles matching the database event.

Matching criteria specified when calling `IIapi_catchEvent()` permits an application to filter database events by name and owner. Database event notifications are compared to all active event handles and may result in callbacks for none, some, or all handles depending on the filtering information.

Since it is possible for a database event notification to be received but not matched to any active event handle, an additional callback function can be registered on the environment handle using `IIapi_setEnvParm()`. This callback function is called for each database event notification that fails to match an active event handle. An event handle is not passed to the callback function, but the output information available from `IIapi_catchEvent()` is passed to the callback function as a structure parameter.

## SQL Syntax

In general, OpenAPI supports SQL syntax identical to that supported by embedded SQL and the Ingres terminal monitors. OpenAPI does not support some embedded SQL statements and supports some SQL statements through OpenAPI functions rather than through an SQL statement. OpenAPI does not support any of the Ingres forms or 4GL statements.

### **More information:**

Mapping of SQL to OpenAPI (see page 151)  
SQL Syntax (see page 158)

## Name Server Query Statement Syntax

OpenAPI provides several types of statements that allow you to access and operate on the Name Server. The create, destroy, and show statements allow you to create, destroy, or show Name Server entities such as login, connection, or attribute definitions.

### **More information:**

Mapping of Name Server Query Statements to OpenAPI (see page 165)

## Query Parameters

OpenAPI does not support host variables, as does embedded SQL. There are several mechanisms by which an application can handle queries for which the parameter values are not known until runtime.

First, the application can use C library string formatting routines, such as `sprintf()`, to build the SQL query text at runtime. Parameter values are formatted as literals in the query text prior to query execution.

Second, the application can use dynamic SQL and provide the parameter values as parameters to the query when the statement is executed or a cursor is opened. (Dynamic SQL is not supported by the Name Server.)

A third method, which uses parameter markers but does not require the statement to be prepared, is also available. This method is the same mechanism used by embedded SQL to handle host variables. It is also the only method that handles runtime parameter values for repeat queries. Since this mechanism is usually hidden to applications, it is not described further here.

**Note:** Parameter strings in queries should be sent as `varchar` type, rather than `chars` type, due to the pattern matching method used by the DBMS Server.

### **More information:**

Query Parameters (see page 187)

## How Unformatted Data is Handled

Long varchar and long byte data types are binary large objects (sometimes called BLOBs) that can store up to 2 GB of data. Since it is often impossible to allocate a storage buffer of this size, special handling is required to segment the unformatted data across the OpenAPI interface. This is done with the `IIapi_getColumns()`, `IIapi_putColumns()`, and `IIapi_putParms()` functions.

Each of these functions contain three common parameters:

### **parmCount or columnCount**

Contain the number of parameters being sent or columns being retrieved in an SQL statement.

### **parmData or columnData**

Contain the buffers of data being sent or retrieved.

### **moreSegments**

Indicates if there are more data segments to be sent or retrieved for a column of long varchar or long byte data type.

Data passes between the application and OpenAPI in a row. Normally, all data in a row or all parameters in an SQL statement are passed with one call to `IIapi_getColumns()`, `IIapi_putColumns()`, or `IIapi_putParms()`. If one of the columns is a long varchar or long byte, however, each segment of the long varchar or long byte must be passed with a single call to the above functions. The `moreSegments` parameter (which is set to `TRUE` or `FALSE`) indicates whether the long varchar or long byte data is completely retrieved or sent. After it is retrieved or sent, the rest of the data, up to the next long varchar or long byte, is passed with one function call.

### **Example—passing unformatted data**

An application calls `IIapi_getColumns()` to retrieve a row of ten columns in a table. The fifth column is a long varchar data type spanning multiple segments. To retrieve the data, the application does the following:

1. Requests the first four columns with a call to `IIapi_getColumns()`.
2. Requests one segment of the long varchar column with a call to `IIapi_getColumns()`.
3. Continues calling `IIapi_getColumns()` until all segments of the long varchar are retrieved (`gc_moreSegments` is `FALSE`).
4. Requests the remaining five columns with a call to `IIapi_getColumns()`.

The same logic is used when an application is sending data to a server with `IIapi_putParms()` or `IIapi_putColumns()`.

## LOB Locators

Rather than retrieving long data along with other query data, an application can request a reference to the long data, called a locator, by setting the `IIAPI_QF_LOCATOR` flag when calling `IIapi_query()`. Locators are 4-byte integer values that reference the long data where it resides in the database. A locator value is only valid on the connection and during the transaction in which it is received.

A locator may be used to retrieve the referenced data by issuing the following query:

```
select ~V
```

The locator value is provided as a query parameter using `IIapi_setDescriptor()` and `IIapi_putParams()`. (See Query Parameters for details on `~V` parameter markers.) To ensure the long data is returned, the `IIAPI_QF_LOCATOR` flag should not be set for this query. The long data is then received and processed in segments as described above.

In addition to retrieving the referenced long data, locators can also be used to operate on the long data as it resides in the database. A `~V` query marker and locator query parameter can be used in SQL statements in places where long data references are permitted. In particular, the following statements provide access to long data while it resides in the database:

```
select length( ~V )
select substring( ~V from start for length )
select position( pattern, ~V , start )
```

## Data Conversion

Most Ingres data types have a corresponding C language data type. Data values passed between the application and a server use the C language data formats. For data types that do not have corresponding C language data types—namely money, decimal, and date—OpenAPI provides two functions, `IIapi_convertData()` and `IIapi_formatData()`, which convert these values to a native C language format. The introduction of new data types is controlled by an OpenAPI level negotiated by the `IIapi_connect()` function.

### More information:

Ingres Data Types (see page 201)  
Data Type Descriptions (see page 204)  
`IIapi_initialize()` Function (see page 85)  
`IIapi_connect()` Function (see page 62)

## Error Handling

This section describes how the application checks for errors.

### How Status Checking Works

An application checks for OpenAPI function errors when the function has completed its tasks, as indicated by the completion flag, `gp_completed`, in the generic parameter block.

When the function successfully or unsuccessfully completes its tasks, the completion flag is set to `TRUE` and the callback function is invoked if the callback address is provided in the parameter block. The value of the generic parameter `gp_status` indicates the success or failure of the function.

A failed task may have additional error information attached to it. To find out if such information exists, an application examines the value of the generic parameter `gp_errorHandle`. If the handle is non-NULL, the application calls `IIapi_getErrorInfo()` to retrieve the additional error information.

#### **More information:**

`IIapi_getErrorInfo()` Function (see page 77)

### How OpenAPI Error Codes Are Generated

When an OpenAPI function detects an error, an error code and text describing the error are generated. This information is available through `IIapi_getErrorInfo()` and the error handle returned in the generic parameters of the function parameter block. When `IIapi_getErrorInfo()` returns with `ge_serverInfoAvail` set to `FALSE`, the error information generated by the OpenAPI and `ge_errorCode` is set to a particular value.

#### **More information:**

Error Codes (see page 191)

SQLSTATE Values and Descriptions (see page 194)



# Chapter 2: OpenAPI Function Reference

---

This section contains the following topics:

[Generic Parameters](#) (see page 41)

[Output Parameters Allocated by OpenAPI](#) (see page 43)

[Function Summary](#) (see page 43)

[OpenAPI Functions](#) (see page 48)

**Note:** For a description of non-standard C data types used in OpenAPI parameter blocks, see OpenAPI Data Types (see page 133).

## Generic Parameters

Each OpenAPI function parameter block has a common substructure. This substructure contains generic parameters for handling asynchronous processing and for communicating the return status of the OpenAPI function.

The substructure is the first element in most parameter blocks:

```
typedef struct _IIAPI_GENPARAM
{
    II_VOID      (II_FAR II_CALLBACK *gp_callback)
                (II_PTR closure, II_PTR parmBlock);
    II_PTR      gp_closure;
    II_BOOL     gp_completed;
    IIAPI_STATUS gp_status;
    PTR        gp_errorHandle;
} IIAPI_GENPARAM;
```

The generic parameters common to each OpenAPI function are as follows:

### **gp\_callback**

**Type:** input

The address of the application function to be invoked when the OpenAPI function completes its tasks, or NULL. If this parameter is NULL, the application can poll for completion by examining `gp_completed`; otherwise, it must be an address to the callback with the following syntax:

```
II_EXTERN II_VOID II_FAR II_CALLBACK callback (II_PTR closure, II_PTR
parmBlock);
```

**gp\_closure**

**Type:** input

The value of the input argument to the function addressed by gp\_callback. OpenAPI does not interpret the value of this parameter, but simply passes it to the callback function unchanged. The application uses this parameter to pass any information it wishes to the callback.

**gp\_completed**

**Type:** immediate and delayed output

The indication that the task has been completed and the callback function has been invoked. If so, this parameter is TRUE; otherwise, it is FALSE.

**gp\_status**

**Type:** delayed output

The status of the OpenAPI function upon its completion.

The following value is returned if the function completes without an error:

IIAPI\_ST\_SUCCESS

The following values indicate that a message (including which type of message) was returned by the server; the message information is available through gp\_errorHandle:

IIAPI\_ST\_MESSAGE  
IIAPI\_ST\_WARNING  
IIAPI\_ST\_ERROR

The following value is returned when a function that normally returns information has nothing to return:

IIAPI\_ST\_NO\_DATA

The following value is returned for general failures, which will have additional information available through gp\_errorHandle:

IIAPI\_ST\_FAILURE

The following values are returned when the OpenAPI is unable to provide additional information because of the error:

IIAPI\_ST\_NOT\_INITIALIZED  
IIAPI\_ST\_INVALID\_HANDLE  
IIAPI\_ST\_OUT\_OF\_MEMORY

**gp\_errorHandle**

**Type:** delayed output

Additional information associated with the completion of the OpenAPI function. If non-NULL, this parameter will be a handle that can be passed to IIApi\_getErrorInfo(). If NULL, no additional information is available.

## Output Parameters Allocated by OpenAPI

An application allocates memory for each OpenAPI function argument and its input parameters.

The application also allocates memory for the output parameters, except for the following parameters, which are allocated by OpenAPI:

Parameter	Function
ce_eventDB	IIapi_catchEvent()
ce_eventName	IIapi_catchEvent()
ce_eventOwner	IIapi_catchEvent()
ce_eventTime.dv_value	IIapi_catchEvent()
cp_fileName	IIapi_getCopyMap()
cp_logName	IIapi_getCopyMap()
cp_dbmsDescr	IIapi_getCopyMap()
cp_fileDescr	IIapi_getCopyMap()
gd_descriptor	IIapi_getDescriptor()
ge_message	IIapi_getErrorInfo()
svr_parmDescr	IIapi_getErrorInfo()
svr_parmValue	IIapi_getErrorInfo()

The output parameters allocated by OpenAPI are maintained until the application calls IIapi\_close() with the statement handle associated with these parameters. To retain the information after IIapi\_close() is invoked, the application copies the parameters into its own buffers.

## Function Summary

The following is a summary of the OpenAPI functions, grouped by tasks.

## OpenAPI Management

### **IIapi\_initialize()**

Initializes OpenAPI.

### **IIapi\_setEnvParam()**

Assigns an environment parameter and value in the environment handle.

### **IIapi\_releaseEnv()**

Releases resources associated with an environment handle.

### **IIapi\_terminate()**

Terminates OpenAPI.

## Session Management

### **IIapi\_abort()**

Closes a server connection forcibly and frees the connection handle.

### **IIapi\_connect()**

Connects to a server and allocates a connection handle.

### **IIapi\_disconnect()**

Closes a server connection, after doing an orderly shutdown, and frees the connection handle.

### **IIapi\_modifyConnect()**

Sends connection parameters to the server.

### **IIapi\_setConnectParam()**

Assigns a connection parameter and value to a connection.

---

## Query Processing

### **IIapi\_query()**

Begins a query statement and allocates a statement handle, and can also allocate a transaction ID handle.

### **IIapi\_batch()**

Adds statements to a group of statements to be executed as a batch. Allocates a statement handle and can also allocate a transaction handle. For more information about IIapi\_batch(), see Batch Statement Execution (see page 27).

Support for batch processing must be enabled in OpenAPI by initializing at API version 7 or greater. To initialize with this version, the application uses IIapi\_initialize() (see page 85) and sets the in\_version parameter to IIAPI\_VERSION\_7.

The DBMS must also support batch processing, which you can determine with the OpenAPI connection level. The connection level is returned by IIapi\_connect() (see page 62) in the co\_apiLevel parameter. A minimum level of IIAPI\_LEVEL\_6 is required.

### **IIapi\_setDescriptor()**

Sends information about the format of data being sent to the server for subsequent IIapi\_putParms() or IIapi\_putColumns() calls.

### **IIapi\_putParms()**

Sends data values for query statement parameters.

### **IIapi\_getCopyMap**

Returns the data format of the copy file and database table involved in a copy statement.

### **IIapi\_putColumns()**

Sends data to be copied from a file to a database table.

### **IIapi\_getDescriptor()**

Returns the format of the data for subsequent IIapi\_getColumns() call.

### **IIapi\_getColumns()**

Returns the results of a query.

### **IIapi\_getColumnInfo()**

Returns additional information about columns retrieved by IIapi\_getColumns().

### **IIapi\_getQueryInfo()**

Returns information about a query.

### **Iiapi\_getErrorInfo()**

Returns additional error or user-defined information.

**IIapi\_position()**

Positions a scrollable cursor.

**IIapi\_scroll()**

Scrolls a scrollable cursor.

**IIapi\_cancel()**

Cancels an outstanding query.

**IIapi\_close()**

Ends a query statement and frees the statement handle.

## Transaction Operations

**IIapi\_registerXID()**

Reserves a unique ID for a two-phase commit transaction.

**IIapi\_savePoint()**

Marks a savepoint for a partial rollback.

**IIapi\_prepareCommit()**

Begins a two-phase commit for a transaction.

**IIapi\_autocommit()**

Enables or disables an autocommit transaction.

**IIapi\_commit()**

Commits a transaction and frees the transaction ID handle.

**IIapi\_rollback()**

Rolls back a transaction and frees the transaction ID handle.

**IIapi\_releaseXID()**

Releases a unique ID for a two-phase commit transaction.

**IIapi\_xaStart()**

Starts an XA transaction.

**IIapi\_xaEnd()**

Ends an XA transaction.

**IIapi\_xaPrepare()**

Prepares an XA transaction for two-phase commit.

**IIapi\_xaCommit()**

Commits an XA transaction.

**IIapi\_xaRollback()**

Rolls back an XA transaction.

## Miscellaneous

### **IIapi\_catchEvent()**

Retrieves a database event and allocates a statement handle.

### **IIapi\_getEvent()**

Provides interface for applications to wait for database events to occur.

### **IIapi\_convertData()**

Converts Ingres data values to a compatible type, using Ingres installation and user configuration settings.

### **IIapi\_formatData()**

Converts Ingres data values to a compatible type, using OpenAPI environment handle settings.

### **IIapi\_wait()**

Blocks control from the application until an outstanding operation completes, or until a user-defined time-out period expires.

## OpenAPI Functions

This section describes each OpenAPI function and provides its syntax.

## IIapi\_abort() Function—Abort a Connection

The IIapi\_abort() function closes a connection opened with IIapi\_connect() and frees all the resources associated with the connection handle.

Any transaction associated with the connection is aborted, and the transaction handle freed, as are all associated statements and database event handles.

While this function can be used to clean up quickly when a problem is detected on a connection, applications are encouraged to continue calling IIapi\_close(), IIapi\_commit() or IIapi\_rollback(), and IIapi\_disconnect() to cleanly shut down server connections.

This function has the following syntax:

```
II_VOID IIapi_abort ( IIAPI_ABORTPARAM *abortParm );
```

```
typedef struct _IIAPI_ABORTPARAM
{
    IIAPI_GENPARAM  ab_genParm;
    II_PTR          ab_connHandle;
} IIAPI_ABORTPARAM;
```

This function has the following parameters:

### **ab\_genParm**

**Type:** input and delayed output

Specifies the generic parameters.

For a description, see Generic Parameters (see page 41).

### **ab\_connHandle**

**Type:** input

Specifies the connection handle to be disconnected immediately. All resources associated with this connection are also freed.

## IIapi\_autocommit() Function—Enable or Disable Autocommit Transactions

The IIapi\_autocommit() function provides an interface for the front-end application to manage the autocommit state in a server. It controls the autocommit state of the server through an autocommit transaction handle. An autocommit transaction handle is required to access the Name Server through OpenAPI.

This function is called with a NULL transaction handle to enable the autocommit state in a server. For a DBMS connection, this is equivalent to the SQL statement 'SET AUTOCOMMIT ON'. An autocommit transaction handle is returned to be used in place of a regular transaction handle in IIapi\_query(). Query statements executed in the context of an autocommit transaction are automatically committed when they complete execution.

When called with an existing autocommit transaction handle, this function disables the autocommit state in the server. This is equivalent to the SQL statement 'SET AUTOCOMMIT OFF'. The autocommit transaction handle is freed and can not be referenced further by the application.

When you set autocommit on, a commit occurs automatically after every statement except prepare and describe. If autocommit is on and you open a cursor, the DBMS does not issue a commit until the close cursor statement is executed because cursors are logically a single statement.

This function has the following syntax:

```
II_VOID IIapi_autocommit( IIAPI_AUTOPARM *autoParm);
```

```
typedef struct _IIAPI_AUTOPARM
{
    IIAPI_GENPARAM ac_genParm;
    II_PTR         ac_connHandle;
    II_PTR         ac_tranHandle;
} IIAPI_AUTOPARM;
```

This function has the following parameters:

### **ac\_genParm**

**Type:** input and delayed output

Specifies the generic parameters.

For a description, see Generic Parameters (see page 41).

### **ac\_connHandle**

**Type:** input

Specifies the connection handle identifying the connection associated with the autocommit transaction. Set to NULL when disabling an existing autocommit transaction.

**ac\_tranHandle**

**Type:** input and immediate output

Specifies the transaction handle to be used while autocommit is enabled. Set to NULL when enabling an autocommit transaction.

**IIapi\_batch() Function—Execute Batch Statements**

The IIapi\_batch() function enables an application to request batch execution of multiple query statements. Using IIapi\_batch() is similar to using IIapi\_query() (see page 95) to execute a query statement. IIapi\_batch() differs from IIapi\_query() in the following ways.

The handling of transaction and statement handles for the first call to IIapi\_batch() for a group of query statements is the same as for calling IIapi\_query(). IIapi\_batch() allocates a statement handle. The statement handle must eventually be freed with the IIapi\_close() function. Additional query statements are added to a batch by calling IIapi\_batch() and providing the statement handle returned by the first IIapi\_batch() call.

IIapi\_batch() permits only a limited set of query types: general query statements (IIAPI\_QT\_QUERY), prepared statements (IIAPI\_QT\_EXEC), and database procedures (IIAPI\_QT\_EXEC\_PROCEDURE). In addition to the query type limitation, query statements in a batch are not permitted to return data rows, either through a SELECT query or row-returning database procedure.

If a query statement added to a batch has parameters, the parameters are provided by calls to IIapi\_setDescriptor() and IIapi\_putParms(), just as is done for a query statement being executed using IIapi\_query(), prior to adding the next query statement to the batch. A single query statement can be executed with multiple parameter sets in a batch by calling IIapi\_batch(), IIapi\_setDescriptor(), and IIapi\_putParms() for each set of parameters.

A single query statement also may be executed with multiple parameter sets using a single call to IIapi\_batch(). Following the first call to IIapi\_batch(), IIapi\_setDescriptor(), and IIapi\_putParms(), to provide the first set of parameters, additional calls to IIapi\_setDescriptor() and IIapi\_putParms() may be made to provide the additional parameter sets without calling IIapi\_batch() to re-submit the statement. The DBMS may be able to execute the statement more efficiently when submitted this way. For more information, see Batch Statement Execution (see page 27).

Batch processing is not supported on Name Server connections.

This function has the following syntax:

```
II_VOID IIapi_batch( IIAPI_BATCHPARAM *batchParm );
typedef struct _IIAPI_BATCHPARAM
{
    IIAPI_GENPARAM    ba_genParm;
    II_PTR            ba_connHandle;
    IIAPI_QUERYTYPE   ba_queryType;
    II_CHAR           *ba_queryText;
    II_BOOL           ba_parameters;
    II_UINT4          ba_flags;
    II_PTR            ba_tranHandle;
    II_PTR            ba_stmtHandle;
} IIAPI_BATCHPARAM;
```

This function has the following parameters:

**ba\_genParm**

**Type:** input and delayed output

Specifies the generic parameters (see page 41).

**ba\_connHandle**

**Type:** input

Specifies the connection handle, identifying the connection to the target server.

**ba\_queryType**

Specifies the query type. Batch processing limits the query types that may be executed to the following types:

IIAPI\_QT\_QUERY

IIAPI\_QT\_EXEC

IIQPI\_QT\_EXEC\_PROCEDURE

**\*ba\_queryText**

**Type:** input

Specifies the syntax of the query statement, if syntax is required. The query text should be a NULL-terminated string containing the query text with zero or more parameter markers.

**ba\_parameters**

**Type:** input

Indicates whether there are parameters to be sent with the batch. TRUE if the application intends to send parameters with the batch using IIapi\_setDescriptor() and IIapi\_putParms(); FALSE if there are no parameters to be sent with the batch.

For more information, see [How Batch Statements with Parameters Are Executed](#) (see page 31) and [How Batch Statements with Multiple Parameter Sets Are Repeated](#) (see page 32).

**ba\_flags**

**Type:** input

Specifies query options. Currently, there are no query options applicable to batch processing. This parameter should be set to zero.

**ba\_tranHandle**

**Type:** input and immediate output

Indicates whether a new transaction should be started or whether an existing transaction should be used. If the parameter is NULL or a transaction ID handle created by `IIapi_registerXID()`, a new transaction should be started. If the parameter is a transaction handle previously returned by another query, the existing transaction should be used.

**ba\_stmtHandle**

**Type:** immediate output

Specifies the statement handle identifying the query statement for future OpenAPI function calls.

This parameter is also an input parameter for batch processing. It should be set to NULL for the first statement in a batch. A statement handle will be returned that should be provided as input for all subsequent statements in the batch.

## IIapi\_cancel() Function—Cancel an Outstanding Query Statement

The IIapi\_cancel() function causes a query statement started with IIapi\_query() or an event retrieval started with IIapi\_catchEvent() to complete its operations.

When IIapi\_cancel() completes successfully, the query may not yet be canceled; it simply means that the server has received the cancellation request. Each canceled query receives a callback.

This function is different from IIapi\_close(), which is used to end a query. The IIapi\_close() function waits for the completion of the query, whereas IIapi\_cancel() attempts to end it before it is completed.

Statement handle output from IIapi\_query() and event handle output from IIapi\_catchEvent() is available when the function returns, so queries can be canceled at anytime thereafter.

The following are possible results of the IIapi\_cancel() function and their corresponding actions:

- Completes with status IIAPI\_ST\_SUCCESS-Cancel request has been issued but the server has not yet canceled the query. There is an outstanding OpenAPI request that will complete with status IIAPI\_ST\_FAILURE and error code E\_AP0009\_QUERY\_CANCELLED, after which the statement handle can be closed using IIapi\_close().
- Completes with status IIAPI\_ST\_WARNING-IIapi\_cancel() has already been called for the statement handle and this call has been ignored; otherwise, same as IIAPI\_ST\_SUCCESS status above.
- Completes with status IIAPI\_ST\_FAILURE and error code E\_AP0009\_QUERY\_CANCELLED-The query has been canceled and statement handle can be closed using IIapi\_close().
- Completes with status IIAPI\_ST\_FAILURE and error code E\_AP0008\_QUERY\_DONE-Query has completed processing and cannot be canceled. Statement handle can be closed using IIapi\_close().
- Completes with status IIAPI\_ST\_FAILURE and error code E\_AP0006\_INVALID\_SEQUENCE-Statement handle is in the process of being closed and no further actions are permitted.

This function has the following syntax:

```
II_VOID IApi_cancel (IIAPI_CANCELPARAM *cancelParm);

typedef struct _IIAPI_CANCELPARAM
{
    IIAPI_GENPARAM  cn_genParm;
    II_PTR          cn_stmtHandle;
} IIAPI_CANCELPARAM;
```

This function has the following parameters:

**cn\_genParm**

**Type:** input and delayed output

Specifies the generic parameters.

For a description, see Generic Parameters (see page 41).

**cn\_stmtHandle**

**Type:** input

Specifies the statement or event handle identifying the query to be canceled.

## IIapi\_catchEvent() Function—Retrieve a Database Event

The IIapi\_catchEvent() function retrieves a database event previously registered by an application. The application first registers for a database event using IIapi\_query() to invoke the register dbevent statement. It then captures each occurrence of the event and retrieves its data by calling IIapi\_catchEvent().

IIapi\_catchEvent() allows the application to specify the database event by the event parameters. If ce\_selectEventName and ce\_selectEventOwner are specified, one specific event is captured when the function completes. If only ce\_selectEventOwner is specified, any events owned by the specific owner are captured. If only ce\_selectEventName is specified, any events with the event name are captured. If neither ce\_selectEventName nor ce\_selectEventOwner are specified, all events are captured.

When IIapi\_catchEvent() returns, it provides an event handle as immediate output. This handle is used to retrieve additional event information, to cancel the event retrieval, or to clean up after the event is retrieved.

Calling IIapi\_catchEvent() captures one occurrence of a database event. After it has captured the event, the application can immediately issue another IIapi\_catchEvent() with the same event handle to capture the next occurrence.

IIapi\_catchEvent() operates in a manner similar to the embedded SQL statement set\_sql(dbeventhandler=<dbevent\_handler>). IIapi\_catchEvent() does not result in any communication with the server. Since database event notification is passed along with other SQL result information, the application must continue with other database operations after calling IIapi\_catchEvent() to enable retrieval of database events.

IIapi\_getEvent() can be used to receive database event notifications when the application has no other database operations to perform. IIapi\_getEvent() operates in a manner similar to the embedded SQL statement GET DBEVENT. See IIapi\_getEvent() for further details.

If IIapi\_catchEvent() completes with ce\_eventInfoAvail set to TRUE, the application should call IIapi\_getDescriptor() and IIapi\_getColumns() to access additional event information.

If the application no longer wants to retrieve a pending database event, it calls IIapi\_cancel() to cancel the event retrieval. A canceled event retrieval completes with a failure status. After an event is canceled, or an event retrieval has completed, the application calls IIapi\_close() to free the event handle and associated internal resources.

If an event notification occurs for which there is no active matching event handle, the database event is ignored. `IIapi_setEnvParam()` permits a callback function to be registered, which will be called for each database event that fails to match to an active event handle and would otherwise be ignored. See `IIAPI_EP_EVENT_FUNC` and related documentation in the description for `IIapi_setEnvParam()`.

This function has the following syntax:

```
II_VOID IIapi_catchEvent (IIAPI_CATCHEVENTPARM *catchEventParm);

typedef struct _IIAPI_CATCHEVENTPARM
{
    IIAPI_GENPARM ce_genParm;
    II_PTR        ce_connHandle;
    II_CHAR       *ce_selectEventName;
    II_CHAR       *ce_selectEventOwner;
    II_PTR        ce_eventHandle;
    II_CHAR       *ce_eventName
    II_CHAR       *ce_eventOwner
    II_CHAR       *ce_eventDB;
    IIAPI_DATAVALUE ce_eventTime;
    II_BOOL       ce_eventInfoAvail;
} IIAPI_CATCHEVENTPARM;
```

This function has the following parameters:

#### **ce\_genParm**

**Type:** input and delayed output

Specifies the generic parameters.

**Note:** For a description, see Generic Parameters (see page 41).

#### **ce\_connHandle**

**Type:** input

Specifies the connection handle, identifying the connection to the target server, or the environment handle.

#### **ce\_selectEventName**

**Type:** input

Specifies the name of the event to retrieve. This parameter is NULL if the application wants to receive the next event indiscriminately; otherwise, it is a NULL-terminated string containing the name of the event the application is requesting.

#### **ce\_selectEventOwner**

**Type:** input

Specifies the owner of the event to retrieve. This parameter is NULL if the application requests events by any owner; otherwise, it is a NULL-terminated string containing the name of the event owner.

**ce\_eventHandle**

**Type:** input and immediate output

Specifies the event handle identifying the event retrieval. Set to NULL to begin a new event retrieval. After receiving the requested event, the application may call `IIapi_catchEvent()` with the same event handle to receive the next event. The application should clean up resources when event retrieval is no longer desired by calling `IIapi_close()` with this event handle.

**ce\_eventName**

**Type:** delayed output

Assigns the name of the event retrieved when the function completes.

The memory for this parameter is managed by the OpenAPI. For more information, see *Output Parameters Allocated by OpenAPI* (see page 43).

**ce\_eventOwner**

**Type:** delayed output

Assigns the owner name of the event retrieved when the function completes.

The memory for this parameter is managed by OpenAPI. For more information, see *Output Parameters Allocated by OpenAPI* (see page 43).

**ce\_eventDB**

**Type:** delayed output

Indicates the server where the event occurred. This parameter contains a NULL-terminated string containing the name of the server.

The memory for this parameter is managed by the OpenAPI. For more information, see *Output Parameters Allocated by OpenAPI* (see page 43).

**ce\_eventTime**

**Type:** delayed output

Indicates the time the event occurred, stored as an `IIAPI_DTE_TYPE` data value.

The memory for this parameter is managed by the OpenAPI. For more information, see *Output Parameters Allocated by OpenAPI* (see page 43).

**ce\_eventInfoAvail**

**Type:** delayed output

Indicates whether there is additional information associated with the event. The parameter is `FALSE` if there is no additional information associated with the event; otherwise, the application should call `IIapi_getDescription()` and `IIapi_getColumns()` with `ce_eventHandle` to retrieve the additional event information.

## IIapi\_close() Function—End a Query Statement or Database Event Retrieval

The IIapi\_close() function ends a query or database event retrieval started with IIapi\_query(), IIapi\_batch(), or IIapi\_catchEvent(). This function frees the event or statement handle and closes any cursor associated with the statement handle.

An application calls IIapi\_close() to end a query that has finished all OpenAPI function calls required by the query type. The application must call IIapi\_cancel() to interrupt a query with outstanding requests prior to calling IIapi\_close().

IIapi\_close() invalidates any data returned by the IIapi\_catchEvent(), IIapi\_getCopyMap(), IIapi\_getDescriptor(), or IIapi\_getErrorInfo() functions for the statement handle.

**Note:** For more information, see Output Parameters Allocated by OpenAPI (see page 43).

For each IIapi\_query(), IIapi\_batch(), or IIapi\_catchEvent(with NULL ce\_eventHandle), there should be a corresponding IIapi\_close().

The following are possible results of the IIapi\_close() function and their corresponding actions:

- Completes with status IIAPI\_ST\_FAILURE and error code E\_AP0006\_INVALID\_SEQUENCE-IIapi\_close() has already been called for the statement handle and this call is ignored.
- Completes with status IIAPI\_ST\_FAILURE and error code E\_AP0007\_INCOMPLETE\_QUERY-OpenAPI query processing of the statement handle has not completed. Query must be canceled using IIapi\_cancel().
- Competes with status IIAPI\_ST\_FAILURE and error code E\_AP000A\_QUERY\_INTERRUPTED-IIapi\_cancel() has been called on the statement handle but the server has not yet canceled the query. There is an outstanding OpenAPI request that will complete with status IIAPI\_ST\_FAILURE and error code E\_AP0009\_QUERY\_CANCELLED, after which the statement handle can be closed using IIapi\_close().

This function has the following syntax:

```
II_VOID Iiapi_close (IIAPI_CLOSEPARAM *closeParm);

typedef struct _IIAPI_CLOSEPARAM
{
    IIAPI_GENPARAM cl_genParm;
    II_PTR         cl_stmtHandle;
} IIAPI_CLOSEPARAM;
```

This function has the following parameters:

**cl\_genParm**

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

**cl\_stmtHandle**

**Type:** input

Specifies the statement or event handle identifying the query or event retrieval for which resources should be freed.

## IIapi\_commit() Function—Commit a Transaction

The IIapi\_commit() function commits a transaction that was started with IIapi\_query() or IIapi\_batch(), or restarted by IIapi\_connect(), and then frees the transaction handle.

Before the transaction is committed, the application must call IIapi\_close() to free the statement handles within the transaction.

This function has the following syntax:

```
II_VOID IIapi_commit (IIAPI_COMMITPARAM *commitParm);

typedef struct _IIAPI_COMMITPARAM
{
    IIAPI_GENPARAM cm_genParm;
    II_PTR cm_tranHandle;
} IIAPI_COMMITPARAM;
```

This function has the following parameters:

### **cm\_genParm**

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

### **cm\_tranHandle**

**Type:** input

Specifies the transaction ID handle identifying the transaction to be committed. The transaction ID handle is either from qy\_tranHandle of IIapi\_query(), ba\_tranHandle of IIapi\_batch(), or co\_tranHandle of IIapi\_connect().

## IIapi\_connect() Function—Connect to a DBMS Server or Name Server

The IIapi\_connect() function establishes a connection to a DBMS Server or the Name Server, based on the connection type, and allocates a connection handle. When connecting to the Name Server an environment handle must be provided as input.

This function returns the co\_apiLevel output parameter, which specifies the level of functionality provided by OpenAPI for the current connection. If the output connection handle is not NULL after this function returns, even if an error occurs, IIapi\_disconnect() or IIapi\_abort() must be called to release the connection handle.

This function has the following syntax:

```
II_VOID IIapi_connect (IIAPI_CONNPARM *connParm);
```

```
typedef struct _IIAPI_CONNPARM
{
    IIAPI_GENPARM    co_genParm;
    II_CHAR          *co_target;
    II_CHAR          *co_username;
    II_CHAR          *co_password;
    II_LONG          co_timeout;
    II_PTR           co_connHandle;
    II_PTR           co_tranHandle;
    II_LONG          co_sizeAdvise;
    II_LONG          co_apiLevel;
    II_LONG          co_type;
} IIAPI_CONNPARM;
```

This function has the following parameters:

### **co\_genParm**

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

### **co\_target**

**Type:** input

The name of the database. When connecting to a DBMS Server, this parameter cannot be NULL. It contains a NULL-terminated string naming the database to which the application will connect. When connecting to a Name Server, NULL is used for the local installation and the vnode name is used for a remote installation.

The syntax of the target name is:

```
[node_id:]dbname[/svr_class]
```

**co\_username****Type:** input

The name of the user connecting to the server. This parameter contains a NULL-terminated string, which is the user name authorized to connect to the server.

If this parameter is NULL, a default user name will be used and co\_password should also be NULL.

**co\_password****Type:** input

A NULL-terminated string containing the password for the user specified in co\_username. If co\_username is NULL, this parameter should be NULL as well.

If connecting to the Name Server, this parameter is not required if the current user has the NET\_ADMIN privilege.

**co\_timeout****Type:** input

Specifies the maximum time in milliseconds to wait for a connection to be established. A value of -1 is used if a timeout is not desired.

Support for timeouts is platform-dependent. If timeouts are not supported, all values are treated the same as -1.

**co\_connHandle****Type:** input and immediate output**Input**

Specifies a connection handle returned by IIapi\_setConnectParam(), an environment handle returned by IIapi\_initialize(), or NULL.

**Output**

Specifies the connection handle identifying the connection for subsequent OpenAPI function calls that invoke query statements and transactions within the context of this connection.

**co\_tranHandle****Type:** input and immediate output

Specifies a transaction ID handle that can be used to re-connect to a distributed transaction, obtained from IIapi\_registerXID(), as input. This parameter (input and output) is usually NULL. A transaction handle for the distributed transaction is returned.

### **co\_sizeAdvise**

**Type:** delayed output

Specifies the advised buffer size for long varchar or long byte segment sizes used within this connection.

### **co\_apiLevel**

**Type:** delayed output

Specifies the OpenAPI level of functionality supported by the server for the connection. Valid values for this parameter are:

#### **IIAPI\_LEVEL\_0**

Specifies level 0. All level 1 data types are supported **except** the following:

IIAPI\_DEC\_TYPE  
IIAPI\_BYTE\_TYPE  
IIAPI\_VBYTE\_TYPE  
IIAPI\_LBYTE\_TYPE  
IIAPI\_LVCH\_TYPE

#### **IIAPI\_LEVEL\_1**

Specifies level 1. All level 2 data types are supported **except** the National Character Set data types:

IIAPI\_NCHA\_TYPE  
IIAPI\_NVCH\_TYPE  
IIAPI\_LNVCH\_TYPE

#### **IIAPI\_LEVEL\_2**

Specifies level 2. (Requires initialization at IIAPI\_VERSION\_3.) All level 3 data types are supported **except** eight-byte integers (bigint):

IIAPI\_INT\_TYPE with length 8

#### **IIAPI\_LEVEL\_3**

Specifies level 3. (Requires initialization at IIAPI\_VERSION\_4.) All level 4 data types are supported **except** ANSI date/time types:

IIAPI\_DATE\_TYPE  
IIAPI\_TIME\_TYPE  
IIAPI\_TMWO\_TYPE  
IIAPI\_TMTZ\_TYPE  
IIAPI\_TS\_TYPE  
IIAPI\_TSWO\_TYPE  
IIAPI\_TSTZ\_TYPE  
IIAPI\_INTYM\_TYPE  
IIAPI\_INTDS\_TYPE

**IIAPI\_LEVEL\_4**

Specifies level 4. (Requires initialization at IIAPI\_VERSION\_5.) All level 5 data types are supported **except** Blob/Clob locator types:

IIAPI\_LCLOC\_TYPE

IIAPI\_LBLOC\_TYPE

IIAPI\_LNLOC\_TYPE

**IIAPI\_LEVEL\_5**

Specifies level 5. (Requires initialization at IIAPI\_VERSION\_6.) All data types in this guide are supported.

**IIAPI\_LEVEL\_6**

Specifies level 6. (Requires initialization at IIAPI\_VERSION\_7.) All data types and capabilities in this guide are supported.

**co\_type**

**Type:** input

Specifies the connection type. Its value is one of the following:

**IIAPI\_CT\_NS**

Establishes a connection to the Name Server.

**IIAPI\_CT\_SQL**

Establishes a connection to a relational DBMS Server.

This parameter is ignored (IIAPI\_CT\_SQL assumed) if co\_connHandle is NULL or is a connection handle that was created with a NULL environment handle.

## IIapi\_convertData() Function—Convert Ingres Data Values to Compatible Types Using Default Settings

The IIapi\_convertData() function converts data values between Ingres data types. The data values are translated and formatted using default settings selected by the user and specified at the operating system level prior to running the application. IIapi\_formatData() can be used to convert data using settings selected by the application.

This function is intended to support Ingres data types that do not have a corresponding C language data type.

Conversions to and from IIAPI\_CHA\_TYPE are supported for the data types IIAPI\_DEC\_TYPE, IIAPI\_MNY\_TYPE, IIAPI\_DTE\_TYPE, IIAPI\_DATE\_TYPE, IIAPI\_TIME\_TYPE, IIAPI\_TMWO\_TYPE, IIAPI\_TMTZ\_TYPE, IIAPI\_TS\_TYPE, IIAPI\_TSWO\_TYPE, IIAPI\_TSTZ\_TYPE, IIAPI\_INTYM\_TYPE, IIAPI\_INTDS\_TYPE.

In addition, the following conversions are supported:

<b>Data Type</b>	<b>Converted To</b>
IIAPI_DEC_TYPE	IIAPI_FLT_TYPE
IIAPI_FLT_TYPE	IIAPI_DEC_TYPE
IIAPI_MNY_TYPE	IIAPI_FLT_TYPE
IIAPI_FLT_TYPE	IIAPI_MNY_TYPE

The source data value must not be NULL (that is, dv\_null set to TRUE).

This function has the following syntax:

```
II_VOID IIApi_convertData(IIAPI_CONVERTPARM *convertParm);

typedef struct _IIAPI_CONVERTPARM
{
    IIAPI_DESCRIPTOR    cv_srcDesc;
    IIAPI_DATAVALUE     cv_srcValue;
    IIAPI_DESCRIPTOR    cv_dstDesc;
    IIAPI_DATAVALUE     cv_dstValue;
    IIAPI_STATUS        cv_status;
} IIAPI_CONVERTPARM;
```

This function has the following parameters:

**cv\_srcDesc**

**Type:** input

Specifies a description of the original data type.

**cv\_srcValue**

**Type:** input

Specifies the original data value. This value must not be NULL.

**cv\_dstDesc**

**Type:** input

Specifies the description of the desired result data type.

**cv\_dstValue**

**Type:** output

Specifies the resulting data value. There must be enough memory allocated to hold the resulting data value, as described by cv\_dstDesc (that is, dv\_length must be equal or greater than ds\_length).

**cv\_status**

**Type:** output

Returns IIAPI\_ST\_SUCCESS if the conversion succeeded;  
IIAPI\_ST\_FAILURE if there is an invalid parameter value, insufficient space to hold the resulting data value, or the input data value could not be converted to the requested type.

## IIapi\_disconnect() Function—Close a Server Connection

The IIapi\_disconnect() function closes the connection to the server opened with IIapi\_connect() and frees all resources associated with the connection handle.

An application must close all outstanding statements and event retrievals, and it must commit or roll back all outstanding transactions before calling IIapi\_disconnect().

This function has the following syntax:

```
II_VOID IIapi_disconnect (IIAPI_DISCONNPARM *disconnParm);

typedef struct _IIAPI_DISCONNPARM
{
    IIAPI_GENPARM    dc_genParm;
    II_PTR           dc_connHandle;
} IIAPI_DISCONNPARM;
```

This function has the following parameters:

### **cd\_genParm**

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

### **dc\_connHandle**

**Type:** input

Specifies the connection handle, identifying the connection to the target server. All resources associated with this connection are also freed.

## IIapi\_formatData() Function—Convert Ingres Data Values to Compatible Types

The IIapi\_formatData() function converts data values between Ingres data types. The data values are translated and formatted using settings selected by the application and specified on an environment handle using IIapi\_setEnvParam(). IIapi\_convertData() can be used to convert data using default settings selected by the user running the application.

For further details, see the description of IIapi\_convertData() (see page 66).

This function has the following syntax:

```
II_VOID IIapi_formatData( IIAPI_FORMATPARM *formatParm );

typedef struct _IIAPI_FORMATPARM
{
    II_PTR                fd_envHandle;
    IIAPI_DESCRIPTOR     fd_srcDesc;
    IIAPI_DATAVALUE      fd_srcValue;
    IIAPI_DESCRIPTOR     fd_dstDesc;
    IIAPI_DATAVALUE      fd_dstValue;
    IIAPI_STATUS         fd_status;
} IIAPI_FORMATPARM;
```

This function has the following parameters:

### **fd\_envHandle**

**Type:** input

Specifies the environment handle for which the data values to be converted.

### **fd\_srcDesc**

**Type:** input

Specifies the description of the original data type.

### **fd\_srcValue**

**Type:** input

Specifies the original data value. The value must not be NULL.

### **fd\_dstDesc**

**Type:** input

Specifies the description of the desired result data type.

### **fd\_dstValue**

**Type:** output

Specifies the resulting data value. There must be enough memory allocated to hold the resulting data value as described by `fd_dstDesc` (that is, `dv_length` must be equal or greater than `ds_length`).

**fd\_status**

**Type:** output

Returns `IIAPI_ST_SUCCESS` if the conversion succeeded. Returns `IIAPI_ST_FAILURE` if there is an invalid parameter value, insufficient space to hold the resulting data value, or the input data value could not be converted to the requested type.

## IIapi\_getColumns() Function—Return Columns from a Previously Invoked Query Statement or Database Event Retrieval

The `IIapi_getColumns()` function returns the results of a query statement or database event retrieval created by `IIapi_query()` or `IIapi_catchEvent()` to the application. It retrieves into the application's buffers the requested number of columns of data.

This function is always preceded with the `IIapi_getDescriptor()` or `IIapi_getCopyMap()` function, which describes the number, order and format of columns to be returned to the application from the server.

`IIapi_getColumns()` assumes the application has allocated sufficient storage for each column. The buffer size of each column should be equal to the `ds_length` parameter of its corresponding descriptor from `IIapi_getCopyMap()` or `IIapi_getDescriptor()`.

`IIapi_getColumns()` can return multiple rows. If there are long varchar, nvarchar, or byte data columns, `IIapi_getColumns()` returns only one row at a time.

If one of the columns is a long varchar, nvarchar, or byte that requires more than one segment to be returned, the application requests the segments individually with single calls to `IIapi_getColumns()`. For example, a row of ten columns with a long varchar, nvarchar, or byte spanning multiple segments as the fifth column is handled as follows. The application:

1. Requests four columns with a call to `IIapi_getColumns()`.
2. Requests one segment of the long varchar, nvarchar, or byte column with a call to `IIapi_getColumns()`.
3. Continues making requests until all segments are retrieved.
4. Requests five columns with a call to `IIapi_getColumns()` to retrieve the rest of the columns in the current row.

Generally, the application calls `IIapi_getColumns()` repeatedly until a request completes with the status `IIAPI_ST_NO_DATA`. A request that returns fewer rows than requested is also an indication that the query is complete. For non-cursor based queries, no other request may be issued until the query completes. Once complete, additional query status information may be obtained by calling `IIapi_getQueryInfo()`. The statement handle should be freed with the `IIapi_close()` function.

For cursor based queries (`IIAPI_QT_OPEN`), each `IIapi_getColumns()` call results in a `FETCH` request being issued against the cursor. Since the query is effectively suspended in between `FETCH` requests, the application is permitted to issue other requests in between calls to `IIapi_getColumns()`. The application can also call `IIapi_getQueryInfo()` after each `IIapi_getColumns()` call to obtain information about the individual `FETCH` requests. `IIapi_getQueryInfo()` can also be called between `IIapi_getDescriptor()` and the first call to `IIapi_getColumns()` to determine the status of the cursor `OPEN` request.

`IIapi_getColumns()` can be preceded by a call to `IIapi_position()` or `IIapi_scroll()` when the associated cursor is scrollable. `IIapi_position()` and `IIapi_scroll()` initiate a positioned `FETCH` request. We recommend that the results of these functions be retrieved in a single call to `IIapi_getColumns()` by requesting the same number of rows as requested for `IIapi_position()` or implied by `IIapi_scroll()`. It is possible, however, to retrieve the results using multiple calls to `IIapi_getColumns()`. As with non-cursor based queries, `IIapi_getColumns()` should be called repeatedly until one of the following occurs:

- The total number of rows returned by the combined `IIapi_getColumns()` calls equals the number of rows requested or implied by the positioned `FETCH`.
- `IIapi_getColumns()` returns fewer rows than requested.
- `IIapi_getColumns()` returns `IIAPI_ST_NO_DATA`.

When one of the conditions listed above occurs, `IIapi_getQueryInfo()` can be called to retrieve additional results of the positioned `FETCH` request. If `IIapi_getQueryInfo()` is called prior to one of these conditions, any remaining results will be discarded.

With scrollable cursors, an `IIAPI_ST_NO_DATA` result or returning fewer rows than requested does not indicate query completion as it does for non-scrollable cursors and non-cursor queries. `IIAPI_ST_NO_DATA` or fewer rows than requested will be returned if one of the following occurs:

- Fewer rows (including 0 rows) were requested by the preceding call to `IIapi_position()` or `IIapi_scroll()`.
- The cursor is positioned before the first row of the result set.
- The cursor reaches a row that has been deleted.
- The cursor reaches the end of the result set.

`IIapi_getColumns()` returns columns sequentially in the order that they appear in the row, as described in `IIapi_getCopyMap()` and `IIapi_getDescriptor()`. When `IIAPI_ST_NO_DATA` is returned, the application should free the statement handle with the `IIapi_close()` function.

`IIapi_getColumns()` assumes the application has allocated sufficient storage for each column. The buffer size of each column should be equal to the `ds_length` parameter of its corresponding descriptor from `IIapi_getCopyMap()` or `IIapi_getDescriptor()`.

This function has the following syntax:

```
II_VOID IIapi_getColumns (IIAPI_GETCOLPARM *getColParm);
```

```
typedef struct _IIAPI_GETCOLPARM
{
    IIAPI_GENPARM    gc_genParm;
    II_PTR           gc_stmtHandle;
    II_INT           gc_rowCount;
    II_INT           gc_columnCount;
    IIAPI_DATAVALUE *gc_columnData;
    II_INT           gc_rowsReturned;
    II_BOOL          gc_moreSegments;
} IIAPI_GETCOLPARM;
```

This function has the following parameters:

**gc\_genParm**

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

**gc\_stmtHandle**

**Type:** input

Specifies the statement handle identifying the query or event handle identifying the database event.

**gc\_rowCount****Type:** input

Specifies the number of rows to fetch. This parameter must be 1 if the query contains any long varchar or long byte values.

**gc\_columnCount****Type:** input

Specifies the number of columns to retrieve. This parameter must contain a non-zero positive integer, and it cannot exceed the total number of columns in a row.

**gc\_columnData****Type:** delayed output

Specifies an array of buffers. The number of buffers in this array must equal  $gc\_rowCount * gc\_columnCount$ . The first  $gc\_columnCount$  buffers receive the columns from the first row. The second row follows immediately after the first. Each buffer must have enough allocated memory to store the value of a column in the order specified by the descriptors returned from `IIapi_getDescriptor()` or `IIapi_getCopyMap()`.

**gc\_rowsReturned****Type:** delayed output

Specifies the number of rows actually returned in `gc_columnData`.

**gc\_moreSegments****Type:** delayed output

Indicates whether there is more data to be retrieved for a column of a long varchar or long byte data type. This parameter is set to `TRUE` if the data type is long varchar or long byte and more data must be retrieved for the current column; otherwise, the parameter is set to `FALSE`.

## IIapi\_getColumnInfo() Function—Return Additional Information about Columns

The IIapi\_getColumnInfo() function retrieves additional information about columns. The information for a column is available after it has been retrieved by IIapi\_getColumns().

This function has the following syntax:

```
II_VOID IIapi_getColumnInfo( IIAPI_GETCOLINFOPARM *getColumnInfoParm );
```

```
typedef struct _ IIAPI_GETCOLINFOPARM
{
    II_PTR          gi_stmtHandle;
    II_INT2        gi_columnNumber;
    IIAPI_STATUS   gi_status;
    II_ULONG       gi_mask;
    II_UINT8       gi_lobLength;
} IIAPI_GETCOLINFOPARM;
```

This function has the following parameters:

### **gi\_stmtHandle**

**Type:** input

Specifies the statement handle identifying the query.

### **gi\_columnNumber**

**Type:** input

Specifies the number of the column for which information is to be returned. This parameter must contain a non-zero positive integer that cannot exceed the total number of columns in a row.

### **gi\_status**

**Type:** immediate output

Returns IIAPI\_ST\_SUCCESS if the column information is successfully returned. Returns IIAPI\_ST\_FAILURE if gi\_columnNumber is less than 1 or greater than the number of columns in rows returned by the statement.

### **gi\_mask**

**Type:** immediate output

Specifies the mask indicating the available column information. This parameter is 0 or a mask of the following values:

IIAPI\_GI\_LOB\_LENGTH

### **gi\_lobLength**

**Type:** immediate output

Specifies the length of a LOB column as indicated by the DBMS. The length is available once the first segment of the LOB has been retrieved. Also provides the length of a LOB value associated with a LOB locator. This parameter is valid only if the `IIAPI_GI_LOB_LENGTH` mask is set in `gq_mask`; otherwise, this parameter should be ignored.

## IIapi\_getCopyMap() Function—Return the Data Format of Copy File and Database Table Involved in a Copy Statement

The `IIapi_getCopyMap()` function returns a pointer to a copy map for data being copied from a file to a database table or from a database table to a file. The copy map describes the data in the file and in the database table.

`IIapi_getCopyMap()` output remains valid until the copy statement is closed with `IIapi_close()`.

This function has the following syntax:

```
II_VOID IIapi_getCopyMap (IIAPI_GETCOPYMAPPARM *getCopyMapParm);
```

```
typedef struct _IIAPI_GETCOPYMAPPARM
{
    IIAPI_GENPARAM    gm_genParm;
    II_PTR            gm_stmtHandle;
    IIAPI_COPYMAP     gm_copyMap;
} IIAPI_GETCOPYMAPPARM;
```

This function has the following parameters:

### **gm\_genParm**

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

### **gm\_stmtHandle**

**Type:** input

Specifies the statement handle identifying the copy statement.

### **gm\_copyMap**

**Type:** delayed output

Specifies the copy map returned from the server. The copy map contains descriptions of the data in the file and in the database table.

## IIapi\_getDescriptor() Function—Communicate Format of Return Data with IIapi\_getColumns()

The IIapi\_getDescriptor() function retrieves information from the server about the format of the data being returned to the application. This information includes the number of columns in a row and the type, length, precision, and scale of each column. The actual values are returned with a subsequent call to IIapi\_getColumns().

IIapi\_getDescriptor() output remains valid until the query is ended with the IIapi\_close() function.

This function has the following syntax:

```
II_VOID IIapi_getDescriptor (IIAPI_GETDESCRPARAM *getDescrParm);

typedef struct _IIAPI_GETDESCRPARAM
{
    IIAPI_GENPARAM      gd_genParm;
    II_PTR              gd_stmtHandle;
    II_LONG             gd_descriptorCount;
    IIAPI_DESCRIPTOR   *gd_descriptor;
} IIAPI_GETDESCRPARAM;
```

This function has the following parameters:

### **gd\_genParm**

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

### **gd\_stmtHandle**

**Type:** input

Specifies the statement handle identifying the query or event handle identifying the database event.

### **gd\_descriptorCount**

**Type:** delayed output

Specifies the number of columns being returned. This parameter is 0 if there is no data that meets the criteria of the query.

### **gd\_descriptor**

**Type:** delayed output

Specifies an array of buffers. The number of buffers is equal to the gd\_descriptorCount value. Each buffer contains the description of a column, including information about the data type, length, precision, and scale.

The memory for this parameter is managed by OpenAPI. For more information, see Output Parameters Allocated by OpenAPI (see page 43).

## IIapi\_getErrorInfo() Function—Return Additional Error or User-defined Information

The IIapi\_getErrorInfo() function returns a set of error parameters or user-defined information specified by an error handle. User-defined information consists of messages declared in a database procedure. These messages are returned during the execution of the database procedure.

Because the error handle can contain more than one set of error parameters, the application should repeat IIapi\_getErrorInfo() until the status IIAPI\_ST\_NO\_DATA is returned by the ge\_status parameter.

The output of IIapi\_getErrorInfo() is valid until another operation is invoked with the same handle.

This function has the following syntax:

```
II_VOID IIapi_getErrorInfo (IIAPI_GETEINFOPARM *getEInfoParm);
```

```
typedef struct _IIAPI_GETEINFOPARM
{
    II_PTR          ge_errorHandle;
    II_LONG         ge_type;
    II_CHAR         ge_SQLSTATE[6];
    II_LONG         ge_errorCode;
    II_CHAR         *ge_message;
    II_BOOL         ge_serverInfoAvail;
    IIAPI_SVR_ERRINFO *ge_serverInfo;
    IIAPI_STATUS    ge_status;
} IIAPI_GETEINFOPARM;
```

This function has the following parameters:

### **ge\_errorHandle**

**Type:** input

Indicates the error handle returned in the generic parameters.

### **ge\_type**

**Type:** immediate output

Indicates the type of message. Its value is one of the following:

- API\_GE\_ERROR
- API\_GE\_WARNING
- API\_GE\_MESSAGE
- API\_GE\_XAERR

**ge\_SQLSTATE**

**Type:** immediate output

Indicates the SQLSTATE value of the error.

**ge\_errorCode**

**Type:** immediate output

Indicates the error code generated by the server.

When ge\_type is API\_GE\_XAERR, the error code will be an XA standard error code (see page 198).

**ge\_message**

**Type:** immediate output

Provides the text of the message.

The OpenAPI manages the memory for this parameter. For more information, see Output Parameters Allocated by OpenAPI (see page 43).

**ge\_serverInfoAvail**

**Type:** immediate output

Indicates whether additional information will be made available. TRUE if the server sent the message; FALSE if the message was generated by OpenAPI. If TRUE, additional information is available in ge\_serverInfo.

**ge\_serverInfo**

**Type:** immediate output

Specifies additional information sent by the server. Only available if ge\_serverInfoAvail is TRUE.

**ge\_status**

**Type:** immediate output

Indicates the status of the function upon its return. Its value is one of the following:

- IIAPI\_ST\_SUCCESS
- IIAPI\_ST\_NO\_DATA
- IIAPI\_ST\_NOT\_INITIALIZED
- IIAPI\_ST\_INVALID\_HANDLE

## IIapi\_getEvent() Function—Wait for Database Events

The IIapi\_getEvent() function provides an interface for applications to wait for database events to occur. The IIapi\_getEvent() function checks for database events that are received, independent of other query results being returned by the server. Database event information is retrieved using IIapi\_catchEvent(), which should be called prior to calling IIapi\_getEvent().

An application prepares to process database events by registering with the server (IIapi\_query()) and OpenAPI (IIapi\_catchEvent()) for specific events. The application can then receive event notification while processing other queries on the connection.

IIapi\_getEvent() can be called to receive database events when the application does not have any query processing to be performed on the desired connection. No queries can be issued on the specified connection handle until IIapi\_getEvent() completes.

This function has the following syntax:

```
II_VOID IIapi_getEvent ( IIAPI_GETEVENTPARM *getEventParm );

typedef struct _IIAPI_GETEVENTPARM
{
    IIAPI_GENPARM    gv_genParm;
    II_PTR           gv_connHandle;
    II_LONG          gv_timeout;
} IIAPI_GETEVENTPARM;
```

This function has the following parameters:

### gv\_genParm

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

### gv\_connHandle

**Type:** input

Specifies the connection handle, identifying the connection to the target server, or the environment handle.

### gv\_timeout

**Type:** input

Specifies the maximum time in milliseconds to wait for the database event to be received. A value of -1 means to wait indefinitely. A value of 0 polls for events without waiting.

**Note:** Support for timeouts is platform-dependent. If timeouts are not supported, all values are treated the same as -1.

## IIapi\_getQueryInfo() Function—Return Information about a Query

The IIapi\_getQueryInfo() function returns data associated with a query statement. It is useful for applications that are updating the database to know if the update was successful. The information includes the following:

- Number of rows affected by a query
- Read-only status of a cursor
- Return status of a database procedure execution
- ID created when a procedure is executed
- ID created when a repeat query is defined
- Table key result of an SQL insert or update query
- Object key result of an SQL insert or update query
- Cursor attributes
- Row position and status

As a common practice, applications should call IIapi\_getQueryInfo() after each query is completed to check if there are any additional errors or response data reported by the server.

The output of IIapi\_getQueryInfo() is valid until the query statement is ended with the IIapi\_close() function.

For query statement batches, IIapi\_getQueryInfo() should be called once for each query statement in the batch. The query results for each statement in the batch will be returned in the order the query statements were submitted to the batch. IIapi\_getQueryInfo() will return a status of IIAPI\_ST\_NO\_DATA if it is called more times than the number of query statements in the batch.

If an error occurs during batch execution which terminates the execution, fewer query results will be returned. In this case, IIapi\_getQueryInfo() will return a status of IIAPI\_ST\_NO\_DATA once all the available query results have been returned.

This function has the following syntax:

```
II_VOID IIapi_getQueryInfo (IIAPI_GETQINFOPARM *getQInfoParm);
```

```

typedef struct _IIAPI_GETQINFOPARM
{
    IIAPI_GENPARAM  gq_genParm;
    II_PTR          gq_stmtHandle;
    II_ULONG        gq_flags;
    II_ULONG        gq_mask;
    II_LONG         gq_rowCount;
    II_BOOL         gq_readonly;
    II_LONG         gq_procedureReturn;
    II_PTR          gq_procedureHandle;
    II_PTR          gq_repeatQueryHandle;
    II_CHAR         gq_tableKey [TBL_KEY_SZ];
    II_CHAR         gq_objectKey [OBJ_KEY_SZ];
    II_ULONG        gq_cursorType;
    II_ULONG        gq_rowStatus;
    II_LONG         gq_rowPosition;
} IIAPI_GETQINFOPARM;

```

This function has the following parameters:

### **gq\_genParm**

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

### **gq\_stmtHandle**

**Type:** input

Specifies the statement handle identifying the query.

### **gq\_flags**

**Type:** delayed output

Provides status flags indicating result of the query. This parameter is 0 or a mask of the following values:

```

IIAPI_GQF_FAIL
IIAPI_GQF_ALL_UPDATED
IIAPI_GQF_NULLS_REMOVED
IIAPI_GQF_UNKNOWN_REPEAT_QUERY
IIAPI_GQF_END_OF_DATA
IIAPI_GQF_CONTINUE
IIAPI_GQF_INVALID_STATEMENT
IIAPI_GQF_TRANSACTION_INACTIVE
IIAPI_GQF_OBJECT_KEY
IIAPI_GQF_TABLE_KEY
IIAPI_GQF_NEW_EFFECTIVE_USER
IIAPI_GQF_FLUSH_QUERY_ID
IIAPI_GQF_ILLEGAL_XACT_STMT

```

### **gq\_mask**

**Type:** delayed output

Specifies the mask indicating the available response data. This parameter is 0 or a mask of the following values:

IIAPI\_GQ\_ROW\_COUNT  
IIAPI\_GQ\_CURSOR  
IIAPI\_GQ\_PROCEDURE\_RET  
IIAPI\_GQ\_PROCEDURE\_ID  
IIAPI\_GQ\_REPEAT\_QUERY\_ID  
IIAPI\_GQ\_TABLE\_KEY  
IIAPI\_GQ\_OBJECT\_KEY  
IIAPI\_GQ\_ROW\_STATUS

This parameter is 0 if no response data is available.

### **gq\_rowCount**

**Type:** delayed output

Specifies the number of rows affected by the query. This parameter is valid only if the IIAPI\_GQ\_ROW\_COUNT mask is set in gq\_mask; otherwise, this parameter should be ignored.

### **gq\_readonly**

**Type:** delayed output

Indicates the type of cursor opened. Set to TRUE if the statement handle represents a cursor opened for READONLY; otherwise, FALSE. This information is available after IIapi\_getDescriptor() or IIapi\_getColumns() completes successfully.

This parameter is valid only if the IIAPI\_GQ\_CURSOR mask is set in gq\_mask; otherwise, this parameter should be ignored.

### **gq\_procedureReturn**

**Type:** delayed output

Indicates the return status of an executed database procedure. This parameter is valid only if the IIAPI\_GQ\_PROCEDURE\_RET mask is set in gq\_mask; otherwise, this parameter should be ignored.

### **gq\_procedureHandle**

**Type:** delayed output

Specifies the procedure handle for a database procedure. This handle can be used for subsequent executions of the procedure. It is used as a parameter of a query when the query type is IIAPI\_QT\_EXEC\_PROCEDURE.

This parameter is valid only if the IIAPI\_GQ\_PROCEDURE\_ID mask is set in gq\_mask; otherwise, this parameter should be ignored.

**gq\_repeatQueryHandle**

**Type:** delayed output

Specifies the repeat query handle for a repeat query. This handle is used for a future invocation of a repeat query. It is used as a parameter of a query when the query type is `IIAPI_QT_EXEC_REPEAT_QUERY`.

This parameter is valid only if the `IIAPI_GQ_REPEAT_QUERY_ID` mask is set in `gq_mask`; otherwise, this parameter should be ignored.

**gq\_tableKey**

**Type:** delayed output

Specifies the table key value. This parameter is valid only if the `IIAPI_GQ_TABLE_KEY` mask is set in `gq_mask`; otherwise, this parameter should be ignored.

**gq\_objectKey**

**Type:** delayed output

Specifies the object key value. This parameter is valid only if the `IIAPI_GQ_OBJECT_KEY` mask is set in `gq_mask`; otherwise, this parameter should be ignored.

**gq\_cursorType**

**Type:** delayed output

Specifies the cursor attributes. This parameter is a mask of the following values:

`IIAPI_CURSOR_UPDATE`

`IIAPI_CURSOR_SCROLL`

This information is available after `IIapi_getDescriptor()` completes successfully.

This parameter is valid only if the `IIAPI_GQ_CURSOR` mask is set in `gq_mask`; otherwise, this parameter should be ignored.

**gq\_rowStatus**

**Type:** delayed output

Specifies the cursor row status. This parameter is a mask of the following values:

`IIAPI_ROW_BEFORE`

`IIAPI_ROW_FIRST`

`IIAPI_ROW_LAST`

`IIAPI_ROW_AFTER`

`IIAPI_ROW_INSERTED`

`IIAPI_ROW_UPDATED`

#### IIAPI\_ROW\_DELETED

This information is available after `IIapi_getColumns()` completes successfully.

This parameter is valid only if the `IIAPI_GQ_ROW_STATUS` mask is set in `gq_mask`; otherwise, this parameter should be ignored.

#### **gq\_rowPosition**

**Type:** delayed output

Specifies the row position of the cursor in the result set. This information is available after `IIapi_getColumns()` completes successfully. This parameter is valid only if the `IIAPI_GQ_ROW_STATUS` mask is set in `gq_mask`; otherwise, this parameter should be ignored.

## IIapi\_initialize() Function—Initialize OpenAPI to a Specified Input Version

The IIapi\_initialize() function prepares OpenAPI for operation. This function allocates an environment handle and returns it to the application. IIapi\_initialize() must be called before the application performs any memory allocation and OpenAPI functions. When IIapi\_initialize() completes, the application can begin issuing other OpenAPI functions. This function can be called more than once, but each call requires a corresponding call to IIapi\_terminate().

The application specifies the OpenAPI interface version used by the application when calling this function. For version IIAPI\_VERSION\_1, a single default environment is used as the context for server connections (the returned environment handle is set to NULL). For version IIAPI\_VERSION\_2 (and higher), a new environment handle is allocated and returned to the application. The application can select environment settings by providing the environment handle as input to IIapi\_setEnvParam(). The application can also make connections utilizing the environment settings by providing the environment handle as input to IIapi\_setConnectParam() or IIapi\_connect(). The environment handle must be released by calling IIapi\_releaseEnv().

This function has the following syntax:

```
II_VOID IIapi_initialize (IIAPI_INITPARM *initParm);
```

```
typedef struct _IIAPI_INITPARM
{
    II_LONG      in_timeout;
    II_LONG      in_version;
    IIAPI_STATUS in_status;
    II_PTR       in_envHandle;
} IIAPI_INITPARM;
```

This function has the following parameters:

### in\_timeout

**Type:** input

Specifies the maximum time in milliseconds to wait for OpenAPI initialization to occur. A value of -1 is used if a timeout is not desired.

Support for timeouts is platform-dependent. If timeouts are not supported, all values are treated the same as -1.

### in\_version

**Type:** input

The OpenAPI interface version being used by the application. Any version of OpenAPI can operate correctly with applications coded and compiled for previous versions of OpenAPI. Its value is one of the following:

**IIAPI\_VERSION\_1**

Specifies the initial version of OpenAPI.

**IIAPI\_VERSION\_2**

Specifies a version that supports an environment handle, which identifies storage for information about a specific environment setup.

**IIAPI\_VERSION\_3**

Indicates support for the National Character Set data types:

- IIAPI\_NCHA\_TYPE
- IIAPI\_NVCH\_TYPE
- IIAPI\_LNVCH\_TYPE

**IIAPI\_VERSION\_4**

Indicates support for eight-byte integers (bigint): IIAPI\_INT\_TYPE.

**IIAPI\_VERSION\_5**

Indicates support for ANSI date/time types:

- IIAPI\_DATE\_TYPE
- IIAPI\_TIME\_TYPE
- IIAPI\_TMWO\_TYPE
- IIAPI\_TMTZ\_TYPE
- IIAPI\_TS\_TYPE
- IIAPI\_TSWO\_TYPE
- IIAPI\_TSTZ\_TYPE
- IIAPI\_INTYM\_TYPE
- IIAPI\_INTDS\_TYPE

**IIAPI\_VERSION\_6**

Indicates support for Blob/Clob locator types:

- IIAPI\_LCLOC\_TYPE
- IIAPI\_LBLOC\_TYPE
- IIAPI\_LNLOC\_TYPE

**IIAPI\_VERSION\_7**

Indicates support for batch processing.

**in\_status**

**Type:** output

The status of the function upon its return. Its value is one of the following:

- IIAPI\_ST\_SUCCESS
- IIAPI\_ST\_OUT\_OF\_MEMORY
- IIAPI\_ST\_FAILURE

**in\_envHandle****Type:** output

Specifies the environment handle returned to the application for the OpenAPI if the version being used by the application is IIAPI\_VERSION\_2.

The value is NULL if the OpenAPI version being used by the application is IIAPI\_VERSION\_1.

**IIapi\_modifyConnect Function—Send Connection Parameters to Server**

The IIapi\_modifyConnect() function sends the connection parameters, assigned in prior calls to IIapi\_setConnectParam(), to the server. Once sent, parameters are cleared so as to not affect subsequent calls to IIapi\_modifyConnect(). This function can be called only when no transaction is active on the specified connection.

**Note:** An Ingres DBMS Server accepts only IIAPI\_CP\_SHARED\_SYS\_UPDATE once a connection is established.

This function has the following syntax:

```
II_VOID IIapi_modifyConnect (IIAPI_MODCONNPARM *modifyConnParm);
```

```
typedef struct _IIAPI_MODCONNPARM
{
    IIAPI_GENPARM  mc_genParm;
    II_PTR         mc_connHandle;
} IIAPI_MODCONNPARM;
```

This function has the following parameters:

**mc\_genParm****Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

**mc\_connHandle****Type:** input

Specifies the connection handle, identifying the connection to the target server, or the environment handle.

## IIapi\_position() Function—Position Cursor and Initiate Row Retrieval

The IIapi\_position() function positions a cursor and requests some number of rows to be retrieved. The data associated with the requested rows can be retrieved with IIapi\_getColumns(), while the status of the cursor position request can be determined with IIapi\_getQueryInfo().

The position of the cursor is specified by a reference point and an offset from the reference point. The reference point can be the beginning of the result set, the current cursor position, or the end of the result set. A 0 offset corresponds to the reference point. Positive offsets reference positions that follow the reference point, while negative offsets reference positions that precede the reference point.

The number of rows to be retrieved can also be specified. A row count of 0 positions the cursor to the requested position without retrieving any rows. A row count of 1 retrieves the target row (if it exists) and leaves the cursor positioned on the target row. A row count greater than 1 retrieves a block of rows and leave the cursor on the last row/position accessed.

A 0 row count should be specified when the requested position is not associated with a row (references IIAPI\_POS\_BEGIN or IIAPI\_POS\_END with an offset of 0). A request for 0 rows should be followed by a call to IIapi\_getQueryInfo(). IIapi\_getColumns() returns a status of IIAPI\_ST\_NO\_DATA if preceded by a position request with a row count of 0.

A request for 1 or more rows should be followed by calling IIapi\_getColumns() for the same number of rows. This ensures that the operation started by IIapi\_position() is completed by the single call to IIapi\_getColumns(). It is possible, by carefully observing certain guidelines, to retrieve the rows requested using multiple calls to IIapi\_getColumns(). After the requested rows have been retrieved, the status of the position request can be determined using IIapi\_getQueryInfo(). If IIapi\_getQueryInfo() is called before all the requested rows have been retrieved, the remaining rows are discarded.

This function has the following syntax:

```
II_VOID IIapi_position( IIAPI_POSPARAM *posParm );
typedef struct _IIAPI_POSPARAM
{
    IIAPI_GENPARAM          po_genParm;
    II_PTR                  po_stmtHandle;
    II_UINT2                po_reference;
    II_INT4                 po_offset;
    II_INT2                 po_rowCount;
} IIAPI_POSPARAM;
```

This function has the following parameters:

**po\_genParm**

**Type:** input and delayed output

Specifies the generic parameters (see page 41).

**po\_stmtHandle**

**Type:** input

Specifies the statement handle identifying the query/cursor.

**po\_reference**

**Type:** input

Specifies the reference point from which the target position is calculated. Its value is one of the following:

**IIAPI\_POS\_BEGIN**

Beginning of the result set. Offset 0 corresponds to the initial position of the cursor (before the first row). Rows in the result set are at positive offsets from this reference point with the first row at offset 1.

**IIAPI\_POS\_CURRENT**

Current position of the cursor. Offset 0 corresponds to the current cursor position. Rows that precede the current cursor position are at negative offsets from this reference point with the prior row at offset -1. Rows that follow the current cursor position are at positive offsets from this reference point with the next row at offset 1.

**IIAPI\_POS\_END**

End of the result set. Offset 0 corresponds to the final position of a forward-only cursor (after the last row). Rows in the result set are at negative offsets from this reference point with the last row at offset -1.

**po\_offset**

**Type:** input

Specifies the row offset of the target position from the reference point.

**po\_rowCount**

**Type:** input

Specifies the number of rows to be retrieved. May be 0 to position cursor without retrieving rows.

## IIapi\_prepareCommit() Function—Begin Two-phase Commit of Transaction

The IIapi\_prepareCommit() function prepares to commit a distributed transaction started with IIapi\_query() or IIapi\_batch(), or restarted by IIapi\_connect().

IIapi\_prepareCommit() secures resources for a transaction in a two-phase commit situation. When IIapi\_prepareCommit() completes successfully, the server has allocated and secured all resources to commit the transaction. The application can then call IIapi\_commit() to commit the transaction or IIapi\_rollback() to abort the transaction.

Normally two-phase commits are desirable when multiple transactions are being committed as a unit. Each transaction is secured for the commit with IIapi\_prepareCommit(). When all transactions have been secured, the application calls IIapi\_commit() to finish the two-phase commit process. If some of the transactions cannot be secured, resulting in an unsuccessful completion of IIapi\_prepareCommit(), the application can call IIapi\_rollback() on each transaction to abort it.

In order to create or restart a distributed transaction, the application calls IIapi\_registerXID() to register a unique global transaction ID. This global transaction ID can be used for multiple transactions. The transaction ID handle returned by IIapi\_registerXID() can then be provided as input to IIapi\_query(), IIapi\_batch(), or IIapi\_connect() to create or restart a distributed transaction. The application used the transaction handle returned by IIapi\_query(), IIapi\_batch(), or IIapi\_connect() as input to IIapi\_prepareCommit() to specify the distributed transaction for which resources are to be secured.

IIapi\_prepareCommit() rejects any transaction handle representing a non-distributed or autocommit transaction.

This function has the following syntax:

```
II_VOID IIapi_prepareCommit (IIAPI_PREPCMPARM *prepCommitParm);
```

```
typedef struct _IIAPI_PREPCMPARM
{
    IIAPI_GENPARAM    pr_genParm;
    II_PTR            pr_tranHandle;
} IIAPI_PREPCMPARM;
```

This function has the following parameters:

### **pr\_genParm**

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

**pr\_tranHandle****Type:** input

Specifies the transaction handle identifying the transaction for the two-phase commit. The value for this handle is from the `qy_tranHandle` of `IIapi_query()`, `ba_tranHandle` of `IIapi_batch()`, or `co_tranHandle` parameter in `IIapi_connect()`.

**IIapi\_putColumns() Function—Send Data to Server to Copy Data from File to Database Table**

The `IIapi_putColumns()` function takes the requested number of columns of data from the application's buffers and copies it to a database table.

This function is preceded with the `IIapi_getCopyMap()` function, which describes the format and number of columns to be copied.

`IIapi_putColumns()` processes columns sequentially in the order that they appear in the row, as described in the `IIapi_getCopyMap()` functions. The application requests that a whole row be sent with one call to `IIapi_putColumns()`, provided it does not contain any long varchar or long byte columns. However, `IIapi_putColumns()` cannot span rows, so the application cannot request a number greater than the number of columns in a row.

If one of the columns is a long varchar or long byte that requires more than one segment to be sent, the application sends the columns individually with single calls to `IIapi_putColumns()`. For example, a row of ten columns with a long varchar or long byte spanning multiple segments as the fifth column is handled as follows. The application:

1. Sends four columns with a call to `IIapi_putColumns()`.
2. Sends one segment of the long varchar or long byte column with a call to `IIapi_putColumns()`.
3. Continues sending individual segments until all are sent.
4. Sends five columns with a call to `IIapi_putColumns()` to send the rest of the columns of the current row.

This function has the following syntax:

```
II_VOID IIApi_putColumns (IIAPI_PUTCOLPARM *putColParm);

typedef struct _IIAPI_PUTCOLPARM
{
    IIAPI_GENPARM      pc_genParm;
    II_PTR             pc_stmtHandle;
    II_LONG            pc_columnCount;
    IIAPI_DATAVALUE   *pc_columnData;
    II_BOOL            pc_moreSegments;
} IIAPI_PUTCOLPARM;
```

This function has the following parameters:

**pc\_genParm**

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

**pc\_stmtHandle**

**Type:** input

Specifies the statement handle identifying the copy from statement.

**pc\_columnCount**

**Type:** input

Specifies the number of columns the application is sending to the server.

**pc\_columnData**

**Type:** input

Specifies an array of buffers containing the data to be sent to the server. The number of buffers in this array must be equal to the value of pc\_columnCount.

**pc\_moreSegments**

**Type:** input

Indicates whether there is more data to be sent for a column of a long varchar or long byte data type. This parameter is set to TRUE if the data type is long varchar or long byte and more data must be sent for the current column; otherwise, the parameter is set to FALSE.

## IIapi\_putParms() Function—Send Query Statement Parameter Values to a Server

The IIapi\_putParms() function allows the application to send parameter data to a server at statement execution time.

This function is preceded with the IIapi\_setDescriptor() function, which describes the format and number of parameters to be sent to the server.

IIapi\_putParms() processes parameters sequentially in the order they appear in the query. The application sends all parameters with a single call to IIapi\_putParms(), provided the parameters do not contain any long varchar or long byte data types. The total number of parameters provided for a query must not be less than the number of parameters the query expects.

If one of the parameters is a long varchar or long byte that requires more than one segment to be sent, the application sends the parameters individually with single calls to IIapi\_putColumns(). For example, a row of ten parameters with a long varchar or long byte spanning multiple segments as the fifth parameter is handled as follows. The application:

1. Sends four parameters with a call to IIapi\_putParms().
2. Sends one segment of the long varchar or long byte parameter with a call to IIapi\_putParms().
3. Continues sending individual segments until all are sent.
4. Sends five parameters with a call to IIapi\_putParms() to send the rest of the parameters in the current row.

This function has the following syntax:

```
II_VOID IIapi_putParms (IIAPI_PUTPARMPARM *putParmParm);
```

```
typedef struct _IIAPI_PUTPARMPARM
{
    IIAPI_GENPARM    pp_genParm;
    II_PTR           pp_stmtHandle;
    II_LONG          pp_parmCount;
    IIAPI_DATAVALUE *pp_parmData;
    II_BOOL          pp_moreSegments;
} IIAPI_PUTPARMPARM;
```

This function has the following parameters:

### **pp\_genParm**

Type: **input and delayed output**

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

**pp\_stmtHandle**

**Type:** input

Specifies the statement handle identifying the query.

**pp\_parmCount**

**Type:** input

Specifies the number of parameters associated with the query.

**pp\_parmData**

**Type:** input

Specifies an array of buffers containing parameter data to be sent to the server. The number of buffers in this array must be equal to the value of pc\_parmCount.

**pp\_moreSegments**

**Type:** input

Indicates whether there is more data to be sent for the current parameter of a long varchar or long byte data type. This parameter is set to TRUE if the data type is long varchar or long byte and more data must be sent for the current parameter; otherwise, it is set to FALSE.

## IIapi\_query() Function—Begin Query Statement and Allocate Statement Handle

The IIapi\_query() function begins a query statement. IIapi\_query() allocates a statement handle. The statement handle must eventually be freed with the IIapi\_close() function.

An application enters parameters for the query statement with subsequent calls to the IIapi\_setDescriptor() and IIapi\_putParms() functions. It can also use the qy\_queryText parameter of IIapi\_query() to enter the syntax of the query statement. The qy\_queryText parameter should contain NULL if query text is not required.

**Note:** For a list of SQL statements that can be invoked by IIapi\_query(), see Accessing a DBMS Using SQL (see page 151).

When entering the syntax of SQL statements in the qy\_queryText parameter, see SQL Syntax (see page 158) and the *SQL Reference Guide*.

An application specifies the beginning of a transaction with the qy\_tranHandle input parameter. If this parameter is NULL or is a transaction ID handle created by IIapi\_registerXID(), a new transaction is begun and a transaction handle is allocated. Otherwise, this parameter must be an existing transaction handle and the statement is executed as part of the current transaction.

A transaction handle allocated by this function must be released using IIapi\_commit() or IIapi\_rollback(). If a transaction handle is allocated for a request which eventually fails, the transaction handle should be released using IIapi\_rollback().

Currently, the GCA protocol does not support multiple, concurrent non-cursor statements in the same transaction, or multiple concurrent transactions on the same connection.

This function has the following syntax:

```
II_VOID IIapi_query (IIAPI_QUERYPARM *queryParm);
```

```
typedef struct _IIAPI_QUERYPARM
{
    IIAPI_GENPARM    qy_genParm;
    II_PTR           qy_connHandle;
    IIAPI_QUERY_TYPE qy_queryType;
    II_BOOL          qy_parameters;
    II_CHAR          *qy_queryText;
    II_PTR           qy_tranHandle;
    II_PTR           qy_stmtHandle;
    II_ULONG         qy_flags;
} IIAPI_QUERYPARM;
```

This function has the following parameters:

**qy\_genParm**

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

**qy\_connHandle**

**Type:** input

Specifies the connection handle, identifying the connection to the target server.

**qy\_queryType**

**Type:** input

Specifies the type of query statement.

**Note:** If connecting to a DBMS, see Accessing a DBMS Using SQL (see page 151) for valid DBMS query type macros.

For a Name Server connection, this parameter must be IIAPI\_QT\_QUERY.

**qy\_queryText**

**Type:** input

Specifies the syntax of the query statement, if syntax is required. The query text should be a NULL-terminated string containing the query text with zero or more parameter markers.

**qy\_parameters**

**Type:** input

Indicates whether there are parameters to be sent with the query. TRUE if the application intends to send parameters with the query using IIapi\_setDescriptor() and IIapi\_putParms(); FALSE if there are no parameters to be sent with the query. OpenAPI requires some parameters for certain query types.

**Note:** For more information, see Queries, Parameters, and Query Data Correlation (see page 160).

If FALSE, the query is sent to the server for processing; otherwise, OpenAPI waits for calls to IIapi\_setDescriptor() and IIapi\_putParms() before sending the query to the server.

**qy\_tranHandle**

**Type:** input and immediate output

Indicates whether a new transaction should be started or whether an existing transaction should be used. If the parameter is NULL or a transaction ID handle created by `IIapi_registerXID()`, a new transaction should be started. If the parameter is a transaction handle previously returned by another query, the existing transaction should be used.

For a Name Server connection, this parameter must be an autocommit transaction handle returned by `IIapi_autoCommit()`.

**qy\_stmtHandle**

**Type:** immediate output

Specifies the statement handle identifying the query statement for future OpenAPI function calls.

**qy\_flags**

**Type:** input

Specifies query options. This parameter is ignored if the OpenAPI version is less than `IIAPI_VERSION_6`. The following flags can be set:

**IIAPI\_QF\_LOCATOR**

Enables retrieval of Blob/Clob locators instead of actual LOB data.

**IIAPI\_QF\_SCROLL**

Requests a scrollable cursor. Only applicable when `qy_queryType` is `IIAPI_QT_OPEN`.

## IIapi\_registerXID() Function—Reserve Unique ID for Two-phase Commit Transaction

The IIapi\_registerXID() function reserves a unique ID to begin a two-phase commit transaction with IIapi\_query() or to restart a previously-aborted two-phase commit transaction with IIapi\_connect(). An application must use IIapi\_registerXID() if a two-phase commit transaction is being used.

After this function is successfully completed, the output parameter rg\_tranIdHandle can be used as input for IIapi\_connect() and IIapi\_query() to specify a transaction. The transaction ID handle can be used on multiple transactions; it does not uniquely identify one transaction.

The transaction ID handle returned by this function must be release using IIapi\_releaseXID().

This function has the following syntax:

```
VOID IIapi_registerXID (IIAPI_REGXIDPARM *regXIDParm);
```

```
typedef struct _IIAPI_REGXIDPARM
{
    IIAPI_TRAN_ID  rg_tranID;
    PTR            rg_tranIdHandle;
    IIAPI_STATUS   rg_status;
} IIAPI_REGXIDPARM;
```

This function has the following parameters:

### **rg\_tranID**

**Type:** input

Specifies the parameter identifying a unique transaction ID. This parameter has the data type of IIAPI\_TRAN\_ID, which specifies a globally unique Ingres transaction identification.

### **rg\_tranIdHandle**

**Type:** immediate output

Specifies the transaction ID handle identifying the transaction ID registered with OpenAPI. This handle is used as input to IIapi\_query() and IIapi\_connect().

### **rg\_status**

**Type:** immediate output

Indicates the status of IIapi\_registerXID() upon completion. Its value can be one of the following:

- IIAPI\_ST\_SUCCESS
- IIAPI\_ST\_FAILURE

- IIAPI\_ST\_OUT\_OF\_MEMORY

## IIapi\_releaseEnv() Function—Release Resources Associated with Environment Handle

The IIapi\_releaseEnv() function frees an environment handle and any resources associated with the environment handle.

An application calls IIapi\_releaseEnv() to free an environment handle allocated by IIapi\_initialize(). Any active server connections associated with the environment handle are aborted and the respective connection handle is freed, as are all associated transaction, statement, and database event handles. While this function can be used to clean up quickly after a connection failure, applications are encouraged to continue calling IIapi\_close(), IIapi\_commit() or IIapi\_rollback() and IIapi\_disconnect() to cleanly shut down server connections. A warning status is returned if a connection is aborted.

The application can continue to make OpenAPI function calls after calling this function, but does so in the context of the default environment handle shared by all IIAPI\_VERSION\_1 applications. The application must still call IIapi\_terminate() to release global resources used by OpenAPI.

This function has the following syntax:

```
II_VOID IIapi_releaseEnv (IIAPI_RELENVPARAM *reEnvParm);

typedef struct _IIAPI_RELENVPARAM
{
    II_PTR          re_envHandle;
    IIAPI_STATUS   re_status;
} IIAPI_RELENVPARAM;
```

This function has the following parameters:

### re\_envHandle

**Type:** input

Specifies the environment handle for which resources should be freed.

### re\_status

**Type:** output

Specifies the status of the function upon its return. Its value is one of the following:

- IIAPI\_ST\_SUCCESS
- IIAPI\_ST\_FAILURE
- IIAPI\_ST\_NOT\_INITIALIZED
- IIAPI\_ST\_INVALID\_HANDLE

## IIapi\_releaseXID() Function—Release Unique ID for Two-phase Commit Transaction

The IIapi\_releaseXID() function releases the transaction ID reserved by IIapi\_registerXID(). It also frees the resources associated with the transaction ID handle.

This function has the following syntax:

```
II_VOID IIapi_releaseXID (IIAPI_RELXIDPARM *relXIDParm);
```

```
typedef struct _IIAPI_RELXIDPARM
{
    II_PTR          rl_tranIdHandle;
    IIAPI_STATUS    rl_status;
} IIAPI_RELXIDPARM;
```

This function has the following parameters:

### **rl\_tranIdHandle**

**Type:** input

Specifies the transaction ID handle identifying the unique transaction ID to be released.

### **rl\_status**

**Type:** immediate output

Specifies the status of IIapi\_releaseXID() upon completion. Its value can be one of the following:

- IIAPI\_ST\_SUCCESS
- IIAPI\_ST\_FAILURE
- IIAPI\_ST\_NOT\_INITIALIZED
- IIAPI\_ST\_INVALID\_HANDLE

## IIapi\_rollback() Function—Roll Back a Transaction

The IIapi\_rollback() function rolls back a transaction started with IIapi\_query() or IIapi\_batch(), or restarted by IIapi\_connect(). It also frees the transaction handle if the rb\_savePointHandle parameter is NULL. If the rb\_savePointHandle parameter is the savepoint handle returned by the IIapi\_savePoint() function, the transaction is rolled back to the savepoint and the transaction handle remains valid.

Before rolling back a transaction, the application must call IIapi\_close() to release all statement handles associated with the transaction.

This function has the following syntax:

```
II_VOID IIapi_rollback (IIAPI_ROLLBACKPARAM *rollbackParm);

typedef struct _IIAPI_ROLLBACKPARAM
{
    IIAPI_GENPARAM    rb_genParm;
    II_PTR            rb_tranHandle;
    II_PTR            rb_savePointHandle;
} IIAPI_ROLLBACKPARAM;
```

This function has the following parameters:

### rb\_genParm

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

### rb\_tranHandle

**Type:** input

Specifies the transaction handle identifying the transaction to be rolled back. Its value is either from qy\_tranHandle of IIapi\_query(), ba\_tranHandle of IIapi\_batch(), or co\_tranHandle parameter of IIapi\_connect().

### rb\_savePointHandle

**Type:** input

Specifies an optional savepoint for the current rollback. This parameter contains a handle of the savepoint, returned by the IIapi\_savePoint() function. If there is no savepoint for the current rollback, this parameter is NULL.

## IIapi\_savePoint() Function—Mark Savepoint in a Transaction for Partial Rollback

The IIapi\_savePoint() function marks a savepoint in a transaction for a partial rollback with the IIapi\_rollback() function.

The sp\_savePointHandle parameter that is output from this function is unique to the transaction and remains valid until the transaction is committed or completely rolled back.

This function has the following syntax:

```
II_VOID IIapi_savePoint (IIAPI_SAVEPTPARM *savePtParm);

typedef struct _IIAPI_SAVEPTPARM
{
    IIAPI_GENPARM  sp_genParm;
    II_PTR         sp_tranHandle;
    II_CHAR        *sp_savePoint;
    II_PTR         sp_savePointHandle;
} IIAPI_SAVEPTPARM;
```

This function has the following parameters:

### sp\_genParm

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

### sp\_tranHandle

**Type:** input

Specifies the transaction handle identifying the transaction for which the savepoint is specified. Its value is either from qy\_tranHandle of IIapi\_query(), ba\_tranHandle of IIapi\_batch(), or co\_tranHandle parameter of IIapi\_connect().

### sp\_savePoint

**Type:** input

Specifies the identifier for the savepoint marker. It is a NULL-terminated string unique within the transaction.

### sp\_savePointHandle

**Type:** delayed output

Specifies the savepoint handle identifying the savepoint marker.

## IIapi\_scroll() Function—Scroll (Position) Cursor and Initiate Row Retrieval

The IIapi\_scroll() function permits a cursor to be scrolled through a result set. The requested rows can be retrieved with IIapi\_getColumns(), while the status of the scroll request can be determined with IIapi\_getQueryInfo().

The scrolling operation is specified by a fetch orientation. In most cases, the fetch orientation specifies a specific row/position. In the case of absolute and relative orientations, an offset is used to determine the row/position relative to the start or end of the result set, or the current cursor position.

For fetch orientations other than IIAPI\_SCROLL\_BEFORE and IIAPI\_SCROLL\_AFTER, IIapi\_scroll() requests a single row to be retrieved. These requests should be followed by a call to IIapi\_getColumns(). After calling IIapi\_getColumns(), IIapi\_getQueryInfo() can be called to determine the status of the scroll request. If IIapi\_getQueryInfo() is called without calling IIapi\_getColumns(), the requested row data is discarded.

For fetch orientations IIAPI\_SCROLL\_BEFORE and IIAPI\_SCROLL\_AFTER, IIapi\_scroll() does not request a row to be retrieved; the cursor is simply positioned to the requested position. These requests should be followed by a call to IIapi\_getQueryInfo(). If one of these requests is followed by a call to IIapi\_getColumns(), a status of IIAPI\_ST\_NO\_DATA is returned.

This function has the following syntax:

```
II_VOID IIapi_scroll( IIAPI_SCROLLPARAM *scrollParm );

typedef struct _IIAPI_SCROLLPARAM
{
    IIAPI_GENPARAM    sl_genParm;
    II_PTR            sl_stmtHandle;
    II_UINT2          sl_orientation;
    II_INT4           sl_offset;
} IIAPI_SCROLLPARAM;
```

This function has the following parameters:

### **sl\_genParm**

**Type:** input and delayed output

Specifies the generic parameters (see page 41).

### **sl\_stmtHandle**

**Type:** input

Specifies the statement handle identifying the query/cursor.

### **sl\_orientation**

**Type:** input

Specifies the scrolling operation to be performed. Its value is one of the following:

**IIAPI\_SCROLL\_BEFORE**

Positions the cursor at the start of the result set (before the first row, initial position of a cursor).

**IIAPI\_SCROLL\_FIRST**

Positions the cursor at the first row of the result set.

**IIAPI\_SCROLL\_PRIOR**

Positions the cursor at the preceding row of the result set.

**IIAPI\_SCROLL\_CURRENT**

Positions the cursor at the current position in the result set. Retrieves the current row without moving the cursor.

**IIAPI\_SCROLL\_NEXT**

Positions the cursor at the following row of the result set.

**IIAPI\_SCROLL\_LAST**

Positions the cursor at the last row of the result set.

**IIAPI\_SCROLL\_AFTER**

Positions the cursor at the end of the result set (after last row, final position of a non-scrollable cursor).

**IIAPI\_SCROLL\_ABSOLUTE**

Positions the cursor at a specific row of the result set. The position is determined by the value of `sl_offset`. An offset of 0 specifies the beginning of the result set (same as `IIAPI_SCROLL_BEFORE`). A positive offset specifies the row position from the start of the result set with 1 being the first row. A negative offset specifies the row position from the end of the result set with -1 being the last row.

**IIAPI\_SCROLL\_RELATIVE**

Positions the cursor relative to the current cursor position. The position is determined by the value of `sl_offset`. An offset of 0 specifies the current position in the result set (same as `IIAPI_SCROLL_CURRENT`). A positive offset specifies the row positions following the current cursor position with 1 being the next row. A negative offset specifies the row positions preceding the current cursor position with -1 being the prior row.

**sl\_offset**

**Type:** input

Specifies the row offset of the target position when `sl_orientation` is `IIAPI_SCROLL_ABSOLUTE` or `IIAPI_SCROLL_RELATIVE`. Ignored for other values of `sl_orientation`.

## IIapi\_setConnectParam() Function—Assign Connection Parameter and Value to a Connection

The IIapi\_setConnectParam() function allows the application to assign connection parameters to a connection. These parameters are sent to the server when IIapi\_connect() or IIapi\_modifyConnect() are called. One parameter and associated value are assigned for each call to IIapi\_setConnectParam().

This function creates a new connection handle if called with sc\_connHandle set to NULL or an environment handle returned by IIapi\_initialize(). The returned connection handle can be used in subsequent calls to assign additional connection parameters to the connection. The connection handle can then be used to call IIapi\_connect() to establish the connection to the server with the desired connection parameters. The connection handle is released using IIapi\_disconnect() or IIapi\_abort().

Once a connection has been established with IIapi\_connect(), IIapi\_setConnectParam() can be called with the connection handle to assign new connection parameters or to modify the parameters established with IIapi\_connect(). Parameters assigned after calling IIapi\_connect() are sent to the server by calling IIapi\_modifyConnect().

This function has the following syntax:

```
II_VOID IIapi_setConnectParam (IIAPI_SETCONPRMPARM *setConPrmParm);
```

```
typedef struct _IIAPI_SETCONPRMPARM
{
    IIAPI_GENPARM  sc_genParm;
    II_PTR         sc_connHandle;
    II_LONG        sc_paramID;
    II_PTR         sc_paramValue;
} IIAPI_SETCONPRMPARM;
```

**Note:** Only IIAPI\_CP\_SHARED\_SYS\_UPDATE is accepted by an Ingres DBMS Server once a connection is established.

This function has the following parameters:

### sc\_genParm

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

### **sc\_connHandle**

**Type:** input and immediate output

Specifies the input and output for a connection handle.

The input is NULL for a new connection associated with the default environment, an environment handle for a new connection associated with the environment, or an existing connection handle.

The output is a connection handle that can be used in subsequent calls to `IIapi_setConnectParam()` and `IIapi_connect()`.

### **sc\_paramID**

**Type:** input

Specifies the parameter being assigned. For valid parameter ID macros, see the following table.

### **sc\_paramValue**

**Type:** input

Specifies the value of the parameter being assigned. The data type and description of the value associated with each parameter ID are as follows (valid parameter ID macros for the `sc_paramID` parameter).

#### **IIAPI\_CP\_APP\_ID**

**Data type:** II\_CHAR

The role ID of the application. If a role password is required, the parameter value should be specified as *"role/password"*.

#### **IIAPI\_CP\_APPLICATION**

**Data type:** II\_LONG

Specifies the value for an internal flag used for debugging a corrupted database. This flag is for Ingres internal applications; it must not be used by non-Ingres applications.

#### **IIAPI\_CP\_CENTURY\_BOUNDARY**

**Data type:** II\_LONG

Specifies the date century boundary.

#### **IIAPI\_CP\_CHAR\_WIDTH**

**Data type:** II\_LONG

Specifies the maximum length of a character (`IIAPI_BYTE_TYPE`, `IIAPI_CHA_TYPE`, `IIAPI_CHR_TYPE`, or `IIAPI_VCH_TYPE`) data type.

**IIAPI\_CP\_DATE\_ALIAS**

**Data type:** II\_CHAR

Specifies the actual data type to be used in place of the "date" data type alias. Its value is one of the following:

IIAPI\_CPV\_INGDATE  
IIAPI\_CPV\_ANSIDATE

**IIAPI\_CP\_DATE\_FORMAT**

**Data type:** II\_LONG

Specifies the date format. Its value can be one of the following:

IIAPI\_CPV\_DFRMT\_US  
IIAPI\_CPV\_DFRMT\_MULTI  
IIAPI\_CPV\_DFRMT\_FINNISH  
IIAPI\_CPV\_DFRMT\_ISO  
IIAPI\_CPV\_DFRMT\_GERMAN  
IIAPI\_CPV\_DFRMT\_YMD  
IIAPI\_CPV\_DFRMT\_MDY  
IIAPI\_CPV\_DFRMT\_DMY

**IIAPI\_CP\_DBMS\_PASSWORD**

**Data type:** II\_CHAR

Specifies the DBMS password for the user.

**IIAPI\_CP\_DECIMAL\_CHAR**

**Data type:** II\_CHAR

Specifies the character identifier of decimal data.

**IIAPI\_CP\_EFFECTIVE\_USER**

**Data type:** II\_CHAR

Specifies the effective user name for the new session.

**IIAPI\_CP\_EXCLUSIVE\_LOCK**

**Data type:** II\_BOOL

Indicates whether the database should be locked for exclusive use of this application. If it should be locked, this parameter is TRUE; otherwise, it is FALSE.

**IIAPI\_CP\_EXCLUSIVE\_SYS\_UPDATE**

**Data type:** II\_BOOL

Indicates whether the system catalog should be updated with an exclusive lock. If so, this parameter is TRUE; otherwise, it is FALSE.

### **IIAPI\_CP\_FLOAT4\_PRECISION**

**Data type:** II\_LONG

Specifies the precision of a float4 (IIAPI\_FLT\_TYPE) data type.

### **IIAPI\_CP\_FLOAT4\_STYLE**

**Data type:** II\_CHAR

Specifies the output column format for the four-byte floating-point data type. Its value can be one of the following:

- IIAPI\_CPV\_EXPONENTIAL-exponential format
- IIAPI\_CPV\_FLOATF-floating-point format
- IIAPI\_CPV\_FLOATDEC-floating-point format, decimal alignment not guaranteed
- IIAPI\_CPV\_FLOATDECALIGN-floating-point format, decimal alignment guaranteed

**Default:** IIAPI\_CPV\_FLOATDEC

### **IIAPI\_CP\_FLOAT4\_WIDTH**

**Data type:** II\_LONG

Specifies the length of a float4 (IIAPI\_FLT\_TYPE) data type.

### **IIAPI\_CP\_FLOAT8\_PRECISION**

**Data type:** II\_LONG

Specifies the precision of a float8 (IIAPI\_FLT\_TYPE) data type.

### **IIAPI\_CP\_FLOAT8\_STYLE**

**Data type:** II\_CHAR

Specifies the output column format for an eight-byte floating-point data type. Its value can be one of the following:

- IIAPI\_CPV\_EXPONENTIAL-exponential format
- IIAPI\_CPV\_FLOATF-floating-point format
- IIAPI\_CPV\_FLOATDEC-(default) floating-point format, decimal alignment not guaranteed
- IIAPI\_CPV\_FLOATDECALIGN-floating-point format, decimal alignment guaranteed

**Default:** IIAPI\_CPV\_FLOATDEC

### **IIAPI\_CP\_FLOAT8\_WIDTH**

**Data type:** II\_LONG

Specifies the length of a float8 (IIAPI\_FLT\_TYPE) data type.

**IIAPI\_CP\_GATEWAY\_PARM****Data type:** II\_CHAR

Specifies Enterprise Access parameters. This parameter can be set multiple times.

**IIAPI\_CP\_GROUP\_ID****Data type:** II\_CHAR

Specifies the group ID of the user.

**IIAPI\_CP\_INT1\_WIDTH****Data type:** II\_LONG

Specifies the length of an integer1 (IIAPI\_INT\_TYPE) data type.

**IIAPI\_CP\_INT2\_WIDTH****Data type:** II\_LONG

Specifies the length of an integer2 (IIAPI\_INT\_TYPE) data type.

**IIAPI\_CP\_INT4\_WIDTH****Data type:** II\_LONG

Specifies the length of an integer4 (IIAPI\_INT\_TYPE) data type.

**IIAPI\_CP\_INT8\_WIDTH****Data type:** II\_LONG

Specifies the length of an integer8 (IIAPI\_INT\_TYPE) data type (ignored if connection level, co\_apiLevel, is less than IIAPI\_LEVEL\_3).

**IIAPI\_CP\_LOGIN\_LOCAL****Data type:** II\_BOOL

Determines how the connection user ID and password are used when a VNODE is included in the target database string. If set to TRUE, the user ID and password are used to locally access the VNODE and the VNODE login information is used to establish the DBMS connection. If set to FALSE, the process user ID is used to access the VNODE and the connection user ID and password are used in place of the VNODE login information to establish the DBMS connection. This parameter is ignored if no VNODE is included in the target database string. The default is FALSE.

### **IIAPI\_CP\_MATH\_EXCP**

**Data type:** II\_CHAR

Specifies the type of error message to be returned because of a numerical overflow/underflow at the server. Its value can be one of the following:

- IIAPI\_CPV\_RET\_FATAL-return fatal message
- IIAPI\_CPV\_RET\_WARN-return warning message
- IIAPI\_CPV\_RET\_IGNORE-ignore exceptions

### **IIAPI\_CP\_MISC\_PARM**

**Data type:** II\_CHAR

Specifies miscellaneous parameters. This parameter can be set multiple times.

### **IIAPI\_CP\_MONEY\_LORT**

**Data type:** II\_LONG

Indicates a leading or trailing money sign. Valid values are one of the following:

- IIAPI\_CPU\_MONEY\_LEAD\_SIGN
- IIAPI\_CPU\_MONEY\_TRAIL\_SIGN

### **IIAPI\_CP\_MONEY\_PRECISION**

**Data type:** II\_LONG

Specifies the precision of the money (IIAPI\_MNY\_TYPE) data type.

### **IIAPI\_CP\_MONEY\_SIGN**

**Data type:** II\_CHAR

Specifies the money sign.

### **IIAPI\_CP\_NATIVE\_LANG**

**Data type:** II\_CHAR

Specifies the name of the language used for error reporting. This is converted to a language code prior to sending to the server. Can be set to NULL to obtain the default installation language.

### **IIAPI\_CP\_NATIVE\_LANG\_CODE**

**Data type:** II\_LONG

Specifies the language code of the native language used for error reporting. IIAPI\_CP\_NATIVE\_LANG can be used when the language code is not known.

**IIAPI\_CP\_NUMERIC\_TREATMENT****Data type:** II\_CHAR

Specifies the treatment of the numeric literal. Its value can be one of the following:

- IIAPI\_CPV\_DECASFLOAT-treat decimal literal as float
- IIAPI\_CPV\_DECASDEC-treat decimal literal as decimal

**IIAPI\_CP\_RESULT\_TBL****Data type:** II\_LONG

Specifies the default result table structure. Its value can be one of the following:

IIAPI\_CPV\_ISAM  
IIAPI\_CPV\_CISAM  
IIAPI\_CPV\_HEAP  
IIAPI\_CPV\_CHEAP  
IIAPI\_CPV\_BTREE  
IIAPI\_CPV\_CBTREE  
IIAPI\_CPV\_HASH  
IIAPI\_CPV\_CHASH

**IIAPI\_CP\_SECONDARY\_INX****Data type:** II\_LONG

Specifies the default secondary index structure. Its value can be one of the following:

IIAPI\_CPV\_ISAM  
IIAPI\_CPV\_CISAM  
IIAPI\_CPV\_BTREE  
IIAPI\_CPV\_CBTREE  
IIAPI\_CPV\_HASH  
IIAPI\_CPV\_CHASH

**IIAPI\_CP\_SERVER\_TYPE****Data type:** II\_LONG

Specifies the type of server function desired. Its value can be one of the following:

IIAPI\_CPV\_SVNORMAL  
IIAPI\_CPV\_MONITOR

**IIAPI\_CP\_SHARED\_SYS\_UPDATE**

**Data type:** II\_BOOL

Indicates whether the system catalog should be updated with a shared lock. If so, this parameter is TRUE; otherwise, it is FALSE.

**IIAPI\_CP\_STRING\_TRUNC**

**Data type:** II\_CHAR

Specifies the parameters for specifying how to handle truncation when a string value is being inserted into a column that is too short to contain the value. Its value can be one of the following:

- IIAPI\_CPV\_RET\_FATAL-the string is not inserted, an error is issued, and the statement is aborted
- IIAPI\_CPV\_RET\_IGNORE-the string is truncated and inserted. No error or warning is issued

**IIAPI\_CP\_TIMEZONE**

**Data type:** II\_CHAR

Specifies the time zone of the application.

**IIAPI\_CP\_TXT\_WIDTH**

**Data type:** II\_LONG

Specifies the maximum length of a text (IIAPI\_TXT\_TYPE) data type.

**IIAPI\_CP\_WAIT\_LOCK**

**Data type:** II\_BOOL

Indicates whether the application is willing to wait for a lock on the database to be released. If so, this parameter is TRUE; otherwise, it is FALSE.

## IIapi\_setDescriptor() Function—Send Information About Data Format

The IIapi\_setDescriptor() function sends information about the format of data that will be provided with the IIapi\_putParms() or IIapi\_putColumns() function. Parameters to IIapi\_setDescriptor() contain the number of columns being inserted or copied to the database, as well as their data type, length, precision, and scale.

Sends information to the server about the format of data to be provided in subsequent calls to IIapi\_putParms() and IIapi\_putColumns().

This function has the following syntax:

```
II_VOID IIapi_setDescriptor
    (IIAPI_SETDESCRPARAM *setDescrParm);

typedef struct _IIAPI_SETDESCRPARAM
{
    IIAPI_GENPARAM    sd_genParm;
    II_PTR            sd_stmtHandle;
    II_LONG           sd_descriptorCount;
    IIAPI_DESCRIPTOR *sd_descriptor;
} IIAPI_SETDESCRPARAM;
```

This function has the following parameters:

### **sd\_genParm**

**Type:** input and delayed output

Specifies the generic parameters. For a description, see Generic Parameters (see page 41).

### **sd\_stmtHandle**

**Type:** input

Specifies the statement handle identifying the query.

### **sd\_descriptorCount**

**Type:** input

Specifies the number of columns being described by sd\_descriptor.

### **sd\_descriptor**

**Type:** input

Specifies an array of buffers containing the descriptions of the columns. The number of buffers in this array must be equal to the sd\_descriptorCount value. Each buffer includes the data type, length, precision, and scale.

## IIapi\_setEnvParam() Function—Assign an Environment Parameter and Value in Environment Handle

The IIapi\_setEnvParam() function allows the application to override the default environment logical settings established by the IIapi\_initialize() for a specific environment handle.

Settings for environment parameters, which correspond to connection parameters, are used as the default connection parameter values unless specifically overridden by IIapi\_setConnectParam().

Environment parameters also affect the conversions performed by IIapi\_formatData() and control interactions between OpenAPI and the application.

This function has the following syntax:

```
II_VOID IIapi_setEnvParam (IIAPI_SETENVPRMPARM *setEnvPrmParm);
```

```
typedef struct _IIAPI_SETENVPRMPARAM
{
    II_PTR      se_envHandle;
    II_LONG     se_paramID
    II_PTR      se_paramValue
    IIAPI_STATUS se_status
} IIAPI_SETENVPRMPARM;
```

This function has the following parameters:

### **se\_envHandle**

**Type:** input

Specifies the environment handle for the parameter being assigned.

### **se\_paramID**

**Type:** input

Specifies the parameter being assigned. For valid parameter ID macros, see the following list.

### **se\_paramValue**

**Type:** input

Specifies the value of the parameter being assigned. For the data type and description of the value associated with each parameter ID, see the following list.

**se\_status**

**Type:** output

Specifies the status of the function upon its return. Its value is one of the following:

IIAPI\_ST\_SUCCESS  
IIAPI\_ST\_FAILURE  
IIAPI\_ST\_NOT\_INITIALIZED  
IIAPI\_ST\_INVALID\_HANDLE

The following table lists valid parameter ID macros for the `se_paramID` parameter:

**IIAPI\_EP\_CHAR\_WIDTH**

**Data type:** II\_LONG

Specifies the maximum length of a character data type (IIAPI\_BYTE\_TYPE, IIAPI\_CHA\_TYPE, IIAPI\_CHR\_TYPE, IIAPI\_VCH\_TYPE).

**IIAPI\_EP\_TXT\_WIDTH**

**Data type:** II\_LONG

Specifies the maximum length of a text data type (IIAPI\_TXT\_TYPE).

**IIAPI\_EP\_INT1\_WIDTH**

**Data type:** II\_LONG

Specifies the length of an integer1 data type (IIAPI\_INT\_TYPE).

**IIAPI\_EP\_INT2\_WIDTH**

**Data type:** II\_LONG

Specifies the length of an integer2 data type (IIAPI\_INT\_TYPE).

**IIAPI\_EP\_INT4\_WIDTH**

**Data type:** II\_LONG

Specifies the length of an integer4 data type (IIAPI\_INT\_TYPE).

**IIAPI\_EP\_FLOAT4\_WIDTH**

**Data type:** II\_LONG

Specifies the length of a float4 data type (IIAPI\_FLT\_TYPE).

**IIAPI\_EP\_INT8\_WIDTH**

**Data type:** II\_LONG

Specifies the length of an integer8 (IIAPI\_INT\_TYPE) data type (ignored if connection level, `co_apiLevel`, is less than `IIAPI_LEVEL_3`).

**IIAPI\_EP\_FLOAT8\_WIDTH**

**Data type:** II\_LONG

Specifies the length of a float8 data type (IIAPI\_FLT\_TYPE).

**IIAPI\_EP\_FLOAT4\_PRECISION**

**Data type:** II\_LONG

Specifies the precision of a float4 data type (IIAPI\_FLT\_TYPE).

**IIAPI\_EP\_FLOAT8\_PRECISION**

**Data type:** II\_LONG

Specifies the precision of a float8 data type (IIAPI\_FLT\_TYPE).

**IIAPI\_EP\_MONEY\_PRECISION**

**Data type:** II\_LONG

Specifies the precision of a money data type (IIAPI\_MNY\_TYPE).

**IIAPI\_EP\_FLOAT4\_STYLE**

**Data type:** II\_CHAR

Specifies the format style of a float4 data type (IIAPI\_FLT\_TYPE).

**IIAPI\_EP\_FLOAT8\_STYLE**

**Data type:** II\_CHAR

Specifies the format style of a float8 data type (IIAPI\_FLT\_TYPE).

**IIAPI\_EP\_NUMERIC\_TREATMENT**

**Data type:** II\_CHAR

Specifies the treatment of the numeric literal. Its value is one of the following:

- IIAPI\_EPV\_DECASFLOAT
- IIAPI\_EPV\_DECASDEC

**IIAPI\_EP\_MONEY\_SIGN**

**Data type:** II\_CHAR

Specifies the money sign of an IIAPI\_MNY\_TYPE data type.

- IIAPI\_EP\_MONEY\_LORT

**Data type:** II\_LONG

Specifies the leading or trailing money sign. Its value is one of the following:

- IIAPI\_CPU\_MONEY\_LEAD\_SIGN
- IIAPI\_CPU\_MONEY\_TRAIL\_SIGN

**IIAPI\_EP\_DECIMAL\_CHAR**

**Data type:** II\_CHAR

Specifies the decimal character.

**IIAPI\_EP\_MATH\_EXCP**

**Data type:** II\_CHAR

Specifies the treatment of math exceptions. Its value is one of the following:

IIAPI\_EPV\_RET\_FATAL  
IIAPI\_EPV\_RET\_WARN  
IIAPI\_EPV\_RET\_IGNORE

**IIAPI\_EP\_STRING\_TRUNC**

**Data type:** II\_CHAR

Specifies the truncation of strings.

**IIAPI\_EP\_DATE\_ALIAS**

**Data type:** II\_CHAR

Specifies the actual data type to be used in place of the "date" data type alias. Its value is one of the following:

IIAPI\_EPV\_INGDATE  
IIAPI\_EPV\_ANSIDATE

**IIAPI\_EP\_DATE\_FORMAT**

**Data type:** II\_LONG

Specifies the date format. Its value is one of the following:

IIAPI\_EPV\_DFRMT\_US  
IIAPI\_EPV\_DFRMT\_MULT  
IIAPI\_EPV\_DFRMT\_FINNISH  
IIAPI\_EPV\_DFRMT\_ISO  
IIAPI\_EPV\_DFRMT\_GERMAN  
IIAPI\_EPV\_DFRMT\_YMD  
IIAPI\_EPV\_DFRMT\_MDY  
IIAPI\_EPV\_DFRMT\_DMY

**IIAPI\_EP\_TIMEZONE**

**Data type:** II\_CHAR

Specifies the name of the time zone.

**IIAPI\_EP\_NATIVE\_LANG**

**Data type:** II\_CHAR

Specifies the name of the language used for error reporting. This is converted to a language code prior to sending it to the server.

**IIAPI\_EP\_NATIVE\_LANG\_CODE**

**Data type:** II\_LONG

Specifies the language code of the native language used for error reporting. IIAPI\_EP\_NATIVE\_LANG can be used when the language code is not known.

**IIAPI\_EP\_CENTURY\_BOUNDARY**

**Data type:** II\_LONG

Specifies the date century boundary.

**IIAPI\_EP\_SEGMENT\_LEN**

**Data type:** II\_LONG

Specifies the value used as the ds\_length value when describing a BLOB column using IIapi\_getDescriptor(), and will therefore be used as the segment length when retrieving a BLOB using IIapi\_getColumns().

**IIAPI\_EP\_TRACE\_FUNC**

**Data type:** II\_PTR

Specifies the callback function pointer to the application for trace messages. For more information on setting this value, see Environment Parameter IIAPI\_EP\_TRACE\_FUNC (see page 121).

**IIAPI\_EP\_EVENT\_FUNC**

**Data type:** II\_PTR

Specifies the callback function pointer to the application for database event notification. For more information on setting this value, see Environment Parameter IIAPI\_EP\_EVENT\_FUNC (see page 122).

**IIAPI\_EP\_CAN\_PROMPT**

**Data type:** II\_PTR

Specifies the callback function pointer to the application for database password prompting. For more information on setting this value, see Environment Parameter IIAPI\_EP\_CAN\_PROMPT (see page 119).

**More information:**

Environment Parameter IIAPI\_EP\_CAN\_PROMPT (see page 119)

Environment Parameter IIAPI\_EP\_TRACE\_FUNC (see page 121)

Environment Parameter IIAPI\_EP\_EVENT\_FUNC (see page 122)

## Environment Parameter `IIAPI_EP_CAN_PROMPT`

An application can set `IIAPI_EP_CAN_PROMPT` in the `IIapi_setEnvParam()` function by providing the address of an application function to call back. OpenAPI invokes this application function to receive a reply from the application for the prompt message from the server using the following C structure:

```
typedef struct _IIAPI_PROMPTPARAM
{
    II_PTR      pd_envHandle;
    II_PTR      pd_connHandle;
    II_LONG     pd_flags;
    II_LONG     pd_timeout;
    II_UINT2    pd_msg_len;
    II_CHAR     *pd_message;
    II_UINT2    pd_max_reply;
    II_LONG     pd_rep_flags;
    II_UINT2    pd_rep_len;
    II_CHAR     *pd_reply;
} IIAPI_PROMPTPARAM;
```

The syntax of the application function is as follows:

```
II_VOID promptFunc(IIAPI_PROMPTPARAM *parm);
```

The application must invoke `IIapi_setEnvParam()` by setting `se_paramID` to `IIAPI_EP_CAN_PROMPT` and `se_paramValue` to the address of an application-supplied function that matches the calling sequence of the `promptFunc()` template function shown.

The `IIAPI_PROMPTPARAM` parameters are described in the following table:

### **pd\_envHandle**

**Type:** input

Specifies the environment handle associated with the connection. It is NULL if the connection belongs to the default environment associated with `IIAPI_VERSION_1` applications.

### **pd\_connHandle**

**Type:** input

Specifies the connection handle identifying the connection.

### **pd\_flags**

**Type:** input

Specifies the prompt flags. Its value is one of the following:

- `IIAPI_PR_NOECHO`
- `IIAPI_PR_TIMEOUT`

IIAPI\_PR\_NOECHO will be set if user input should not be echoed, generally indicating a request for a password. IIAPI\_PR\_TIMEOUT will be set if the pd\_timeout parameter has a valid value.

**pd\_timeout**

**Type:** input

Specifies the maximum time, in seconds, to wait for user input. Ignore this parameter if IIAPI\_PR\_TIMEOUT is not set in pd\_flags.

Support for timeouts is platform-dependent.

**pd\_msg\_len**

**Type:** input

Specifies the length of the prompt message in pd\_message.

**pd\_message**

**Type:** input

Specifies the message to be displayed when prompting for user input.

**pd\_max\_reply**

**Type:** input

Specifies the maximum length of user input to be returned in pd\_reply.

**pd\_rep\_flags**

**Type:** output

Specifies the response flags. The valid value is IIAPI\_REPLY\_TIMEOUT. Set this value if timeout occurred waiting for user response. (See IIAPI\_PR\_TIMEOUT in the pd\_flags description.)

**pd\_rep\_len**

**Type:** output

Specifies the length of the user response in pd\_reply.

**pd\_reply**

**Type:** output

Specifies the user response.

## Environment Parameter `IIAPI_EP_TRACE_FUNC`

An application can set `IIAPI_EP_TRACE_FUNC` in the `IIapi_setEnvParam()` function by providing an address of an application function to call back. OpenAPI invokes this application function to send trace information from the server to the application using the following C structure:

```
typedef struct _IIAPI_TRACEPARAM
{
    II_PTR      tr_envHandle;
    II_PTR      tr_connHandle;
    II_INT4     tr_length;
    II_CHAR     *tr_message;
} IIAPI_TRACEPARAM;
```

The syntax of the application function is as follows:

```
II_VOID traceFunc(IIAPI_PROMPTPARAM *parm);
```

The application must invoke `IIapi_setEnvParam()` by providing `se_paramID` to be `IIAPI_EP_TRACE_FUNC` and `se_paramValue` to the address of an application-supplied function that matches the calling sequence of the `traceFunc()` function template shown in this section.

The `IIAPI_TRACEPARAM` parameters are described in the following table:

### **tr\_envHandle**

**Type:** input

Specifies the environment handle associated with the connection. It is NULL if the connection belongs to the default environment associated with `IIAPI_VERSION_1` applications.

### **tr\_connHandle**

**Type:** input

Specifies the connection handle identifying the connection.

### **tr\_length**

**Type:** input

Specifies the length of the output trace message in `tr_message`.

### **tr\_message**

**Type:** input

Specifies the output trace message.

Trace output can also be directed to a file. For more information, see `II_API_SET` environment variable (see page 178).

## Environment Parameter `IIAPI_EP_EVENT_FUNC`

An application can set `IIAPI_EP_EVENT_FUNC` in the `IIapi_setEnvParam()` function by providing an address of an application function to callback. OpenAPI invokes this application function to send information on database events which failed to match an existing OpenAPI event handle (and would otherwise be ignored) to the application using the following C structure:

```
typedef struct _IIAPI_EVENTPARM
{
    II_PTR          ev_envHandle;
    II_PTR          ev_connHandle;
    II_CHAR         *ev_eventName;
    II_CHAR         *ev_eventOwner;
    II_CHAR         *ev_eventDB;
    IIAPI_DATAVALUE ev_eventTime;
} IIAPI_EVENTPARM;
```

The syntax of the application function is as follows:

```
II_VOID eventFunc(IIAPI_EVENTPARM *parm);
```

The application must invoke `IIapi_setEnvParam()` by providing `sc_paramID` to be `IIAPI_EP_EVENT_FUNC` and `sc_paramValue` to the address of an application-supplied function that matches the calling sequence of the `eventFunc()` function template shown in this section.

The `IIAPI_EVENTPARM` parameters are described in the following table:

### **ev\_envHandle**

**Type:** input

Specifies the environment handle associated with the connection. It is NULL if the connection is associated with the default environment assigned to `IIAPI_VERSION_1` applications.

### **ev\_connHandle**

**Type:** input

Specifies the connection handle identifying the connection.

### **ev\_eventName**

**Type:** input

Specifies the name of the database event.

### **ev\_eventOwner**

**Type:** input

Specifies the owner of the database event.

**ev\_eventDB****Type:** input

Specifies the name of the database that raised the event.

**ev\_eventTime****Type:** input

Specifies the time the event occurred, stored as an IIAPI\_DTE\_TYPE data value.

## IIapi\_terminate() Function—Terminate OpenAPI

The IIapi\_terminate() function cleans up all OpenAPI resources. This function should be called when OpenAPI functions are no longer used in the application. If multiple calls to IIapi\_initialize() have been made, the OpenAPI is shut down only when the last corresponding call to this function is made; a warning status is returned otherwise.

This function has the following syntax:

```
II_VOID IIapi_terminate (IIAPI_TERMPARM *termParm);
```

```
typedef struct _IIAPI_TERMPARM
{
    IIAPI_STATUS  tm_status;
} IIAPI_TERMPARM;
```

This statement has the following parameter:

**tm\_status****Type:** output

The status of the function upon its return. Its value is one of the following:

- IIAPI\_ST\_SUCCESS
- IIAPI\_ST\_NOT\_INITIALIZED
- IIAPI\_ST\_WARNING

## IIapi\_wait() Function—Block Application Control Until Outstanding Operation Completes or User-defined Timeout Expires

The IIapi\_wait() function enables an application to give control to OpenAPI until an outstanding task completes. Because OpenAPI is asynchronous and many operating systems do not provide an interrupt mechanism for receiving incoming messages, IIapi\_wait() provides a way to yield the flow of control to OpenAPI. By occasionally calling IIapi\_wait(), an application ensures that it will receive incoming messages from the server.

IIapi\_wait() can be used in two programming styles: synchronous and asynchronous. To support the synchronous style of programming, the application can invoke an OpenAPI function and call IIapi\_wait() until the application notes the completion of the function (by checking gp\_completed) before calling the next function.

In an asynchronous application, the application code itself is event-driven (the events being OpenAPI function completions). Once the first function is invoked, the application calls IIapi\_wait() in a continuous top-level loop to drive all subsequent functions asynchronously.

This function has the following syntax:

```
II_VOID IIapi_wait (IIAPI_WAITPARAM *waitParm);

typedef struct _IIAPI_WAITPARAM
{
    II_LONG      wt_timeout;
    II_API_STATUS wt_status;
} IIAPI_WAITPARAM;
```

This function has the following parameters:

### **wt\_timeout**

**Type:** input

(Optional.) Indicates the maximum time in milliseconds to wait for the outstanding operations to complete. If timeout is not desired, this parameter is -1.

Support for timeouts is platform-dependent. If timeouts are not supported, all values are treated the same as -1.

### **wt\_status**

**Type:** output

Specifies the status of the function upon its return. Its value is one of the following:

- IIAPI\_SUCCESS
- IIAPI\_FAILURE

## IIapi\_xaCommit() Function—Commit an XA Transaction

The IIapi\_xaCommit() function commits an XA transaction started with IIapi\_xaStart(). The association between the transaction and a connection must have been dropped using IIapi\_xaEnd(). The transaction state of the connection used to commit the XA transaction is not affected. The connection handle does not need to be the same as was originally associated with the XA transaction when started by IIapi\_xaStart().

Generally, the XA transaction should have been prepared for two-phase commit using IIapi\_xaPrepare(). If the transaction has not been prepared for two-phase commit, the transaction flag IIAPI\_XA\_1PC must be specified to indicate a one-phase commit operation.

This function has the following syntax:

```
II_VOID IIapi_xaCommit( IIAPI_XACOMMITPARM *commitParm );
```

```
typedef struct _IIAPI_XACOMMITPARM
{
    IIAPI_GENPARM          xc_genParm;
    II_PTR                 xc_connHandle;
    IIAPI_TRAN_ID         xc_tranID;
    II_ULONG               xc_flags;
} IIAPI_XACOMMITPARM;
```

This function has the following parameters:

### **xc\_genParm**

**Type:** input and delayed output

Specifies the generic parameters (see page 41).

### **xc\_connHandle**

**Type:** input

Specifies the connection handle identifying the connection to be used to commit the XA transaction.

### **xc\_tranID**

**Type:** input

Specifies the XA transaction ID of the transaction which is to be committed.

### **xp\_flags**

**Type:** input

Specifies optional transaction flags. This parameter is 0 or a mask of the following values:

#### **IIAPI\_XA\_1PC**

Transaction has not been prepared (one-phase commit).

## **IIapi\_xaEnd() Function—End an XA Transaction Association**

The `IIapi_xaEnd()` function ends the association between a connection and an XA transaction. Before the XA transaction association can end, the application must call `IIapi_close()` to free the statement handles within the transaction. The XA transaction handle is freed and cannot be referenced further by the application.

The connection handle provided as input to this function must be associated with an XA transaction whose transaction ID matches the transaction ID provided as input.

Once the transaction association has been dropped, the XA transaction can be prepared for commit (two-phase) using `IIapi_xaPrepare()`, committed directly (one-phase) using `IIapi_xaCommit()`, or aborted using `IIapi_xaRollback()`.

This function has the following syntax:

```
II_VOID IIApi_xaEnd( IIAPI_XAENDPARAM *endParm );

typedef struct _IIAPI_XAENDPARAM
{
    IIAPI_GENPARAM    xe_genParm;
    II_PTR            xe_connHandle;
    IIAPI_TRAN_ID    xe_tranID;
    II_ULONG          xe_flags;
} IIAPI_XAENDPARAM;
```

This function has the following parameters:

**xe\_genParm**

**Type:** input and delayed output

Specifies the generic parameters (see page 41).

**xe\_connHandle**

**Type:** input

Specifies the connection handle identifying the connection associated with the XA transaction.

**xe\_tranID**

**Type:** input

Specifies the XA transaction ID of the transaction whose connection association is to end.

**xe\_flags**

**Type:** input

Specifies optional transaction flags. This parameter is 0 or a mask of the following values:

**IIAPI\_XA\_FAIL**

Transaction has failed (may only be rolled back).

## IIapi\_xaPrepare() Function—Prepare an XA transaction for Two-Phase Commit

The IIapi\_xaPrepare() function prepares to commit an XA transaction started with IIapi\_xaStart(). The association between the transaction and a connection must have been dropped using IIapi\_xaEnd(). The transaction state of the connection used to prepare the XA transaction is not affected. The connection handle does not need to be the same as was originally associated with the XA transaction when started by IIapi\_xaStart().

IIapi\_xaPrepare() secures resources for a transaction in a two-phase commit situation. When IIapi\_xaPrepare() completes successfully, the server has allocated and secured all resources to commit the transaction. The application can then call IIapi\_xaCommit() to commit the transaction or IIapi\_xaRollback() to abort the transaction.

This function has the following syntax:

```
II_VOID IIapi_xaPrepare( IIAPI_XAPREPPARM *prepParm );
```

```
typedef struct _IIAPI_XAPREPPARM
{
    IIAPI_GENPARAM      xp_genParm;
    II_PTR              xp_connHandle;
    IIAPI_TRAN_ID      xp_tranID;
    II_ULONG            xp_flags;
} IIAPI_XAPREPPARM;
```

This function has the following parameters:

### **xp\_genParm**

**Type:** input and delayed output

Specifies the generic parameters (see page 41).

### **xp\_connHandle**

**Type:** input

Specifies the connection handle identifying the connection to be used to prepare the XA transaction.

### **xp\_tranID**

**Type:** input

Specifies the XA transaction ID of the transaction which is to be prepared.

### **xp\_flags**

**Type:** input

Specifies optional transaction flags. Currently, no additional flags are defined. This parameter should be set to 0.

## IIapi\_xaRollback() Function—Rollback an XA Transaction

The IIapi\_xaRollback() function aborts an XA transaction started with IIapi\_xaStart(). The connection handle does not need to be the same as was originally associated with the XA transaction when started by IIapi\_xaStart().

If the XA transaction is associated with a connection (IIapi\_xaEnd() has not been used to drop the association), IIapi\_rollback() will need to be used to free the resources of the active transaction handle. Otherwise, the transaction state of the connection used to commit the XA transaction is not affected.

This function has the following syntax:

```
II_VOID IIapi_xaRollback( IIAPI_XAROLLPARAM *rollbackParm );

typedef struct _IIAPI_XAROLLPARAM
{
    IIAPI_GENPARAM      xr_genParm;
    II_PTR              xr_connHandle;
    IIAPI_TRAN_ID      xr_tranID;
    II_ULONG            xr_flags;
} IIAPI_XAROLLPARAM;
```

This function has the following parameters:

### **xr\_genParm**

**Type:** input and delayed output

Specifies the generic parameters (see page 41).

### **xr\_connHandle**

**Type:** input

Specifies the connection handle identifying the connection to be used to prepare the XA transaction.

### **xr\_tranID**

**Type:** input

Specifies the XA transaction ID of the transaction which is to be rolled back.

### **xr\_flags**

**Type:** input

Specifies optional transaction flags. Currently, no additional flags are defined. This parameter should be set to 0.

## IIapi\_xaStart() Function—Start an XA Transaction

The IIapi\_xaStart() function starts an XA transaction on a connection. An XA transaction handle is returned, which can be used to execute queries using IIapi\_query() or IIapi\_batch() within the context of the XA transaction.

The XA transaction handle allocated by this function should be released using IIapi\_xaEnd(). The transaction handle can also be released using IIapi\_commit() or IIapi\_rollback() if conditions do not require full distributed XA processing.

This function has the following syntax:

```
II_VOID IIapi_xaStart( IIAPI_XASTARTPARAM *startParm );

typedef struct _IIAPI_XASTARTPARAM
{
    IIAPI_GENPARAM      xs_genParm;
    II_PTR              xs_connHandle;
    IIAPI_TRAN_ID      xs_tranID;
    II_ULONG            xs_flags;
    II_PTR              xs_tranHandle;
} IIAPI_XASTARTPARAM;
```

This function has the following parameters:

### **xs\_genParm**

**Type:** input and delayed output

Specifies the generic parameters (see page 41).

### **xs\_connHandle**

**Type:** input

Specifies the connection handle identifying the connection associated with the XA transaction.

### **xs\_tranID**

**Type:** input

Specifies the XA transaction ID to be associated with the XA transaction.

### **xs\_flags**

**Type:** input

Specifies optional transaction flags. This parameter is 0 or a mask of the following values:

#### **IIAPI\_XA\_JOIN**

Joins an existing XA transaction.

**xs\_tranHandle**

**Type:** immediate output

Specifies the transaction handle to be used during the XA transaction.



# Chapter 3: OpenAPI Data Types

---

This section contains the following topics:

[OpenAPI Generic Data Types](#) (see page 133)

[OpenAPI Data Types](#) (see page 134)

[OpenAPI Data Structures](#) (see page 137)

This chapter describes the data structures and data types that are used in OpenAPI parameter structures.

## OpenAPI Generic Data Types

The OpenAPI data types are used to define other OpenAPI data types and data structures. The definition of these data types is platform-dependent.

**Note:** For the platform-specific definitions of these data types, see the OpenAPI header files `iiapi.h` and `iiapidep.h`.

The following are the OpenAPI generic data types:

### **II\_BOOL**

Describes a boolean data type (TRUE or FALSE).

### **II\_CHAR**

Describes a character data type.

### **II\_FLOAT4**

Describes a float4 (4-byte) data type.

### **II\_FLOAT8**

Describes a float8 (8-byte) data type.

### **II\_INT**

Describes an integer data type.

### **II\_INT1**

Describes an integer1 (1-byte) data type.

### **II\_INT2**

Describes an integer2 (2-byte) data type.

**II\_INT4**

Describes an integer4 (4-byte) data type.

**II\_LONG**

Describes a long integer data type.

**II\_PTR**

Describes a generic pointer data type.

**II\_UCHAR**

Describes an unsigned character data type.

**II\_UINT1**

Describes an unsigned integer1 (1-byte) data type.

**II\_UINT2**

Describes an unsigned integer2 (2-byte) data type.

**II\_UINT4**

Describes an unsigned integer4 (4-byte) data type.

**II\_ULONG**

Describes an unsigned long integer data type.

**II\_VOID**

Describes data of an unknown data type.

## OpenAPI Data Types

The following are OpenAPI data types:

- IIAPI\_DT\_ID data type
- IIAPI\_QUERYTYPE data type
- IIAPI\_STATUS data type

## IIAPI\_DT\_ID Data Type—Describe Data Type of Database Columns and Query Parameters

This data type has the following syntax:

```
typedef II_INT2 IIAPI_DT_ID;
```

The value of the IIAPI\_DT\_ID data type can be any one of the following:

- IIAPI\_BYTE\_TYPE
- IIAPI\_CHA\_TYPE
- IIAPI\_CHR\_TYPE
- IIAPI\_DEC\_TYPE
- IIAPI\_DTE\_TYPE
- IIAPI\_FLT\_TYPE
- IIAPI\_HNDL\_TYPE
- IIAPI\_INT\_TYPE
- IIAPI\_LOGKEY\_TYPE
- IIAPI\_LBYTE\_TYPE
- IIAPI\_LTXT\_TYPE
- IIAPI\_LVCH\_TYPE
- IIAPI\_MNY\_TYPE
- IIAPI\_TABKEY\_TYPE
- IIAPI\_TXT\_TYPE
- IIAPI\_VBYTE\_TYPE
- IIAPI\_VCH\_TYPE
- IIAPI\_NCHA\_TYPE
- IIAPI\_NVCH\_TYPE
- IIAPI\_LNVCH\_TYPE
- IIAPI\_DATE\_TYPE
- IIAPI\_TIME\_TYPE
- IIAPI\_TMWO\_TYPE
- IIAPI\_TMTZ\_TYPE
- IIAPI\_TS\_TYPE
- IIAPI\_TSWO\_TYPE
- IIAPI\_TSTZ\_TYPE

- IIAPI\_INTYM\_TYPE
- IIAPI\_INTDS\_TYPE
- IIAPI\_LCLOC\_TYPE
- IIAPI\_LBLOC\_TYPE
- IIAPI\_LNLOC\_TYPE

**More Information**

Ingres Data Types (see page 201).

## IIAPI\_QUERYTYPE Data Type—Describe Type of Query Being Invoked

This data type has the following syntax:

```
typedef II_ULONG IIAPI_QUERYTYPE;
```

The value of the IIAPI\_QUERYTYPE data type can be any one of the following:

- IIAPI\_QT\_QUERY
- IIAPI\_QT\_SELECT\_SINGLETON
- IIAPI\_QT\_EXEC
- IIAPI\_QT\_OPEN
- IIAPI\_QT\_CURSOR\_DELETE
- IIAPI\_QT\_CURSOR\_UPDATE
- IIAPI\_QT\_DEF\_REPEAT\_QUERY
- IIAPI\_QT\_EXEC\_REPEAT\_QUERY
- IIAPI\_QT\_EXEC\_PROCEDURE

## IIAPI\_STATUS Data Type—Describe the Return Status of an OpenAPI Function

This data type has the following syntax:

```
typedef II_ULONG IIAPI_STATUS;
```

The value of the IIAPI\_STATUS data type can be any one of the following:

- IIAPI\_ST\_SUCCESS
- IIAPI\_ST\_MESSAGE
- IIAPI\_ST\_WARNING
- IIAPI\_ST\_ERROR
- IIAPI\_ST\_NO\_DATA
- IIAPI\_ST\_FAILURE
- IIAPI\_ST\_NOT\_INITIALIZED
- IIAPI\_ST\_INVALID\_HANDLE
- IIAPI\_ST\_OUT\_OF\_MEMORY

## OpenAPI Data Structures

The OpenAPI data structures are described in the following sections.

## IIAPI\_COPYMAP Data Type—Provide Information on How to Execute the SQL Copy Statement

The IIAPI\_COPYMAP data type provides information needed to execute the copy statement, including the copy file name, log file name, number of columns in a row, and a description of the data.

This data type has the following syntax:

```
typedef struct _IIAPI_COPYMAP
{
    II_BOOL                cp_copyFrom;
    II_ULONG               cp_flags;
    II_LONG                cp_errorCount;
    II_CHAR II_FAR        *cp_fileName;
    II_CHAR II_FAR        *cp_logName;
    II_INT2                cp_dbmsCount;
    IIAPI_DESCRIPTOR II_FAR *cp_dbmsDescr;
    II_INT2                cp_fileCount;
    IIAPI_FDATADESCR II_FAR *cp_fileDescr;
} IIAPI_COPYMAP;
```

This data type has the following parameters:

### **cp\_copyFrom**

Indicates what kind of copy operation is being used. TRUE if the query is COPY FROM; FALSE if the query is COPY INTO.

### **cp\_dbmsCount**

Specifies the number of columns in a row in the database table.

### **cp\_dbmsDescr**

Specifies an array of data describing the database table column. The number of entries in this array is cp\_dbmsCount.

The memory for this parameter is managed by the OpenAPI.

**Note:** For more information, see Output Parameters Allocated by OpenAPI (see page 43).

### **cp\_errorCount**

Specifies the maximum number of errors allowed to occur before the copy statement is aborted.

**cp\_fileCount**

Specifies the number of data items in the copy file.

**cp\_fileDesc**

Specifies an array of data describing the copy file data items. The number of entries in this array is cp\_fileCount.

The memory for this parameter is managed by the OpenAPI.

**Note:** For more information, see Output Parameters Allocated by OpenAPI (see page 43) .

**cp\_fileName**

Specifies a NULL-terminating string containing the name of the copy file. This parameter cannot be NULL.

The memory for this parameter is managed by the OpenAPI.

**Note:** For more information, see Output Parameters Allocated by OpenAPI (see page 43).

**cp\_flags**

Specifies flag bits for the copy statement. Currently, there are no flag bits defined. (This parameter is reserved for future use.)

**cp\_logName**

Specifies a NULL-terminating string containing the name of the log file that logs all errors occurring during the copy. This parameter is NULL if logging of copy errors is not desired.

**Note:** For more information, see Output Parameters Allocated by OpenAPI (see page 43).

The memory for this parameter is managed by the OpenAPI.

## IIAPI\_DATAVALUE Data Type—Provide Value for OpenAPI Data

The IIAPI\_DATAVALUE data type contains the value for OpenAPI data. If the value is unavailable, this data type contains a NULL value and the `dv_null` parameter is TRUE.

This data type has the following syntax:

```
typedef struct _IIAPI_DATAVALUE
{
    IIAPI_BOOL      dv_null;
    IIAPI_UINT2     dv_length;
    IIAPI_PTR       dv_value;
} IIAPI_DATAVALUE;
```

This data type has the following parameters:

### **dv\_null**

Indicates whether the data is NULL. If it is NULL, this parameter is TRUE; otherwise, it is FALSE.

### **dv\_length**

Specifies the length of the data.

### **dv\_value**

Specifies the value of the data. When used to provide input to an OpenAPI function, this parameter must contain `dv_length` bytes of data. When used to receive output from an OpenAPI function, this parameter must be large enough to hold the datatype described by the corresponding descriptor (`ds_length` from IIAPI\_DESCRIPTOR).

## IIAPI\_DESCRIPTOR Data Type—Provide Description for OpenAPI Data

The IIAPI\_DESCRIPTOR data type describes OpenAPI data, including its type, length, precision, scale, and usage. This data is normally stored in an array of II\_DATAVALUE data type values.

This data type has the following syntax:

```
typedef struct _IIAPI_DESCRIPTOR
{
    IIAPI_DT_ID    ds_dataType;
    II_BOOL        ds_nullable;
    II_UINT2       ds_length;
    II_INT2        ds_precision;
    II_INT2        ds_scale;
    II_INT2        ds_columnType;
    II_CHAR        *ds_columnName;
} IIAPI_DESCRIPTOR;
```

This data type has the following parameters:

### **ds\_dataType**

Specifies the data type of the value being described.

### **ds\_nullable**

Indicates whether the data is nullable. If so, this parameter is TRUE; otherwise, it is FALSE.

### **ds\_length**

Specifies the length of the value being described.

### **ds\_precision**

Specifies the precision of the value being described. It is valid only when the ds\_dataType is IIAPI\_DEC\_TYPE or IIAPI\_FLT\_TYPE.

### **ds\_scale**

Specifies the scale of the value being described. It is valid only when the ds\_dataType parameter is IIAPI\_DEC\_TYPE.

### **ds\_columnType**

Specifies the usage of the value being described. Its value can be one of the following:

- IIAPI\_COL\_TUPLE
- IIAPI\_COL\_PROCBYREFPARAM
- IIAPI\_COL\_PROCPARM
- IIAPI\_COL\_SVCPARM
- IIAPI\_COL\_QPARAM
- IIAPI\_COL\_PROCPINPARAM

**Note:** IIAPI\_COL\_PROCINPARAM is a synonym for IIAPI\_COL\_PROCPARM.

- IIAPI\_COL\_PROCOUTPARAM
- IIAPI\_COL\_PROCINOUTPARAM

**Note:** IIAPI\_COL\_PROCINOUTPARAM is a synonym for IIAPI\_COL\_PROCBYREFPARAM.

For a description of how IIApi\_query() uses column types to describe its parameters, see Query Data Correlation (see page 162). Except for the information there, copy and tuple data always have the column type of IIAPI\_DS\_TUPLE.

**ds\_columnName**

Specifies the symbolic name of the OpenAPI data. It is used only when the data is copy file data, or a procedure parameter.

## IIAPI\_FDATABDESCR Data Type—Describe Column Data in a Copy File

This data type describes the data in a copy file. It also describes how the file should be formatted.

This data type has the following syntax:

```
typedef struct _IIAPI_FDATABDESCR
{
    II_CHAR          *fd_name;
    II_INT2          fd_type;
    II_INT2          fd_length;
    II_INT2          fd_prec;
    II_LONG          fd_column;
    II_LONG          fd_funcID;
    II_LONG          fd_cvLen;
    II_LONG          fd_cvPrec;
    II_BOOL          fd_delimiter;
    II_INT2          fd_delimLength;
    II_CHAR          *fd_delimValue;
    II_BOOL          fd_nullable;
    II_BOOL          fd_nullInfo;
    IIAPI_DESCRIPTOR fd_nullDescr;
    IIAPI_DATAVALUE  fd_nullValue;
} IIAPI_FDATABDESCR;
```

This data type has the following parameters:

### **fd\_name**

Specifies the name of the column in the file.

### **fd\_type**

Specifies the datatype of the column as stored in the file.

### **fd\_length**

Specifies the length of the column as stored in the file.

### **fd\_prec**

Specifies the encoded precision and scale of the column as stored in the file.

### **fd\_column**

Specifies the index in cp\_dbmsDescr of the corresponding column in the table (see IIAPI\_COPYMAP).

### **fd\_funcID**

Specifies an Ingres Abstract Data Facility function ID used for conversion between the table column datatype and file datatype.

### **fd\_cvLen**

Specifies the length used during conversion.

**fd\_cvPrec**

Specifies the encoded precision and scale used during conversion.

**fd\_delimiter**

Indicates whether a separator is used between columns. This parameter is TRUE if there is a separator between this and the next column; otherwise, it is FALSE.

**fd\_delimLength**

Specifies the length of the separator between this and the next column. This parameter is valid only if fd\_isDelimiter is TRUE.

**fd\_delimValue**

Specifies the value of the separator between this and the next column. This parameter is valid only if fd\_isDelimiter is TRUE.

**fd\_nullable**

Indicates whether a NULL value is allowed in the column. This parameter is TRUE if NULL is allowed; otherwise, it is FALSE.

**fd\_nullInfo**

Indicates whether a symbol representing a NULL value should be specified for this column. If so, this parameter is TRUE; otherwise, it is FALSE. This parameter is valid only if fd\_nullable is TRUE.

**fd\_nullDescr**

Specifies the description of the NULL symbol. This parameter is valid if fd\_nullable and fd\_nullInfo are TRUE.

**fd\_nullValue**

Specifies the value of the NULL symbol. This parameter is valid if fd\_nullable and fd\_nullInfo are TRUE.

## IIAPI\_II\_DIS\_TRAN\_ID Data Type—Identify Distributed Ingres Transaction ID

The IIAPI\_II\_DIS\_TRAN\_ID data type specifies and names a distributed Ingres transaction.

This data type has the following syntax:

```
typedef struct _IIAPI_II_DIS_TRAN_ID
{
    IIAPI_II_TRAN_ID    ii_tranID;
    II_CHAR             ii_tranName[IIAPI_TRAN_MAXNAME];
} IIAPI_II_DIS_TRAN_ID;
```

This data type has the following parameters:

### **ii\_tranID**

Specifies the unique transaction ID of the distributed transaction. This ID is an 8-byte ID stored in the IIAPI\_II\_TRAN\_ID data type.

### **ii\_tranName**

Specifies the unique transaction name of the Ingres transaction. It is a NULL-terminated string representing the transaction. The name must not be more than 63 bytes, leaving one byte for NULL.

## IIAPI\_II\_TRAN\_ID Data Type—Identify Local Ingres Transaction ID

The IIAPI\_II\_TRAN\_ID data type specifies a local Ingres transaction ID.

This data type has the following syntax:

```
typedef struct _IIAPI_II_TRAN_ID
{
    II_UINT4           it_highTran;
    II_UINT4           it_lowTran;
} IIAPI_II_TRAN_ID;
```

This data type has the following parameters:

### **it\_highTran**

Specifies the high order bytes of the transaction ID.

### **it\_lowTran**

Specifies the low order bytes of the transaction ID.

## IIAPI\_SVR\_ERRINFO Data Type—Describe Additional Server Information Associated with Error Messages

The IIAPI\_SVR\_ERRINFO data type contains additional server-specific information that was received along with an error, warning, or user message.

This data type has the following syntax:

```
typedef struct _IIAPI_SVR_ERRINFO
{
    II_LONG          svr_id_error;
    II_LONG          svr_local_error;
    II_LONG          svr_id_server;
    II_LONG          svr_server_type;
    II_LONG          svr_severity;
    II_INT2          svr_parmCount;
    II_API_DESCRIPTOR *svr_parmDescr;
    II_API_DATAVALUE *svr_parmValue;
} IIAPI_SVR_ERRINFO;
```

This data type has the following parameters:

### **svr\_id\_error**

Specifies the generic error code or encoded SQLSTATE of the message.

### **svr\_local\_error**

Specifies the server specific error code.

### **svr\_id\_server**

Specifies the ID of the server.

### **svr\_server\_type**

Specifies the type of the server.

### **svr\_severity**

Specifies the message type. Valid values can be a combination of the following:

- IIAPI\_SVR\_DEFAULT
- IIAPI\_SVR\_MESSAGE
- IIAPI\_SVR\_WARNING
- IIAPI\_SVR\_FORMATTED

### **svr\_parmCount**

Specifies the number of message parameters.

### **svr\_parmDescr**

Specifies the description of the message parameters. The memory for this parameter is managed by the OpenAPI.

**Note:** For more information, see Output Parameters Allocated by OpenAPI (see page 43).

#### **svr\_parmValue**

Specifies the value of the message parameters. The memory for this parameter is managed by the OpenAPI.

**Note:** For more information, see Output Parameters Allocated by OpenAPI (see page 43) .

## **IIAPI\_TRAN\_ID Data Type—Identify an OpenAPI Transaction**

The IIAPI\_TRAN\_ID data type specifies and names an OpenAPI transaction. OpenAPI supports Ingres transaction management. The union in this data type will increase as more transaction managers are supported.

This data type has the following syntax:

```
typedef struct _IIAPI_TRAN_ID
{
    II_ULONG                               ti_type;
    union
    {
        IIAPI_II_DIS_TRAN_ID             iiXID;
        IIAPI_XA_DIS_TRAN_ID             xaXID;
    } ti_value;
} IIAPI_TRAN_ID;
```

This data type has the following parameters:

#### **ti\_type**

Specifies the type of transaction being identified. The valid values are:

- IIAPI\_TI\_IIXID
- IIAPI\_TI\_XAXID

#### **ti\_value.iiXID**

Specifies the unique ID of an Ingres transaction. This parameter is valid only if ti\_type is IIAPI\_TI\_IIXID.

#### **ti\_value.xaXID**

Specifies the unique ID of an XA transaction. This parameter is valid only if ti\_type is IIAPI\_TI\_XAXID.

## IIAPI\_XA\_DIS\_TRAN\_ID Data Type—Identify a Distributed XA Transaction ID

The IIAPI\_XA\_DIS\_TRAN\_ID datatype specifies a distributed XA transaction ID.

This data type has the following syntax:

```
typedef struct _IIAPI_XA_DIS_TRAN_ID
{
    IIAPI_XA_TRAN_ID    xa_tranID;
    II_INT4             xa_branchSeqnum;
    II_INT4             xa_branchFlag;
} IIAPI_XA_DIS_TRAN_ID;
```

This data type has the following parameters:

### **xa\_tranID**

Specifies the unique transaction ID of an XA transaction.

### **xa\_branchSeqnum**

Specifies the branch sequence number.

### **xa\_branchFlag**

Specifies branch flags. A combination of one of the following flags:

- IIAPI\_XA\_BRANCH\_FLAG\_NOOP
- IIAPI\_XA\_BRANCH\_FLAG\_FIRST
- IIAPI\_XA\_BRANCH\_FLAG\_LAST
- IIAPI\_XA\_BRANCH\_FLAG\_2PC
- IIAPI\_XA\_BRANCH\_FLAG\_1PC

## IIAPI\_XA\_TRAN\_ID Data Type—Identify an XA Transaction ID

The IIAPI\_XA\_TRAN\_ID datatype specifies an XA transaction ID.

This data type has the following syntax:

```
typedef struct _IIAPI_XA_TRAN_ID
{
    II_LONG      xt_formatID;
    II_LONG      xt_gtridLength;
    II_LONG      xt_bqualLength;
    II_CHAR      xt_data[128];
} IIAPI_XA_TRAN_ID;
```

This data type has the following parameters:

### **xt\_formatID**

Specifies the format ID set by TP monitor.

### **xt\_gtridLength**

Specifies the length of gtrid value in xt\_data (first data item).

### **xt\_bqualLength**

Specifies the length of bqual value in xt\_data (second data item).

### **xt\_data**

Specifies the concatenated gtrid and bqual data values. Length is xt\_gtridLength + xt\_bqualLength.



# Chapter 4: Accessing a DBMS Using SQL

This section contains the following topics:

[Mapping of SQL to OpenAPI](#) (see page 151)

[SQL Syntax](#) (see page 158)

[Queries, Parameters, and Query Data Correlation](#) (see page 160)

## Mapping of SQL to OpenAPI

Most SQL statements are invoked by calling `IIapi_query()` with a query type of `IIAPI_QT_QUERY`, `iiapi_setDescriptor()`, and `Iapi_putParms()` for query parameters (optional), followed by `IIAPI_getQueryInfo()` and `IIapi_close()`. Some SQL statements require additional OpenAPI functions to provide query parameters and retrieve result data. As a common practice, an application should call `IIapi_getQueryInfo()` after each query is completed to check if the data source reported any errors.

**Note:** For more information on issuing SQL statement parameters, see [Queries, Parameters, and Query Data Correlation](#) (see page 160). For the syntax of SQL statements, see [SQL Syntax](#) (see page 158) and the *SQL Reference Guide*.

The following table shows the general SQL statement, as well as those SQL statements that are invoked differently:

SQL Statement	OpenAPI Function	Description	Query Type
<i>general SQL</i>	<code>IIapi_query()</code>	Invokes the SQL statement	<code>IIAPI_QT_QUERY</code>
	<code>[IIapi_setDescriptor()]</code>	Optional. Sends the information about the format of data to be sent with subsequent <code>IIapi_putParms()</code> calls	
	<code>[IIapi_putParms()]</code>	Optional. Sends data value for SQL statement parameters	
	<code>IIapi_getQueryInfo()</code>	Retrieves query results	
	<code>IIapi_close()</code>	Frees resources associated with the SQL statement	
<i>close</i>	<code>IIapi_close()</code>	Closes a cursor along with the statement handle	
<i>commit</i>	<code>IIapi_commit()</code>	Commits a transaction	

SQL Statement	OpenAPI Function	Description	Query Type
connect	IIapi_connect()	Connects to a data source	
copy from	IIapi_query()	Copies data from a file into a table	IIAPI_QT_QUERY
	IIapi_getCopyMap()	Retrieves copy data descriptors	
	IIapi_putColumns()	Submits copy data for the copy from statement. This function is called repeatedly until all data is sent.	
	IIapi_getQueryInfo()	Retrieves query results	
	IIapi_close()	Frees resources associated with the SQL statement	
copy into	IIapi_query()	Copies data from a table into a file	IIAPI_QT_QUERY
	IIapi_getCopyMap()	Retrieves copy data descriptors	
	IIapi_getColumns()	Retrieves copy data for the copy into statement. This function is called repeatedly until all data is retrieved.	
	IIapi_getQueryInfo()	Retrieves query results	
	IIapi_close()	Frees resources associated with the SQL statement	
delete (cursor)	IIapi_query()	Deletes rows from a table using a cursor	IIAPI_QT_CURSOR_DELETE
	IIapi_setDescriptor()	Sends information about the format of data to be sent with subsequent IIapi_putParms() calls	
	IIapi_putParms()	Sends data values for SQL statement parameters	
	IIapi_getQueryInfo()	Retrieves the row count	
	IIapi_close()	Frees resources associated with the SQL statement	
delete (repeat)	IIapi_query()	Defines a repeated delete	First repeat: IIAPI_QT_DEF_REPEAT_QUERY
	IIapi_setDescriptor()	Sends information about the format of data to be sent with	

SQL Statement	OpenAPI Function	Description	Query Type
		subsequent IApi_putParms() calls	
	IApi_putParms()	Sends data values for SQL statement parameters	
	IApi_getQueryInfo()	Retrieves a repeat query ID	
	IApi_close()	Frees resources associated with the SQL statement	
	IApi_query()	Executes a repeated delete	Subsequent repeats: IIAPI_QT_EXEC_REPEAT_QUERY
	IApi_setDescriptor()	Sends information about the format of data to be sent with subsequent IApi_putParms()	
	IApi_putParms()	Sends data values for SQL statement parameters	
	IApi_getQueryInfo()	Retrieves row count and query results	
	IApi_close()	Frees resources associated with the SQL statement	
describe	IApi_query()	Describes a previously prepared statement	IIAPI_QT_QUERY
	IApi_getDescriptor()	Retrieves descriptors if a prepared statement is a select statement	
	IApi_getQueryInfo()	Retrieves query results	
	IApi_close()	Frees resources associated with the SQL statement	
disconnect	IApi_disconnect()	Disconnects from a data source	
execute	IApi_query()	Executes a previously-prepared statement	IIAPI_QT_EXEC
	[IApi_setDescriptor()]	Optional. Sends information about the format of data to be sent with subsequent IApi_putParms() calls	
	[IApi_putParms()]	Optional. Sends data values for SQL statement parameters	
	IApi_getQueryInfo()	Retrieves query results	

SQL Statement	OpenAPI Function	Description	Query Type
	IIapi_close()	Frees resources associated with the SQL statement	
execute procedure	IIapi_query()	Invokes a database procedure	IIAPI_QT_EXEC_PROCEDURE
	IIapi_setDescriptor()	Sends information about the format of data to be sent with subsequent IIapi_putParms()	
	IIapi_putParms()	Sends data values for SQL statement parameters	
	[IIapi_getDescriptor]	Retrieves descriptors of output parameters or columns of row-returning procedures	
	[IIapi_getColumns]	Retrieves output parameter values or column values of row-returning procedures	
	IIapi_getQueryInfo()	Retrieves query results	
	IIapi_close()	Frees resources associated with the SQL statement	
fetch	IIapi_getColumns()	Retrieves rows of data	
	[IIapi_getQueryInfo()]	Retrieves query results	
get dbevent	IIapi_catchEvent()	Retrieves a database event	
	[IIapi_getEvent()]	Wait for a database event to be received	
	IIapi_close()	Frees resources associated with the SQL statement	
insert (repeat)	IIapi_query()	Defines a repeated insert	First repeat: IIAPI_QT_DEF_REPEAT_QUERY
	IIapi_setDescriptor()	Sends information about the format of data to be sent with subsequent IIapi_putParms()	
	IIapi_putParms()	Sends data values for SQL statement parameters	
	IIapi_getQueryInfo()	Retrieves a repeat query ID	
	IIapi_close()	Frees resources associated with the SQL statement	

SQL Statement	OpenAPI Function	Description	Query Type
	IIapi_query()	Executes a repeated insert	Subsequent repeats: IIAPI_QT_EXEC_ REPEAT-QUERY
	IIapi_setDescriptor()	Sends information about the format of data to be sent with subsequent IIapi_putParms() calls	
	IIapi_putParms()	Sends data values for SQL statement parameters	
	IIapi_getQueryInfo()	Retrieves row count and query results	
	IIapi_close()	Frees resources associated with the SQL statement	
open	IIapi_query()	Opens a cursor	IIAPI_QT_OPEN
	[IIapi_setDescriptor()]	Sends information about the format of cursor to be sent with subsequent IIapi_putParms() calls	
	[IIapi_putParms()]	Sends data values for the cursor parameter	
	IIapi_getDescriptor()	Retrieves descriptors for data	
	[IIapi_getQueryInfo()]	Retrieves query results	
prepare	IIapi_query()	Prepares a statement to be executed later	IIAPI_QT_QUERY
	[IIapi_getDescriptor]	Retrieves descriptor when "into sqllda" is used in prepare query text	
	IIapi_getQueryInfo()	Retrieves query results	
	IIapi_close()	Frees resources associated with the SQL statement	
prepare to commit	IIapi_prepareCommit()	Polls the server to determine the commit status of a local transaction associated with a specified distributed transaction, and secures the transaction	
rollback	IIapi_rollback()	Rolls back a transaction to its beginning or to a savepoint	

SQL Statement	OpenAPI Function	Description	Query Type
savepoint	IIapi_savepoint()	Declares a savepoint marker within a transaction	
select	IIapi_query()	Retrieves values from one or more tables	IIAPI_QT_QUERY or IIAPI_QT_SELECT_SINGLETON
	IIapi_getDescriptor()	Retrieves descriptors for data	
	IIapi_getColumns()	Retrieves rows of data. This function is called repeatedly until all data is retrieved.	
	IIapi_getQueryInfo()	Retrieves query results	
	IIapi_close()	Frees resources associated with the SQL statement	
select (repeat)	IIapi_query()	Defines a repeated select	First repeat: IIAPI_QT_DEF_REPEAT_QUERY
	IIapi_setDescriptor()	Sends information about the format of data to be sent with subsequent IIapi_putParms() calls	
	IIapi_putParms()	Sends data values for SQL statement parameters	
	IIapi_getQueryInfo()	Retrieves repeat query ID	
	IIapi_close()	Frees resources associated with the SQL statement	
	IIapi_query()	Executes a repeated select	Subsequent repeats: IIAPI_QT_EXEC_REPEAT_QUERY
	IIapi_setDescriptor()	Sends information about the format of data to be sent with subsequent IIapi_putParms() calls	
	IIapi_putParms()	Sends data values for SQL statement parameters	
	IIapi_getDescriptor()	Retrieves descriptors for one row of data	
	IIapi_getColumns()	Retrieves rows of data. This	

SQL Statement	OpenAPI Function	Description	Query Type
		function is called repeatedly until all data is retrieved.	
	IIapi_getQueryInfo()	Retrieves query results	
	IIapi_close()	Frees resources associated with the SQL statement	
set autocommit on/off	IIapi_autocommit()	Enables or disables the autocommit state in the server	
update (cursor)	IIapi_query()	Updates column values in a table using a cursor	API_QT_CURSOR_UPDATE
	II_api_setDescriptor()	Sends information about the format of data to be sent with subsequent IIapi_putParms() calls	
	IIapi_putParms()	Sends data values for SQL statement parameters	
	IIapi_getQueryInfo()	Retrieves row count, logical key, and query results	
	IIapi_close()	Frees resources associated with the SQL statement	
update (repeat)	IIapi_query()	Defines a repeated update	First repeat: IIAPI_QT_DEF_REPEAT_QUERY
	IIapi_setDescriptor()	Sends information about the format of data to be sent with subsequent IIapi_putParms() calls	
	IIapi_putParms()	Sends data values for SQL statement parameters	
	IIapi_getQueryInfo()	Retrieves a repeated query ID	
	IIapi_close()	Frees resources associated with the SQL statement	
	IIapi_query()	Executes the repeated update	Subsequent repeats: IIAPI_QT_EXEC_REPEAT_QUERY
	IIapi_setDescriptor()	Sends information about the format of data to be sent with subsequent IIapi_putParms()	

SQL Statement	OpenAPI Function	Description	Query Type
		calls	
	IIfapi_putParms()	Sends data values for SQL statement parameters	
	IIfapi_getQueryInfo()	Retrieves row count, logical key, and query results	
	IIfapi_close()	Frees resources associated with the SQL statement	

## SQL Syntax

This section describes the statements for which the syntax required by OpenAPI **differs** from that of embedded SQL.

### Describe Statement

This statement has the following syntax in OpenAPI:

```
describe statement_name
```

Do not include an into or using clause.

### Execute Statement

This statement has the following syntax in OpenAPI:

```
execute statement_name
```

Do not include a using clause.

## Declare Statement, Open Cursor Statement

The declare cursor statement is not supported; the information associated with declare cursor is provided when the cursor is opened.

This statement has the following syntax in OpenAPI:

```
statement_name | select_statement [for readonly]
```

The open cursor syntax consists of the statement name or select statement text which in embedded SQL would be provided in the declare cursor statement, along with an optional for readonly clause. Do not include a using clause with the open cursor query. The application can provide a cursor name as a parameter to the open cursor query. If the application does not provide a cursor name, a default name of the form IIAPICURSOR $n$  (where  $n$  is an integer) is used.

## Cursor Delete Statement

This statement has the following syntax in OpenAPI:

```
delete from tablename
```

Do not include a where clause. The application provides the cursor ID as a parameter to the cursor delete query. The cursor ID is the statement handle returned when the cursor is opened. OpenAPI builds the where clause using the information in the cursor ID.

## Cursor Update Statement

This statement has the following syntax in OpenAPI:

```
update tablename set_clause
```

Do not include a where clause. The application provides the cursor ID as a parameter to the cursor update query. The cursor ID is the statement handle returned when the cursor is opened. OpenAPI builds the where clause using the information in the cursor ID.

## Execute Procedure

OpenAPI does not require any query text to execute a database procedure. The application provides the database procedure name, procedure owner, and procedure parameters as parameters to the query and sets the query type to `IIAPI_QT_EXEC_PROCEDURE`. A database procedure can be executed as a regular query with query type `IIAPI_QT_QUERY` using the non-dynamic syntax, but procedure return values and output parameters are not available.

## Repeat Queries

OpenAPI does not support the repeated keyword in delete, insert, select, and update statements. Repeat queries are supported in OpenAPI using the same underlying mechanism as used by embedded SQL. While embedded SQL hides the repeat query mechanism from the application, OpenAPI applications must manage this mechanism themselves.

Using Repeat Queries with OpenAPI (see page 185) describes how an OpenAPI application can use repeat queries.

## Queries, Parameters, and Query Data Correlation

Use this section as a guide for entering SQL statement parameters with `IIapi_query()`. It also provides valid column and data types for parameters when sending data formats with `IIapi_setDescriptor()`.

## Queries and Parameters

Each SQL statement invoked with `IIapi_query()` may or may not have query parameters associated with it. In general, query input parameters are needed when:

- An application specifies a dynamic parameter marker (?) in the query text
- A procedure ID, procedure name, procedure owner, or procedure parameters are required for the SQL statement
- A cursor ID or cursor name is required for the SQL statement
- A repeat query ID, repeat query parameters, or repeat query handle is required for a repeated statement

The table in this section lists the query types for SQL statements that can have query parameters associated with them. The first three columns provide information for sending an SQL statement to the DBMS Server. The Query Type column lists valid entries for the `qy_queryType` parameter of `IIapi_query()`. The Query Text? column indicates whether a query text entry is required in the `qy_queryText` parameter of `IIapi_query()`. Query text is not required for some SQL statements because OpenAPI can construct the query text based solely on the query type.

When SQL statements require query parameters, the application invokes `IIapi_query()` and then enters the parameters in subsequent calls to `IIapi_setDescriptor()` and `IIapi_putParms()`. The Query Input Parameters column indicates what parameters are needed.

**Note:** For a description of SQL statements and their parameters, see the *SQL Reference Guide*.

When an application calls `IIapi_getQueryInfo()`, it receives information about the status of a previously-invoked SQL statement. For some statements, `IIapi_getQueryInfo()` also returns available response data in the `gq_mask` parameter. This information is given for applicable SQL statements in the Result Parameters column of the table.

Query Type	Query Text?	Query Input Parameters	Result Parameters
API_QT_EXEC	Yes *	[dynamic parameter values]	
API_QT_OPEN	Yes *	[cursor name] [dynamic parameter values]	IIAPI_GQ_CURSOR
API_QT_CURSOR_DELETE	Yes *	Cursor ID	

Query Type	Query Text?	Query Input Parameters	Result Parameters
API_QT_CURSOR_UPDATE	Yes *	Cursor ID [parameter values]	IIAPI_GQ_TABLE_KEY or IIAPI_GQ_OBJECT_KEY
API_QT_DEF_REPEAT_QUERY	Yes *	[repeat query ID] [repeat query parameters]	IIAPI_GQ_REPEAT_QUERY_ID
API_QT_EXEC_REPEAT_QUERY	No	Repeat query handle [repeat query parameters]	
API_QT_EXEC_PROCEDURE	No	Procedure name or procedure ID [procedure owner] [procedure parameters]	IIAPI_GQ_PROCEDURE_ID  IIAPI_GQ_PROCEDURE_RET
API_QT_QUERY	Yes	[parameter values]	
API_QT_SELECT_SINGLETON	Yes	[parameter values]	

\* The query syntax for these statements differs from that used in embedded SQL. See SQL Syntax (see page 158).

## Query Data Correlation

When an application calls `IIapi_setDescriptor()` to send information to the DBMS Server about parameter formats, it should make sure that the `sd_descriptor` parameter contains the appropriate column type and data type for each query parameter of an SQL statement.

The following table provides query parameters and their corresponding `ds_columnType` and `ds_dataType` values in the `API_DESCRIPTOR` structure:

Query Parameters	Descriptor Column Type	Data Type
Cursor name	IIAPI_COL_SVCPARM	IIAPI_CHA_TYPE
Cursor ID	IIAPI_COL_SVCPARM	IIAPI_APIHNDL_TYPE
Procedure ID	IIAPI_COL_SVCPARM	IIAPI_HNDL_TYPE
Procedure name	IIAPI_COL_SVCPARM	IIAPI_CHA_TYPE
Procedure owner	IIAPI_COL_SVCPARM	IIAPI_CHA_TYPE
Procedure parameters	IIAPI_COL_PROCPARM or IIAPI_COL_PROCBYREFPARM	Any of the Ingres data types in Ingres Data Types the "Data

<b>Query Parameters</b>	<b>Descriptor Column Type</b>	<b>Data Type</b>
	IIAPI_COL_PROCPINPARAM IIAPI_COL_PROPCOUTPARAM IIAPI_COL_PROCPINOUTPARAM	Types" appendix
Parameter values	IIAPI_COL_QPARAM	Any of the Ingres data types in Ingres Data Types the "Data Types" appendix
Repeat query ID	IIAPI_COL_SVCPARM IIAPI_COL_SVCPARM IIAPI_COL_SVCPARM	IIAPI_INT_TYPE IIAPI_INT_TYPE IIAPI_CHA_TYPE
Repeat query handle	IIAPI_COL_SVCPARM	IIAPI_APIHNDL_TYPE
Repeat query parameters	IIAPI_COL_QPARAM	Any of the Ingres data types in Ingres Data Types the "Data Types" appendix
Dynamic parameter values	IIAPI_COL_QPARAM	Any of the Ingres data types in Ingres Data Types the "Data Types" appendix



# Chapter 5: Accessing the Name Server

---

This section contains the following topics:

[Mapping of Name Server Query Statements to OpenAPI](#) (see page 165)

[Name Server Query Statement Syntax](#) (see page 166)

[Name Server Query Syntax](#) (see page 167)

## Mapping of Name Server Query Statements to OpenAPI

The following table shows the mapping of Name Server query statements to OpenAPI functions:

Name Server Query Statement	OpenAPI Function	Description	Query Type
connect	IIapi_connect()	Connects to the Name Server	
	IIapi_autocommit()	(Required) Enables or disables autocommit transactions	
create, destroy	IIapi_query()		IIAPI_QT_QUERY
	[IIapi_setDescriptor()]		
	[IIapi_putParms()]		
	IIapi_getQueryInfo()		
	IIapi_close()		
show	IIapi_query()		IIAPI_QT_QUERY
	[IIapi_setDescriptor()]		
	[IIapi_putParms()]		
	Iiapi_getDescriptor()		
	IIapi_getColumns()		
	IIapi_getQueryInfo()		
	IIapi_close()		
disconnect	IIapi_autocommit()		
	IIapi_disconnect()		

## Name Server Query Statement Syntax

The Name Server query text is divided into fields, which are separated by a blank space. The first four fields of a query text statement describe the action to be performed and the virtual node name with which the action is associated. These four fields appear in the following order in each query statement:

Field	Parameter	Value	Description
1	Function	create, destroy, or show	The task that will be performed.
2	Type	global or private	The registration type of the object. A global object is available to all users on the local node. A private object is available to a single user. Private objects are created and destroyed for the currently logged-in user. Only a user with the NET_ADMIN GCA privilege (generally a system administrator) can create and destroy global objects.
3	Object	login, connection, or attribute	The object to be created, destroyed, or shown. Login is the remote user authorization definition (user name and password). Connection refers to a connection data definition. Attribute refers to the attribute of the virtual node.
4	Virtual node name	vnode name	The virtual node name (vnode) associated with the operation. Each query statement must contain a vnode identifier.

**Note:** A single-letter abbreviation is sufficient for the first three fields (function, type, and object). Virtual node names cannot be abbreviated. For query text statements that specify either the destroy or show function, you can enter the asterisk character (\*) as a wildcard in any field other than the function, type, and object fields. Wildcards cannot be used with the create function.

## Name Server Query Syntax

This section describes the OpenAPI syntax for the statements used to access the Name Server.

### Create Login Statement—Create a Login Definition

This statement creates a remote user authorization.

This statement has the following syntax:

```
create global|private login vnode username password
```

This statement has the following parameters:

**global|private**

Specifies a global or private object. A global object is available to all users on the local node. A private object is available to a single user.

***vnode***

Specifies the virtual node name to be associated with this authorization.

***username***

Specifies the name of the account to be used on the host machine of a remote installation.

***password***

Specifies the password of the remote account.

### Examples: Create login statement

The following example creates a private user authorization on vnode apitest for user Tom:

```
create private login apitest tom tompassword
```

The following example creates a global user authorization on vnode dbtest using an installation password:

```
create global login dbtest * installationpassword
```

## Destroy Login Statement—Destroy a Login Definition

This statement deletes a remote user authorization.

This statement has the following syntax:

```
destroy global|private login vnode
```

This statement has the following parameters:

### **global|private**

Specifies a global or private object. A global object is available to all users on the local node. A private object is available to a single user.

### ***vnode***

Specifies the virtual node name to be associated with this authorization.

### Examples: Destroy login statement

The following example destroys a private user authorization on vnode apitest:

```
destroy private login apitest
```

The following example destroys a private user authorization on all the vnodes where it occurs. Using a wildcard for the *vnode* parameter lets you destroy all instances of a particular login with a single query text statement:

```
destroy private login *
```

## Create Password Statement—Define an Installation Password

This statement creates an installation password for the local installation.

This statement has the following syntax:

```
create global login local_vnode * password
```

This statement has the following parameters:

### ***local\_vnode***

Specifies the name that has been configured as *local\_vnode* parameter on the local installation. You can find this name on the Parameters tab for the Name Server in the Configuration Manager utility. (Alternatively, you can use the *cbf* utility at the command line.)

### ***password***

Specifies the installation password to be assigned to this installation.

### Example: Create password statement

The following example defines an installation password for the local installation, which has a `local_vnode` name of `apitest`:

```
create global login apitest * apitest_password
```

## Create Connection Statements—Create a Connection Data Definition

This statement creates a connection data definition. If a connection data definition exists that matches the specified one in all respects, the operation will have no effect, and no error will be reported.

This statement has the following syntax:

```
create connection global|private vnode net_addr protocol port
```

This statement has the following parameters:

### **global|private**

Specifies a global or private object. A global object is available to all users on the local node. A private object is available to a single user.

### ***vnode***

Specifies the virtual node name to be associated with this connection data definition.

### ***net\_addr***

Specifies the address or name of the remote node.

Your network administrator specified this address or name when the network software was installed. Normally, the node name, as defined at the remote node, is sufficient for this parameter.

The format of a net address depends on the type of network software that the node is using.

### ***protocol***

Specifies the Ingres keyword for the protocol used to connect to the remote node. Its value can be one of the following:

- `tcp_ip`
- `wintcp`
- `lanman`
- `nvlspx`
- `decnet`

**port**

Specifies the unique identifier used by the remote Communications Server for interprocess communication.

The format of a listen address depends on the network protocol.

**Example: Create connection statement**

The following example creates a global connection data definition on vnode *apitest*, where *net\_addr=apitest*, *protocol=TCP/IP*, and *port=mg0*:

```
create global connection apitest apitest tcp_ip mg0
```

**Note:** The virtual node name and the host are different objects, although typically they have the same value.

**Destroy Connection Statement—Destroy a Data Definition**

This statement deletes a connection data definition.

This statement has the following syntax:

```
destroy global|private connection vnode net_addr protocol port
```

This statement has the following parameters:

**global|private**

Specifies a global or private object. A global object is available to all users on the local node. A private object is available to a single user.

**vnode**

Specifies the virtual node name to be associated with this connection data definition.

**net\_addr**

Specifies the address or name of the remote node.

Your network administrator specifies this address or name when the network software is installed. Normally, the node name, as defined at the remote node, is sufficient for this parameter.

The format of a net address depends on the type of network software that the node is using.

***protocol***

Specifies the Ingres keyword for the protocol used to connect to the remote node. Its value can be one of the following:

- tcp\_ip
- wintcp
- lanman
- nvlspix
- decnet

***port***

Specifies the unique identifier used by the remote Communications Server for interprocess communication.

The format of a listen address depends on the network protocol.

**Examples: Destroy connection statement**

The following example destroys a private connection data definition on vnode *apitest*, where *net\_addr*=*apitest*, *protocol*=TCP/IP, and *port*=*mg2*:

```
destroy private connection apitest apitest tcp_ip mg2
```

The following example destroys all global connection data definitions on vnode *payroll* that include the TCP/IP protocol:

```
destroy global connection payroll * tcp_ip *
```

## Show Connection Statement—Display Connection Data Definitions

This statement displays a connection data definition.

This statement has the following syntax:

```
show global|private connection vnode net_addr protocol port
```

This statement has the following parameters:

### **global|private**

Specifies a global or private object. A global object is available to all users on the local node. A private object is available to a single user.

### ***vnode***

Specifies the virtual node name to be associated with this connection data definition.

### ***net\_addr***

Specifies the address or name of the remote node.

Your network administrator specifies this address or name when the network software is installed. Normally, the node name, as defined at the remote node, is sufficient for this parameter.

The format of a net address depends on the type of network software that the node is using.

### ***protocol***

Specifies the Ingres keyword for the protocol used to connect to the remote node. Its value can be one of the following:

- tcp\_ip
- wintcp
- lanman
- nvlspix
- decnet

### ***port***

Specifies the unique identifier used by the remote Communications Server for interprocess communication.

The format of a listen address depends on the network protocol.

### Example: Show connection statement

The following example displays global connection data definitions on vnode apitest, where net\_addr is apitest:

```
show global connection apitest apitest* *
```

The following is an example of sample output from this operation:

```
global connection apitest apitest tcp_ip mg2
```

### Create Attribute Statement—Create an Attribute Data Definition

This statement creates an attribute data definition.

This statement has the following syntax:

```
create global|private attribute vnode attr_name attr_value
```

This statement has the following parameters:

**global|private**

Specifies a global or private object. A global object is available to all users on the local node. A private object is available to a single user.

**vnode**

Specifies the virtual node name to be associated with this attribute data definition.

**attr\_name**

Specifies the name of the attribute.

**attr\_value**

Specifies the value of the attribute.

### Example: Create attribute statement

The following example creates a global attribute definition on vnode apitest, where *attr\_name*=connection\_type and *attr\_value*=direct:

```
create global attribute apitest connection_type direct
```

## Destroy Attribute Statement—Destroy an Attribute Data Definition

This statement deletes an attribute data definition.

This statement has the following syntax:

```
destroy global|private attribute vnode attr_name attr_value
```

This statement has the following parameters:

**global|private**

Specifies a global or private object. A global object is available to all users on the local node. A private object is available to a single user.

***vnode***

Specifies the virtual node name to be associated with this attribute data definition.

***attr\_name***

Specifies the name of the attribute.

***attr\_value***

Specifies the value of the attribute.

### Examples: Destroy attribute statement

The following example destroys a private attribute definition on vnode apitest, where *attr\_name*=connection\_type and *attr\_value*=direct:

```
destroy private attribute connection_type direct
```

The following example destroys all global attribute data definitions for vnode payroll that include the connection\_type attribute:

```
destroy global attribute payroll connection_type *
```

## Display Attribute Statement—Display an Attribute Data Definition

This statement displays an attribute data definition.

This statement has the following syntax:

```
show global|private attribute vnode attr_name attr_value
```

This statement has the following parameters:

### **global|private**

Specifies a global or private object. A global object is available to all users on the local node. A private object is available to a single user.

### ***vnode***

Specifies the virtual node name to be associated with this attribute data definition.

### ***attr\_name***

Specifies the name of the attribute.

### ***attr\_value***

Specifies the value of the attribute.

## Examples: Display attribute statement

The following example displays all global attribute data definitions on vnode apitest:

```
show global attribute apitest * *
```

The following is an example of sample output from this operation:

```
global attribute apitest connection_type direct
```

## Show Server Statement—Display Servers in the Local Installation

This statement displays servers in the local installation.

This statement has the following syntax:

```
show server server_class
```

This statement has the following parameter:

### ***server\_class***

Specifies the name of the server class of the server to be displayed. To display all servers in the local installation, specify "servers."

### Examples: Show server statement

The following example displays all servers in the local installation:

```
show server servers
```

The following example displays an Ingres server in the local installation:

```
show server ingres
```

### How to Use ~V Marker in the Name Server Query Text

You can use a ~V marker in any of the subfields of the objects (such as login, connection, or attribute) and in the Type field in the Name Server query text. The ~V marker must be preceded and followed by a space character. Values for the parameter markers are sent with the query using `IIapi_setDescriptor()` and `IIapi_putParams()`, in the same order as the parameter markers appear in the query text.

### Example: Using the ~V marker in name server query text

The following query text is a valid query text with the ~V marker:

```
show server ~V
```

When sent with parameter value of "servers," it is identical to the query:

```
show server servers
```

The advantage is that the query using ~V does not require the application to build the query text at runtime using values not available at compile time.

# Chapter 6: Creating an Application with OpenAPI

---

This section contains the following topics:

[How You Can Create an OpenAPI Application](#) (see page 177)

[Sample Code](#) (see page 179)

## How You Can Create an OpenAPI Application

This section describes the header file, library, and environment variables used by an OpenAPI application.

**Note:** For more information, see the readme.

### Header Files

Each application source file that invokes an OpenAPI function must include the OpenAPI header file (iiapi.h). This header file includes a platform-dependent header file, iiapidep.h, which configures OpenAPI for a particular platform.

### Library

When using OpenAPI, an application must link with the OpenAPI library. Where applicable, a shared library can also be provided. Applications may also need to be linked with the standard Ingres runtime library.

### Environment Variables

The following user environment variables are used by OpenAPI:

- II\_API\_TRACE
- II\_API\_LOG
- II\_API\_SET

## II\_API\_TRACE

The II\_API\_TRACE environment variable specifies the desired trace level of OpenAPI. When it is not defined, it has a value of 0. You can set the II\_API\_TRACE environment variable to one of the following values:

Value	Action
1	Display fatal error messages
2	Display non-fatal error messages
3	Display warning messages
4	Display checkpoint messages, such as which OpenAPI function is being executed
5	Display detail information, such as values of input and output parameters

## II\_API\_LOG

The II\_API\_LOG environment variable specifies the desired output file for all OpenAPI tracing.

## II\_API\_SET

The II\_API\_SET environment variable allows GCA\_TRACE messages to be written to a file.

Set II\_API\_SET to PRINTTRACE to write GCA\_TRACE messages to default output file iiprttrc.c. To direct output to a different file, add the TRACEFILE option followed by the file name.

For example, on UNIX, to enable OpenAPI tracing of GCA\_TRACE messages to output file tracing.log:

```
setenv II_API_SET "printtrace; tracefile tracing.log"
```

## Sample Code

The Ingres installation includes OpenAPI sample code that demonstrates synchronous and asynchronous execution of OpenAPI functions. The sample code provides an example of how an application uses OpenAPI functions and combines them to access database information.

The sample code is located in the API directory, which is under the Ingres demo directory. Makefiles for compiling and linking the demo programs are provided for UNIX (Makefile) and Windows NT (makefile.wnt) environments.

### How the Synchronous Sample Code Works

The synchronous sample code consists of a number of source files whose names begin with the prefix "apis". Each source file demonstrates a single aspect of the OpenAPI interface and compiles into its own executable demo program.

OpenAPI functions are called synchronously: each OpenAPI function call is followed by a loop that calls `IIapi_wait()` repeatedly until the original OpenAPI call completes.

**Note:** Database event processing is inherently asynchronous and is not demonstrated in the synchronous sample code. The asynchronous sample code provides an extensive example of database event handling using OpenAPI.

### main() Function

The `main()` function at the beginning of the source file usually contains the specific aspect being demonstrated. OpenAPI function calls that support the demonstration, such as connection or transaction management, are at the end of the file. Sometimes, when the demonstration calls for multiple executions of an OpenAPI function, you can place the OpenAPI function call in its own function following `main()`.

## Error Checking

Typically, the sample programs do not perform error checking on the results of OpenAPI function calls. Error processing is demonstrated in one of the sample programs.

In general, these source files show:

- How OpenAPI function parameters are established
- The order in which OpenAPI functions are called to perform a specific action
- How OpenAPI function output is processed

While the samples can be compiled and executed, the actual runtime actions performed provide little useful information.

## How You Can Run a Program

Each sample program takes a single command line argument, which is the target database to run against. For example:

```
apisconn mydb
```

## Source File Descriptions

The following list identifies the synchronous sample source files and the OpenAPI functionality they demonstrate.

### **apisinit.c**

Demonstrates OpenAPI initialization, termination, and handling of environment handles.

### **apisdata.c**

Demonstrates data conversion using `IIapi_convertData()` and `IIapi_formatData()`.

### **apiserr.c**

Demonstrates processing OpenAPI status results and error handling.

### **apisconn.c**

Demonstrates establishing, terminating, and aborting connections.

### **apiscomm.c**

Demonstrates committing transactions.

**apisroll.c**

Demonstrates rolling back transactions, setting savepoints, and rolling back to a savepoint.

**apisauto.c**

Demonstrates starting and ending autocommit transactions.

**apis2ph1.c**

Demonstrates distributed transactions (part 1): register transaction ID and prepare to commit.

**apis2ph2.c**

Demonstrates distributed transactions (part 2): connect to transaction, rollback, release transaction ID.

**apisparm.c**

Demonstrates handling of query parameters.

**apisell.c**

Demonstrates retrieving data using a select loop and canceling a query.

**apiselc.c**

Demonstrates retrieving data using a cursor.

**apiscdel.c**

Demonstrates deleting rows using a cursor.

**apiscupd.c**

Demonstrates updating rows using a cursor.

**apisproc.c**

Demonstrates executing database procedures.

**apisprbr.c**

Demonstrates executing database procedures with BYREF parameters.

**apisprrp.c**

Demonstrates executing row-producing database procedures.

**apisprgt.c**

Demonstrates executing database procedures with global temporary table parameters.

**apiscopy.c**

Demonstrates using COPY TABLE statement to transfer data between application and DBMS.

**apisrept.c**

Demonstrates defining and executing repeat statements.

**apisblob.c**

Demonstrates processing BLOB parameters and results.

**apisname.c**

Demonstrates accessing Name Server VNODE information.

## How the Asynchronous Sample Code Works

The asynchronous sample code consists of a number of source files whose names begin with the prefix "apia". The source files are compiled and linked to produce a single comprehensive demo program that provides a real-life example of using OpenAPI in a client/server environment. OpenAPI functions are called asynchronously: each OpenAPI function call is provided with a callback function, which notifies the demo program of the completion of the function call.

Though single-threaded, the demo program consists of two independent execution units: a server unit, which creates, registers, and waits for database events, and a client unit, which periodically raises database events. The main processing control in the demo program initializes one, the other, or both of the execution units, then loops calling `IIapi_wait()` until all asynchronous activity has ended. Each execution unit, when initialized, issues its first asynchronous request. The units regain execution control when their requests complete and continue issuing new requests until they reach their termination conditions.

## How the Client Execution Unit Works

The client execution unit demonstrates asynchronous processing using unique callbacks. Each asynchronous OpenAPI call is given a unique callback function, which determines the next action to be performed when the call completes. Each callback function checks the OpenAPI function call status, processes any result data, and determines the next action to be performed. Operation continues by making a new asynchronous OpenAPI function call and providing the next function in the processing sequence as the callback function. After all requested database events have been raised (including the optional server termination event), the client unit stops making asynchronous OpenAPI requests.

## How the Server Execution Unit Works

The server execution unit demonstrates asynchronous processing using a Finite State Machine. A single callback function is used for all asynchronous OpenAPI calls. A control block passed with the asynchronous call contains the information necessary, including the current state, to process the call results and determine the next state and actions to be performed. The server unit continues waiting for and processing database events until the termination event is received. Upon receiving the termination event, the server unit cancels any active requests, frees resources, and terminates.

## How You Can Run the Demo Program

You can run the demo program in a variety of configurations: client-only, server-only, and client and server. You can run multiple instances of the demo program simultaneously. Each server instance receives and displays events raised by each client instance.

You run the demo program with the following command line syntax:

```
apiademo [-s] [-c] [-t] [-i] db [n]
```

This command uses the following parameters:

**-s**

Runs the server execution unit.

**-c**

Runs the client execution unit.

**-t**

Sends termination indication to demonstration servers.

**db**

Specifies the name of the target database.

**n**

Specifies the number of events to be raised by the client.

**Default:** 5

### Examples: apiademo command

The following commands are equivalent:

This command...	Is the same as this command...
apiademo dbname	apiademo -s -c -t dbname 5

<b>This command...</b>	<b>Is the same as this command...</b>
<code>apiademo -c dbname</code>	<code>apiademo -c dbname 5</code>
<code>apiademo -t dbname</code>	<code>apiademo -c -t db 0</code>

## Source File Descriptions

The asynchronous sample source files are as follows:

### **apiademo.c**

Demonstrates main processing control.

### **apiacnt.c**

Demonstrates client execution unit.

### **apiasvr.c**

Demonstrates server execution unit.

### **apiautil.c**

Demonstrates utility functions.

# Chapter 7: Using Repeat Queries with OpenAPI

---

This section contains the following topics:

[Repeat Queries](#) (see page 185)

[Repeat Query ID](#) (see page 186)

[Query Parameters](#) (see page 187)

[Repeat Query Parameters](#) (see page 188)

This chapter provides information on implementing repeat queries using OpenAPI, unique repeat query IDs, and the query parameter mechanism used for repeat queries.

## Repeat Queries

Embedded SQL provides the ability to mark delete, insert, select, and update statements with the keyword `repeated`. Through a protocol hidden by the embedded preprocessor and implemented by the embedded runtime system, these queries are optimized in the server for repeated execution. The queries are shared among all clients running a given application.

OpenAPI also lets you use repeat queries, though not with the ease provided by embedded SQL. For an OpenAPI application to use repeat queries, it must adhere to the same protocol implemented in the embedded runtime system. This protocol is explained in the following paragraphs.

## How the Repeat Query Protocol Works

You must first define a repeat query for a given server connection. When a repeat query is defined, the server determines if another client has defined the repeat query previously. This is done through a unique repeat query ID that the application provides. If the repeat query has not been defined previously, the server builds a query plan for the repeat query. All subsequent attempts to define the repeat query for that server inherits the original query plan.

Once a repeat query has been defined, the application invokes (executes) the query. The repeat query can be invoked repeatedly once it has been defined, and subsequent definitions are not generally required. However, it is possible that a server will need to drop a query plan for a repeat query, and the application must be prepared for such an occurrence. If the server returns an indication that a repeat query is no longer defined, the application must redefine the query and redo the invocation that failed.

An OpenAPI application defines a repeat query by calling `IIapi_query()` with the query type set to `IIAPI_QT_DEF_REPEAT_QUERY` and the query text containing the statement to be repeatedly executed (not including the repeated keyword). A unique repeat query ID (see page 186) is passed as a parameter using `IIapi_setDescriptor()` and `IIapi_putParms()`. If there are no errors processing the query, the application obtains a repeat query handle by calling `IIapi_getQueryInfo()`. The repeat query handle will be used to invoke the repeat query.

To invoke a repeat query, the application calls `IIapi_query()` with the query type set to `IIAPI_QT_EXEC_REPEAT_QUERY` and no query text. The repeat query handle returned by `IIapi_getQueryInfo()` when the query was defined is passed as a query parameter using `IIapi_setDescriptor()` and `IIapi_putParms()`. If an OpenAPI error code of `E_AP0014_INVALID_REPEAT_ID` is returned (or the `IIAPI_GQF_UNKNOWN_REPEAT_QUERY` flag is returned by `IIapi_getQueryInfo()`), the application must redefine and invoke the repeat query.

## Repeat Query ID

A server uses a unique repeat query ID to identify a repeat query being used by multiple clients running the same application. This ID must be unique across all applications and all versions of a given application. A repeat query ID consists of two integers (32 bits each) and a character string (64 bytes maximum). Embedded SQL uses the source file name, a positional value, and a timestamp to uniquely identify a query at preprocessing time.

## Compile-time and Runtime IDs

An OpenAPI application can use either a compile-time or runtime ID for repeat queries. A compile-time ID is more difficult to maintain, while a runtime ID is less efficient because different clients running the same application will not share repeat queries.

The application programmer must maintain a compile-time ID. If a query is changed and recompiled, the query ID must be changed to distinguish the new query from the previous version. The application programmer must create the query ID in such a way as to guarantee uniqueness, not just in that application, but across **all** existing applications.

A runtime ID can use information about the client runtime environment, such as machine name and process ID, to create a unique ID for that client. This is easier to manage than a compile-time ID, but clients running the same application and executing identical queries will duplicate resources on the server. OpenAPI can create a runtime-unique repeat query ID for the application. The application is not required to provide a repeat query ID when the repeat query is defined. If the repeat query ID is omitted, OpenAPI generates a runtime-unique ID that uses host, process, timestamp, and application profile information. An application should not mix usage of OpenAPI-generated IDs and application-provided IDs.

## Query Parameters

Repeat queries usually require parameters that take on different values each time the query is invoked. These parameters are represented by host variables in embedded SQL. OpenAPI does not support embedded host variable references and dynamic parameter markers cannot be used since the query is not being prepared (although the define/invoke mechanism appears similar to the prepare/execute scenario of dynamic SQL).

This section describes a parameter marker mechanism used by embedded SQL to implement support for host variables. An OpenAPI application can use this mechanism as an alternative to dynamic SQL for standard queries. The following section extends the parameter marker mechanism for use in repeat queries.

## How the ~V Mechanism Works

In place of dynamic parameter markers (?) and embedded host variables (:host\_var), you can form a query using the parameter marker ~V. A ~V marker can appear any place a dynamic parameter marker or embedded host variable is allowed. The ~V marker must be preceded and followed by a space character. Values for the parameter markers are sent with the query using `IIapi_setDescriptor()` and `IIapi_putParms()`, in the same order the parameter markers appear in the query text.

As an example, the following query:

```
select * from employee where dept = ~V and age > ~V
```

when sent with parameter values of 'admin' and 35 is identical to the query:

```
select * from employee where dept = 'admin' and age > 35
```

The advantage is that the query using ~V does not require the application to build the query text at runtime using values not available at compile time; neither does it require the use of a cursor to perform the query dynamically.

### **More information:**

Example: Repeat query using the ~V marker (see page 189)

## Repeat Query Parameters

The query parameters (see page 187) are evaluated when the query is parsed. For repeat queries, this occurs when the query is defined, which means the values will be fixed at execution time, regardless of the values sent when executed.

Repeat query parameters that take on unique values at execution time are possible using an extension to the query parameter mechanism described in Query Parameters (see page 187). These extended parameter markers take the form:

```
 $\$n = \sim V$ 
```

where  $n$  is replaced by sequentially increasing integers starting with 0.

## Example: Repeat query using the ~V marker

The query from the previous section, if used as a repeat query would be:

```
select * from employee where dept = $0 = ~V and age > $1 = ~V
```

When implementing repeat queries, parameter values must be sent when the repeat query is defined, as well as when the query is invoked. Query text is not sent when a repeat query is invoked, but the parameters sent at invocation must be in the same order as when the repeat query was defined.

### **More information:**

How the ~V Mechanism Works (see page 188)



# Appendix A: Error Handling

---

This section contains the following topics:

[Error Codes](#) (see page 191)

This appendix describes how an application using OpenAPI checks for and handles errors.

## Error Codes

OpenAPI generates the following error codes.

### **E\_AP0001\_CONNECTION\_ABORTED**

The connection between the application and the server has been severed. A server error message also may be available.

### **E\_AP0002\_TRANSACTION\_ABORTED**

The server aborted the current transaction. A server error message also may be available.

### **E\_AP0003\_ACTIVE\_TRANSACTIONS**

The application requested an operation that can be performed only when no transactions are active on the connection.

### **E\_AP0004\_ACTIVE\_QUERIES**

The application requested an operation that can be performed only when no queries are active on the connection.

### **E\_AP0005\_ACTIVE\_EVENTS**

The application requested an operation that can be performed only when no database event requests are active on the connection.

### **E\_AP0006\_INVALID\_SEQUENCE**

The application invoked a function that violated the OpenAPI order of processing.

**E\_AP0007\_INCOMPLETE\_QUERY**

The application attempted to close a query that had not yet completed processing. The application should cancel the query instead.

**E\_AP0008\_QUERY\_DONE**

The application attempted to cancel a query that had completed processing. The application should close the query instead.

**E\_AP0009\_QUERY\_CANCELLED**

The query was cancelled by a call to `IIapi_cancel()`.

**E\_AP000A\_QUERY\_INTERRUPTED**

Copy statement processing interrupted by the server.

**E\_AP000B\_COMMIT\_FAILED**

The server was unable to commit the transaction.

**E\_AP000C\_2PC\_REFUSED**

The server was unable to prepare the distributed transaction to be committed. The transaction should be rolled back.

**E\_AP000D\_PARAMETERS\_REQUIRED**

The specified query type requires parameters, but the application indicated that no parameters would be provided.

**E\_AP000E\_INVALID\_CONNECT\_PARM**

The server rejected a connection parameter or value.

**E\_AP000F\_NO\_CONNECT\_PARMS**

`IIapi_modifyConnect()` was called with no prior calls to `IIapi_setConnectParam()`.

**E\_AP0010\_INVALID\_COLUMN\_COUNT**

The number of parameter or column values being sent or retrieved did not match what the OpenAPI expected.

**E\_AP0011\_INVALID\_PARAM\_VALUE**

A parameter value was not within the permissible range of values.

**E\_AP0012\_INVALID\_DESCR\_INFO**

The ds\_dataType or ds\_columnType of at least one of the descriptor entries was invalid.

**E\_AP0013\_INVALID\_POINTER**

A required pointer parameter was NULL.

**E\_AP0014\_INVALID\_REPEAT\_ID**

The repeat query being executed was no longer available to the server. The application should define and re-execute the repeat query.

**E\_AP0015\_INVALID\_TRANS\_STMT**

The application attempted a transaction commit or rollback by using IIapi\_query(). Use IIapi\_commit() or IIapi\_rollback() to manage transactions.

**E\_AP0016\_ROW\_DISCARDED**

A cursor pre-fetch result row was received, but all OpenAPI buffers were filled with unread result rows. The row was discarded.

**E\_AP0017\_SEGMENT\_DISCARDED**

The column and row count parameters for IIapi\_getColumns() only permitted the first segment of a BLOB column to be returned.

**E\_AP0018\_INVALID\_DISCONNECT**

A disconnect request was made while the connection was already in the process of disconnecting.

**E\_AP0020\_BYREF\_UNSUPPORTED**

BYREF procedure parameters are not supported at connection level IIAPI\_LEVEL\_0.

**E\_AP0021\_GTT\_UNSUPPORTED**

Global Temporary Table procedure parameters are not supported at connection level IIAPI\_LEVEL\_0.

**E\_AP0023\_INVALID\_NULL\_DATA**

A data value was marked as NULL (dv\_null set to TRUE) when the associated descriptor indicated non-nullable (ds\_nullable set to FALSE).

**E\_AP0024\_INVALID\_DATA\_SIZE**

The ds\_length of at least one of the descriptor entries was invalid.

**E\_AP0025\_SVC\_DATA\_TYPE**

The ds\_dataType of at least one of the descriptor entries, whose ds\_columntType was IIAPI\_COL\_SVCPARM, was invalid.

**E\_AP0028\_LVL1\_DATA\_TYPE**

The ds\_dataType of at least one of the descriptor entries specified a type that is not supported below connection level IIAPI\_LEVEL\_1.

**E\_AP0029\_LVL2\_DATA\_TYPE**

The ds\_dataType of at least one of the descriptor entries specified a type that is not supported below connection level IIAPI\_LEVEL\_2.

**E\_AP002A\_LVL3\_DATA\_TYPE**

The ds\_dataType of at least one of the descriptor entries specified a type that is not supported below connection level IIAPI\_LEVEL\_3.

**E\_AP002B\_LVL4\_DATA\_TYPE**

The ds\_dataType of at least one of the descriptor entries specified a type that is not supported below connection level IIAPI\_LEVEL\_4.

**E\_AP002C\_LVL5\_DATA\_TYPE**

The ds\_dataType of at least one of the descriptor entries specified a type that is not supported below connection level IIAPI\_LEVEL\_5.

## SQLSTATE Values and Descriptions

OpenAPI- and server-generated errors are accompanied by an SQLSTATE value. SQLSTATE is a standard way to report SQL errors. The following table contains SQLSTATE values that may accompany an OpenAPI or server error:

SQLSTATE	Value	Description
II_SS00000_SUCCESS	00000	Success
II_SS01000_WARNING	01000	Warning
II_SS02000_NO_DATA	02000	No more data
II_SS08004_CONNECTION_REJECTED	08004	Connection rejected
II_SS08006_CONNECTION_FAILURE	08006	Connection failure
II_SS0A500_INVALID_QRY_LANG	0A500	Invalid query language
II_SS21000_CARD_VIOLATION	21000	Cardinality violation
II_SS22000_DATA_EXCEPTION	22000	Data exception

<b>SQLSTATE</b>	<b>Value</b>	<b>Description</b>
II_SS22001_STRING_RIGHT_TRUNC	22001	String data, right truncation
II_SS22002_NULLVAL_NO_IND_PARAM	22002	Null value, no indicator parameter
II_SS22003_NUM_VAL_OUT_OF_RANGE	22003	Numeric value out of range
II_SS22005_ASSIGNMENT_ERROR	22005	Error in assignment
II_SS22008_DATETIME_FLD_OVFLOW	22008	Datetime field overflow
II_SS22011_SUBSTRING_ERROR	22011	Substring error
II_SS22012_DIVISION_BY_ZERO	22012	Division by zero
II_SS22015_INTERNAL_FLD_OVFLOW	22015	Internal field overflow
II_SS22022_INDICATOR_OVFLOW	22022	Indicator overflow
II_SS22500_INVALID_DATA_TYPE	22500	Invalid data type
II_SS23000_CONSTR_VIOLATION	23000	Integrity constraint violation
II_SS23501_UNIQUE_CONS_VIOLATION	23501	UNIQUE constraint violation
II_SS24000_INV_CURS_STATE	24000	Invalid cursor state
II_SS25000_INV_XACT_STATE	25000	Invalid transaction state
II_SS26000_INV_SQL_STMT_NAME	26000	Invalid SQL statement name
II_SS27000_TRIG_DATA_CHNG_ERR	27000	Triggered data change violation
II_SS28000_INV_AUTH_SPEC	28000	Invalid authorization specification
II_SS40001_SERIALIZATION_FAIL	40001	Serialization failure
II_SS42000_SYN_OR_ACCESSERR	42000	Syntax error or access rule violation
II_SS42500_TBL_NOT_FOUND	42500	Table not found
II_SS42501_COL_NOT_FOUND	42501	Column not found
II_SS42502_DUPL_OBJECT	42502	Duplicate object
II_SS42503_INSUF_PRIV	42503	Insufficient privilege
II_SS42504_UNKNOWN_CURSOR	42504	Cursor not found
II_SS42506_INVALID_IDENTIFIER	42506	Invalid identifier
II_SS50001_INVALID_DUP_ROW	50001	Invalid duplicate row
II_SS50002_LIMIT_EXCEEDED	50002	Limit has been exceeded
II_SS50003_EXHAUSTED_RESOURCE	50003	Resource exhausted
II_SS50004_SYS_CONFIG_ERROR	50004	System configuration error
II_SS50005_GW_ERROR	50005	Gateway-related error

<b>SQLSTATE</b>	<b>Value</b>	<b>Description</b>
II_SS50006_FATAL_ERROR	50006	Fatal error
II_SS50007_INVALID_SQL_STMT_ID	50007	Invalid SQL statement ID
II_SS50008_UNSUPPORTED_STMT	50008	Unsupported statement
II_SS50009_ERROR_RAISED_IN_DBPROC	50009	Procedure error raised
II_SS5000A_QUERY_ERROR	5000A	Query error
II_SS5000B_INTERNAL_ERROR	5000B	Internal error
II_SS5000C_FETCH_ORIENTATION	5000C	Fetch orientation has value zero
II_SS5000D_INVALID_CURSOR_NAME	5000D	Invalid cursor name
II_SS5000E_DUP_STMT_ID	5000E	Duplicate SQL statement ID
II_SS5000H_UNAVAILABLE_RESOURCE	5000H	Unknown or unavailable resource
II_SS01001_CURS_OPER_CONFLICT	01001	Cursor operation conflict
II_SS01002_DISCONNECT_ERROR	01002	Error during disconnect
II_SS01003_NULL_ELIM_IN_SETFUNC	01003	NULL eliminated in set() function
II_SS01004_STRING_RIGHT_TRUNC	01004	String data, right truncation
II_SS01005_INSUF_DESCR_AREAS	01005	Insufficient descriptor items
II_SS01006_PRIV_NOT_REVOKED	01006	Privilege not revoked
II_SS01007_PRIV_NOT_GRANTED	01007	Privilege not granted
II_SS01008_IMP_ZERO_BIT_PADDING	01008	Implicit zero-bit padding
II_SS01009_SEARCH_COND_TOO_LONG	01009	Search condition too long
II_SS0100A_QRY_EXPR_TOO_LONG	0100A	Query expression too long
II_SS01500_LDB_TBL_NOT_DROPPED	01500	Star local database table not dropped
II_SS01501_NO_WHERE_CLAUSE	01501	No where clause: update, delete
II_SS07000_DSQL_ERROR	07000	Dynamic SQL error
II_SS07001_USING_PARM_MISMATCH	07001	Using clause/parameter mismatch
II_SS07002_USING_TARG_MISMATCH	07002	Using clause/target mismatch
II_SS07003_CAN_EXEC_CURS_SPEC	07003	Cannot execute cursor spec
II_SS07004_NEED_USING_FOR_PARAMS	07004	Using clause required: params
II_SS07005_STMT_NOT_CURS_SPEC	07005	Prepared statement not cursor specification
II_SS07006_RESTRICT_DT_ATTR_ERR	07006	Restricted data type violation
II_SS07007_NEED_USING_FOR_RES	07007	Using clause required: result

<b>SQLSTATE</b>	<b>Value</b>	<b>Description</b>
II_SS07008_INV_DESCR_CNT	07008	Invalid descriptor count
II_SS07009_INV_DESCR_IDX	07009	Invalid descriptor index
II_SS07500_CONTEXT_MISMATCH	07500	Execution context not valid
II_SS08000_CONNECTION_EXCEPTION	08000	Connection error
II_SS08001_CANT_GET_CONNECTION	08001	Cannot establish connection
II_SS08002_CONNECT_NAME_IN_USE	08002	Connection name in use
II_SS08003_NO_CONNECTION	08003	Connection does not exist
II_SS08007_XACT_RES_UNKNOWN	08007	Transaction result unknown
II_SS08500_LDB_UNAVAILABLE	08500	Star local database unavailable
II_SS0A000_FEATUR_NOT_SUPPORTED	0A000	Feature not supported
II_SS0A001_MULT_SERVER_XACTS	0A001	Multiple server transactions
II_SS22007_INV_DATETIME_FMT	22007	Invalid datetime format
II_SS22009_INV_TZ_DISPL_VAL	22009	Invalid timezone displacement
II_SS22018_INV_VAL_FOR_CAST	22018	Invalid character value in cast
II_SS22019_INV_ESCAPE_CHAR	22019	Invalid escape character
II_SS22021_CHAR_NOT_IN_RPRTR	22021	Character not in set
II_SS22023_INV_PARAM_VAL	22023	Invalid parameter value
II_SS22024_UNTERM_C_STRING	22024	Unterminated string
II_SS22025_INV_ESCAPE_SEQ	22025	Invalid escape sequence
II_SS22026_STRING_LEN_MISMATCH	22026	String data, length mismatch
II_SS22027_TRIM_ERROR	22027	Error in trim() function
II_SS2B000_DEP_PRIV_EXISTS	2B000	Dependent privileges exist
II_SS2C000_INV_CH_SET_NAME	2C000	Invalid character set name
II_SS2D000_INV_XACT_TERMINATION	2D000	Invalid transaction termination
II_SS2E000_INV_CONN_NAME	2E000	Invalid connection name
II_SS33000_INV_SQL_DESCR_NAME	33000	Invalid descriptor name
II_SS34000_INV_CURS_NAME	34000	Invalid cursor name
II_SS35000_INV_COND_NUM	35000	Invalid condition number
II_SS3C000_AMBIG_CURS_NAME	3C000	Ambiguous cursor name
II_SS3D000_INV_CAT_NAME	3D000	Invalid catalog name

SQLSTATE	Value	Description
II_SS3F000_INV_SCHEMA_NAME	3F000	Invalid schema name
II_SS40000_XACT_ROLLBACK	40000	Transaction rollback
II_SS40002_CONSTR_VIOLATION	40002	Integrity constraint violation
II_SS40003_STMT_COMPL_UNKNOWN	40003	Statement results unknown
II_SS42505_OBJ_NOT_FOUND	42505	Object not found
II_SS42507_RESERVED_IDENTIFIER	42507	Reserved identifier
II_SS44000_CHECK_OPTION_ERR	44000	With check option violation
II_SS50000_MISC_ING_ERRORS	50000	Miscellaneous Ingres errors
II_SS5000F_TEXTUAL_INFO	5000F	Textual information
II_SS5000G_DBPROC_MESSAGE	5000G	Database procedure message
II_SS5000I_UNEXP_LDB_SCHEMA_CHNG	5000I	Star local database schema change
II_SS5000J_INCONSISTENT_DBMS_CAT	5000J	Inconsistent DBMS catalog
II_SS5000K_SQLSTATE_UNAVAILABLE	5000K	SQLSTATE code unavailable
II_SS5000L_PROTOCOL_ERROR	5000L	Protocol error
II_SS5000M_IPC_ERROR	5000M	IPC error
II_SS5000N_OPERAND_TYPE_MISMATCH	5000N	Operand type mismatch
II_SS5000O_INVALID_FUNC_ARG_TYPE	5000O	Invalid function argument
II_SS5000P_TIMEOUT_ON_LOCK_REQUEST	5000P	Lock request timeout
II_SS5000Q_DB_REORG_INVALIDATED_QP	5000Q	Query plan invalidated
II_SS5000R_RUN_TIME_LOGICAL_ERROR	5000R	Runtime logical error
II_SSHZ000_RDA	HZ000	Remote database access error

## XA Error Codes

The following symbols define the values that correspond to error codes defined by the XA interface standard.

Symbol	Description
IIAPI_XAER_ASYNC	Asynchronous operation already outstanding
IIAPI_XAER_RMERR	Resource manager error in transaction branch
IIAPI_XAER_NOTA	Invalid XID

<b>Symbol</b>	<b>Description</b>
IIAPI_XAER_INVALID	Invalid arguments
IIAPI_XAER_PROTO	Routine invoked in improper context
IIAPI_XAER_RMFAIL	Resource manager unavailable
IIAPI_XAER_DUPID	XID already exists
IIAPI_XAER_OUTSIDE	Resource manager doing work outside global transaction
IIAPI_XA_RDONLY	Transaction branch read-only
IIAPI_XA_RETRY	Routine returned with no effect and may be reissued
IIAPI_XA_HEURMIX	Transaction branch heuristically committed and rolled back
IIAPI_XA_HEURRB	Transaction branch heuristically rolled back
IIAPI_XA_HEURCOM	Transaction branch heuristically committed
IIAPI_XA_HEURHAZ	Transaction branch may have been heuristically completed
IIAPI_XA_NOMIGRATE	Resumption must occur where suspension occurred
IIAPI_XA_RBROLLBACK	Rollback caused by unspecified reason
IIAPI_XA_RBCOMMFAIL	Rollback caused by communication failure
IIAPI_XA_RBDEADLOCK	Deadlock was detected
IIAPI_XA_RBINTEGRITY	Resource integrity violation detected
IIAPI_XA_RBOTHER	Rollback caused by some other reason
IIAPI_XA_RBPROTO	Protocol error in resource manager
IIAPI_XA_RBTIMEOUT	Transaction branch timed out
IIAPI_XA_RBTRANSIENT	Transaction branch may be retried



# Appendix B: Data Types

---

This section contains the following topics:

[Ingres Data Types](#) (see page 201)

[Data Type Descriptions](#) (see page 204)

This appendix describes the Ingres data types used by OpenAPI.

## Ingres Data Types

Ingres data types are described by the information conveyed in the `IIAPI_DESCRIPTOR` structure. This structure describes the data type, length, and precision of OpenAPI data.

The following table provides the data type, length, and precision value for each Ingres data type and maps the data type to its corresponding C type and SQL type. The following symbols are used in the table:

***n***

Specifies the length of the specific data value. This value varies, but is limited by the maximum length supported by the DBMS, typically between 2 KB and 32 KB.

***p***

Specifies the precision for the specific data value. This value varies, but is limited by the maximum precision allowed by the length of the data.

***s***

Specifies the scale for the specific data value. This value varies, but is limited by the precision value.

Lengths, precisions, or scales that are not required are shown as 0 in the table.

If the C type is shown as none, a character buffer of the appropriate length can be used to store the data value and `IIapi_convertData()` or `IIapi_formatData()` can be used to convert the data value to a type that has a corresponding C type.

The following table lists the Ingres data types:

<b>Data Type</b>	<b>Length</b>	<b>Precision/ Scale</b>	<b>C Type</b>	<b>SQL Type</b>
IIAPI_BYTE_TYPE	<i>n</i>	0/0	char*	byte( <i>n</i> )
IIAPI_CHA_TYPE	<i>n</i>	0/0	char*	char( <i>n</i> )
IIAPI_CHR_TYPE	<i>n</i>	0/0	char*	c( <i>n</i> )
IIAPI_HNDL_TYPE	4	0/0	void*	n/a
IIAPI_DATE_TYPE	4	0/0	none	ansidate
IIAPI_DEC_TYPE	1-16 (dep. on precision)	<i>p/s</i>	none	decimal( <i>p,s</i> )
IIAPI_INTDS_TYPE	12	<i>p/0</i>	none	interval day to second
IIAPI_DTE_TYPE	12	0/0	none	ingresdate
IIAPI_FLT_TYPE	4 8	<i>p/0</i>	float double	float4/real float8/float
IIAPI_INT_TYPE	1 2 4 8	0/0	char short long long long	integer1 integer2/smallint integer4/integer integer8/bigint
IIAPI_INTYM_TYPE	3	<i>p/0</i>	none	interval year to month
IIAPI_LOGKEY_TYPE	16	0/0	char*	object_key
IIAPI_LBYTE_TYPE	(max 2 GB)	0/0	char*	long byte
IIAPI_NBLOC_TYPE	4	0/0	long	long byte locator
IIAPI_NCLOC_TYPE	4	0/0	long	long varchar locator
IIAPI_LNLOC_TYPE	4	0/0	long	long nvarchar locator
IIAPI_LNVCH_TYPE	(max 2 GB)	0/0	short*	long nvarchar
IIAPI_LVCH_TYPE	(max 2 GB)	0/0	char*	long varchar
IIAPI_LTXT_TYPE	<i>n</i>	0/0	char*	null
IIAPI_MNY_TYPE	8	<i>p/0</i>	none	money
IIAPI_NCHA_TYPE	<i>n</i>	0/0	short*	nchar( <i>n</i> )
IIAPI_NVCH_TYPE	<i>n</i>	0/0	short*	nvarchar( <i>n</i> )
IIAPI_TABKEY_TYPE	8	0/0	char*	table_key
IIAPI_TIME_TYPE	10	<i>p/0</i>	none	time with local time zone

<b>Data Type</b>	<b>Length</b>	<b>Precision/ Scale</b>	<b>C Type</b>	<b>SQL Type</b>
IIAPI_TMWO_TYPE	10	<i>p</i> /0	none	time without time zone
IIAPI_TMTZ_TYPE	10	<i>p</i> /0	none	time with time zone
IIAPI_TSWO_TYPE	14	<i>p</i> /0	none	timestamp without time zone
IIAPI_TSTZ_TYPE	14	<i>p</i> /0	none	timestamp with time zone
IIAPI_TS_TYPE	14	<i>p</i> /0	none	timestamp with local time zone
IIAPI_TXT_TYPE	<i>n</i>	0/0	char*	text( <i>n</i> )
IIAPI_VBYTE_TYPE	<i>n</i>	0/0	char*	varbyte( <i>n</i> )
IIAPI_VCH_TYPE	<i>n</i>	0/0	char*	varchar( <i>n</i> )

The following symbols are defined for the values in the Length column of the table:

- IIAPI\_I1\_LEN
- IIAPI\_I2\_LEN
- IIAPI\_I4\_LEN
- IIAPI\_I8\_LEN
- IIAPI\_F4\_LEN
- IIAPI\_F8\_LEN
- IIAPI\_MNY\_LEN
- IIAPI\_DTE\_LEN (Synonym: IIAPI\_IDATE\_LEN)
- IIAPI\_TIME\_LEN
- IIAPI\_DATE\_LEN (Synonym: IIAPI\_ADATE\_LEN)
- IIAPI\_TS\_LEN
- IIAPI\_INTYM\_LEN
- IIAPI\_INTDS\_LEN
- IIAPI\_LOGKEY\_LEN
- IIAPI\_TABKEY\_LEN
- IIAPI\_LOCATOR\_LEN

## Data Type Descriptions

Ingres data types are described as follows.

### **IIAPI\_BYTE\_TYPE**

Specifies a fixed-length binary string containing data with the declared length. The declared length is stored as the `ds_length` parameter in the corresponding data descriptor.

**Limits:** Length is 1 to DBMS maximum bytes.

### **IIAPI\_CHA\_TYPE**

Specifies a fixed-length character string that is stored blank-padded to the declared length. The declared length is stored as the `ds_length` parameter in the corresponding data descriptor. Embedded blanks are significant. Valid characters for this data type include printing, non-printing, and NULL characters.

**Limits:** Length is 1 to DBMS maximum characters.

### **IIAPI\_CHR\_TYPE**

Specifies a fixed-length character string that is stored blank-padded to the declared length. The declared length is stored as the `ds_length` parameter in the corresponding data descriptor. Embedded blanks are insignificant. Valid characters for this data type include printing characters only. This data type is supported for previous Ingres versions; when possible, use `IIAPI_CHA_TYPE`.

**Limits:** Length is 1 to DBMS maximum characters.

### **IIAPI\_HNDL\_TYPE**

Specifies a data type used only by OpenAPI and the application. This data type describes a handle, which is a pointer to one of the control blocks created by OpenAPI. OpenAPI translates the information in the control block into data acceptable by the data source. This data type does not appear in queries.

**IIAPI\_DATE\_TYPE**

Specifies an ANSI date data type that is stored in 4 bytes. IIApi\_convertData() or IIApi\_formatData() can be used to convert to character string representation.

IIAPI\_ADATE\_TYPE is a synonym for IIAPI\_DATE\_TYPE.

**IIAPI\_DEC\_TYPE**

Specifies a packed-decimal data type stored in 1 to 16 bytes depending on the precision of the value as follows:  $len = (precision)/2 + 1$ .

IIApi\_convertData() or IIApi\_formatData() can be used to convert to character string or floating point representation.

**IIAPI\_DTE\_TYPE**

Specifies an Ingres internal date data type that is stored in 12 bytes. IIApi\_convertData() or IIApi\_formatData() can be used to convert to character string or floating point representation.

IIAPI\_IDATE\_TYPE is a synonym for IIAPI\_DTE\_TYPE.

**IIAPI\_FLT\_TYPE**

Specifies a floating-point data type.

**Limits:** Data value range is -1.0e+38 to +1.0e+38.

**IIAPI\_INT\_TYPE**

Specifies a data type supporting varying data value ranges. It is dependent on the ds\_length of the data value.

**Limits:**

If the length is 1, the range is -128 to +127.

If the length is 2, the range is -32,768 to +32,767.

If the length is 4, the range is -2,147,483,648 to +2,147,483,647.

If the length is 8, the range is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

#### **IIAPI\_INTDS\_TYPE**

Specifies an ANSI day to second interval data type that is stored in 12 bytes. `IIapi_convertData()` or `IIapi_formatData()` can be used to convert to character string representation.

#### **IIAPI\_INTYM\_TYPE**

Specifies an ANSI year to date interval data type that is stored in 3 bytes. `IIapi_convertData()` or `IIapi_formatData()` can be used to convert to character string representation.

#### **IIAPI\_LOGKEY\_TYPE**

Specifies a data type with values unique within the data source. An application should not attempt to update a column containing a system-maintained object key.

#### **IIAPI\_LBYTE\_TYPE**

Specifies a variable-length binary string. The actual length of the large byte segment is stored in the first two bytes of the data buffer, followed by the large byte data.

**Limits:** Length is 0 to 2,000,000,000 bytes.

#### **IIAPI\_LNVCH\_TYPE**

Specifies a variable-length National Character Set string. Each character in the string is represented by an unsigned two-byte integer holding a UTF-16-encoded character. The actual length in characters of the large nvarchar segment is stored in the first two bytes of the data buffer, followed by the large nvarchar data.

**Limits:** Length is 0 to 1,000,000,000 NCS characters.

**IIAPI\_LVCH\_TYPE**

Specifies a variable-length character string. The actual length of the large varchar segment is stored in the first two bytes of the data buffer, followed by the large varchar data. Valid characters for this data type include printing, non-printing, and NULL characters. Trailing blanks are insignificant in a varchar data type.

**Limits:** Length is 0 to 2,000,000,000 characters.

**IIAPI\_LTXT\_TYPE**

Specifies a variable-length character string. The maximum size for this data type is stored as the ds\_length parameter in the corresponding descriptor. The actual length of the text data is stored in the first two bytes of the data buffer, followed by the text data.

Valid characters for this data type include printing and nonprinting characters. All blanks are significant in a text data buffer. This type is supported for typeless NULL values. A NULL value of this type can be coerced into any other data type. Otherwise, use IIAPI\_VCH\_TYPE.

**Limits:** Length is 1 to DBMS maximum characters.

**IIAPI\_LCLOC\_TYPE**

Specifies a long character locator—a reference to a long varchar string that resides in the database. A locator value is an unsigned 4-byte integer.

**IIAPI\_LBLOC\_TYPE**

Specifies a long binary locator—a reference to a long byte string that resides in the database. A locator value is an unsigned 4-byte integer.

**IIAPI\_LNLOC\_TYPE**

Specifies a long National Character Set locator—a reference to a long nvarchar string that resides in the database. A locator value is an unsigned 4-byte integer.

**IIAPI\_MNY\_TYPE**

Specifies the Ingres money data type stored in 8 bytes with two fixed decimal places. IIAPI\_convertData() or IIAPI\_formatData() can be used to convert to character string or floating point representation.

**IIAPI\_NCHA\_TYPE**

Specifies a fixed length National Character Set string that is stored blank-padded to the declared length. Each character in the string is represented by an unsigned two-byte integer holding a UTF-16 encoded character. The full byte length (twice the declared character length) is stored as the ds\_length parameter in the corresponding data descriptor.

**Limits:** Length is 1 to DBMS maximum NCS characters.

### **IIAPI\_NVCH\_TYPE**

Specifies a variable-length National Character Set string. Each character in the string is represented by an unsigned two-byte integer holding a UTF-16 encoded character. The maximum size in bytes for nvarchar data is stored as the ds\_length parameter in the corresponding descriptor. The actual length in characters of the varchar data is stored in the first two bytes of the data buffer, followed by the nvarchar data.

**Limits:** Length is 0 to DBMS maximum NCS characters.

### **IIAPI\_TABKEY\_TYPE**

Specifies a data type with a value that is unique within the table where the table key is stored. An application should not attempt to update a column containing a system-maintained table key.

### **IIAPI\_TIME\_TYPE**

Specifies an Ingres time data type with local time zone that is stored in 10 bytes. Iiapi\_convertData() or Iiapi\_formatData() can be used to convert to character string representation.

### **IIAPI\_TMTZ\_TYPE**

Specifies an ANSI time data type with timezone that is stored in 10 bytes. Iiapi\_convertData() or Iiapi\_formatData() can be used to convert to character string representation.

### **IIAPI\_TMWO\_TYPE**

Specifies an ANSI time data type without time zone that is stored in 10 bytes. Iiapi\_convertData() or Iiapi\_formatData() can be used to convert to character string representation.

### **IIAPI\_TS\_TYPE**

Specifies an Ingres timestamp data type with local timezone that is stored in 14 bytes. Iiapi\_convertData() or Iiapi\_formatData() can be used to convert to character string representation.

### **IIAPI\_TSTZ\_TYPE**

ANSI timestamp data type with timezone that is stored in 14 bytes. Iiapi\_convertData() or Iiapi\_formatData() can be used to convert to character string representation.

### **IIAPI\_TSWO\_TYPE**

Specifies an ANSI timestamp data type without time zone that is stored in 14 bytes. Iiapi\_convertData() or Iiapi\_formatData() can be used to convert to character string representation.

**IIAPI\_TXT\_TYPE**

Specifies a variable-length character string. The maximum size for this data type is stored as the `ds_length` parameter in the corresponding descriptor. The actual length of the text data is stored in the first two bytes of the data buffer, followed by the text data. Valid characters for this data type include printing and non-printing characters. All blanks are significant in a text data buffer. This type is supported for previous Ingres versions; when possible, use `IIAPI_VCH_TYPE`.

**Limits:** Length is 1 to DBMS maximum characters.

**IIAPI\_VBYTE\_TYPE**

Specifies a variable-length binary string. The maximum size for varbyte data is stored as the `ds_length` parameter in the corresponding descriptor. The actual length of the varbyte data is stored in the first two bytes of the data buffer, followed by the varbyte data.

**Limits:** Length is 0 to 2,000 bytes.

**IIAPI\_VCH\_TYPE**

Specifies a variable-length character string. The maximum size for varchar data is stored as the `ds_length` parameter in the corresponding descriptor. The actual length of the varchar data is stored in the first two bytes of the data buffer, followed by the varchar data. Trailing blanks are insignificant in a varchar data type.

**Limits:** Length is 0 to 2,000 characters.



# Index

---

## A

### allocating

- connection handle • 62
- environment handle • 85
- event handle • 56
- statement handle • 95
- transaction handle • 98

### API functions

- generic parameters • 41
  - IIapi\_abort() • 49
  - IIapi\_autocommit() • 50
  - IIapi\_cancel() • 23, 54
  - IIapi\_catchEvent() • 17, 56
  - IIapi\_close() • 21, 43, 59
  - IIapi\_commit() • 20, 61
  - IIapi\_connect() • 17, 18, 62
  - IIapi\_convertData() • 66
  - IIapi\_disconnect() • 68
  - IIapi\_formatData() • 69
  - IIapi\_getColumns() • 24, 37, 70
  - IIapi\_getCopyMap() • 75
  - IIapi\_getDescriptor() • 24, 76
  - IIapi\_getErrorInfo() • 77
  - IIapi\_getEvent() • 79
  - IIapi\_getQueryInfo() • 21, 80
  - IIapi\_initialize() • 85
  - IIapi\_modifyConnect() • 87
  - IIapi\_prepareCommit() • 90
  - IIapi\_putColumns() • 37, 91
  - IIapi\_putParms() • 21, 37, 93
  - IIapi\_query() • 20, 21, 95
  - IIapi\_registerXID() • 20, 98
  - IIapi\_releaseEnv() • 99
  - IIapi\_releaseXID() • 100
  - IIapi\_rollback() • 20, 101
  - IIapi\_savePoint() • 102
  - IIapi\_setConnectParam() • 105
  - IIapi\_setDescriptor() • 21, 113
  - IIapi\_setEnvParam() • 114
  - IIapi\_terminate() • 123
  - IIapi\_wait() • 16, 124
  - mapping to SQL statements • 151
- ### application programming interfaces
- characteristics • 10

- comparison with API • 11
- ### applications
- blocking control from • 124
  - requirements for creating • 177
- ### asynchronous code
- description • 15
  - sample • 182
  - using IIapi\_wait() • 124

## B

- backward compatibility, description • 12
- ### BLOBS
- description • 37
  - retrieving from server • 70
  - sending to server • 91
- blocking control from application • 124

## C

- ### callback
- description • 15
  - generic parameter • 41
- canceling SQL statements • 54
- ### closure
- description • 15
  - generic parameter • 41
- ### committing
- transactions • 61
  - two-phase commit transactions • 90
- compatibility, backward • 12
- ### connection handle
- allocating • 62, 105
  - description • 17
  - freeing • 49, 68
- ### connections
- DBMS server • 18, 62
  - ending • 49
  - parameters • 87, 105
- conversion, data types • 38, 66, 69
- ### copying
- data formats for • 75
  - from file to database • 91
- creating OpenAPI applications • 177
- ### cursors
- description • 25

---

opening • 26

## D

data formats

retrieving from DBMS server • 24, 76

sending to DBMS server • 113

data types

common • 133

conversion • 38, 66, 69

copying unformatted data • 91

data types • 201

IIAPI\_COPYMAP • 138

IIAPI\_DATAVALUE • 140

IIAPI\_DESCRIPTOR • 141

IIAPI\_DT\_ID • 135

IIAPI\_FDATADESCR • 143

IIAPI\_II\_DIS\_TRAN\_ID • 145

IIAPI\_II\_TRAN\_ID • 145

IIAPI\_QUERYTYPE • 136

IIAPI\_STATUS • 137

IIAPI\_SVR\_ERRINFO • 146

IIAPI\_TRAN\_ID • 147

IIAPI\_XA\_DIS\_TRAN\_ID • 148

IIAPI\_XA\_TRAN\_ID • 149

list of • 133

OpenAPI parameter structures • 133

data types, list of • 201

database events

checking for • 79

description • 33

retrieving • 56, 70

DBMS server

closing connection • 68

connecting • 62

connection parameters • 105

delayed output parameters • 13

disconnecting server • 68

distributed transaction, description • 19

## E

ending connections • 49

ending SQL statements • 59

environment handle

allocating • 85

description • 17

freeing • 99

environment variables used by OpenAPI • 177

environment, releasing resources • 99

errors

list of messages • 194

method for handling • 39

returning from DBMS server • 77

event handle

allocating • 56

description • 17

freeing • 59

## F

formats

copying data • 75

retrieving data from DBMS server • 24, 76

sending data to DBMS server • 113

freeing

connection handle • 49, 68

environment handle • 99

event handle • 59

statement handle • 59

transaction handle • 100

## G

GCA (General Communications Architecture),  
relationship to OpenAPI • 11

General Communications Architecture (GCA) •  
11

generic parameters

description • 13, 41

gp\_callback • 41

gp\_closure • 41

gp\_status • 41

## H

header file required for API • 177

## I

II\_API\_LOG environment variable • 178

II\_API\_SET environment variable • 178

II\_API\_TRACE environment variable • 178

IIapi\_abort() • 49

IIapi\_autocommit() • 50

IIapi\_cancel() • 23, 54

IIapi\_catchEvent() • 17, 56

IIapi\_close()

description • 21, 59

sample code • 43

IIapi\_commit() • 20, 61

---

- IIapi\_connect() • 17, 18, 62
- IIapi\_convertData() • 66
- IIAPI\_COPYMAP data type • 138
- IIAPI\_DATAVALUE data type • 140
- IIAPI\_DESCRIPTOR data type • 141
- IIapi\_disconnect() • 68
- IIAPI\_DT\_ID data type • 135
- IIAPI\_FDATADESCR data type • 143
- IIapi\_formatData() • 69
- IIapi\_getColumns() • 24, 37, 70
- IIapi\_getCopyMap() • 75
- IIapi\_getDescriptor()
  - description • 76
  - formats for retrieving data • 24
- IIapi\_getErrorInfo() • 77
- IIapi\_getEvent() • 79
- IIapi\_getQueryInfo() • 21, 80
- IIAPI\_II\_DIS\_TRAN\_ID data type • 145
- IIAPI\_II\_TRAN\_ID data type • 145
- IIapi\_initialize() • 85
- IIapi\_modifyConnect() • 87
- IIapi\_prepareCommit() • 90
- IIapi\_putColumns() • 37, 91
- IIapi\_putParms() • 21, 37, 93
- IIapi\_query()
  - description • 20, 21, 95
  - invoking Name server query statements • 21
  - invoking SQL statements • 21
- IIAPI\_QUERYTYPE data type • 136
- IIapi\_registerXID() • 20, 98
- IIapi\_releaseEnv() • 99
- IIapi\_releaseXID() • 100
- IIapi\_rollback() • 20, 101
- IIapi\_savePoint() • 102
- IIapi\_setConnectParam() • 105
- IIapi\_setDescriptor() • 21, 113
- IIapi\_setEnvParam() • 114
- IIAPI\_STATUS data type • 137
- IIAPI\_SVR\_ERRINFO data type • 146
- IIapi\_terminate() • 123
- IIAPI\_TRAN\_ID data type • 147
- IIapi\_wait() • 16, 124
- IIAPI\_XA\_DIS\_TRAN\_ID data type • 148
- IIAPI\_XA\_TRAN\_ID data type • 149
- immediate output parameters • 13
- initializing API • 85

## L

- library required for API • 177
- local transaction, description • 19
- long byte
  - data type • 37
  - retrieving from server • 70
  - sending to server • 91
- long varchar
  - data type • 37
  - retrieving from server • 70
  - sending to server • 91

## M

- mapping SQL statements to API functions • 151

## N

- Name server
  - closing connection • 68
  - connection parameters • 105
- Name server query statements • 165
  - invoking • 21, 95
  - sending parameters to server • 93

## O

- OpenAPI concepts • 12

## P

- parameter block
  - data types used in • 133
  - description • 13
- parameters
  - delayed output • 13
  - immediate output • 13
  - query type requirements • 161
  - sending to server • 87, 93

## Q

- queries
  - invoking • 95
  - parameter requirements • 161
  - parameters for repeated • 187
  - sending parameters to server • 93

---

## R

- releasing
  - ID for two-phase commit • 100
  - resources for environment handle • 99
- repeat queries
  - description • 185
  - ID • 186
  - parameters • 188
- resources, environment, releasing • 99
- retrieving
  - database events • 56
  - unformatted data • 37
- return status
  - check for completion • 13
  - generic parameters • 41
- returning
  - columns from DBMS server • 70
  - data formats for copy • 75
  - error information • 77
  - SQL statement information • 80
- rolling back
  - partial • 102
  - transactions • 101

## S

- sample code, description • 177
- savepoint, marking for a transaction • 102
- sending
  - connection parameters to server • 87
  - copy information to server • 91
  - query statement parameters to server • 93
  - unformatted data to DBMS server • 37
- server, connecting • 62
- SQL statements
  - canceling • 54
  - ending • 59
  - invoking • 21, 95
  - mapping to API functions • 151
  - query parameters • 161
  - repeated queries • 185
  - retrieving data • 24, 70, 80
  - sending parameters to server • 93
- SQLSTATE, error messages • 194
- starting query statements • 95
- statement handle
  - allocating • 95
  - description • 17
  - freeing • 59

- status, checking • 39
- structures, data types used in • 133
- synchronous code
  - description • 16
  - sample • 179

## T

- terminating OpenAPI • 123
- timeout, initialization • 85
- transaction handle
  - allocating • 98
  - description • 17
  - freeing • 100, 101
- transactions
  - assigning ID for two-phase commit • 98
  - beginning • 95
  - committing • 61
  - data type for defining • 145
  - description • 19
  - enabling autocommit • 50
  - marking a savepoint • 102
  - releasing ID for two-phase commit • 100
  - rolling back • 101
- two-phase commit
  - assigning transaction ID • 98
  - preparation • 90
  - releasing transaction ID • 100

## U

- unformatted data
  - description • 37
  - retrieving from server • 70
  - sending to server • 91

## W

- wait, blocking control from application • 124