Ingres® 2006 Release 3

OpenAPI® User Guide

INGRES®

October 2007

Contents

Chapter 1: Introduction

This guide provides programmers with the information necessary to use Ingres® OpenAPI® to develop applications.

This chapter describes concepts and processes basic to OpenAPI.

Audience

This guide is designed for programmers who want to use OpenAPI to develop applications. Some chapters assume you are already familiar with:

- Ingres components and SQL programming
- The C programming language

Contact Ingres Technical Support

For online technical assistance and a complete list of locations and phone numbers, contact Ingres Technical Support at: http://ingres.com/support.

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.

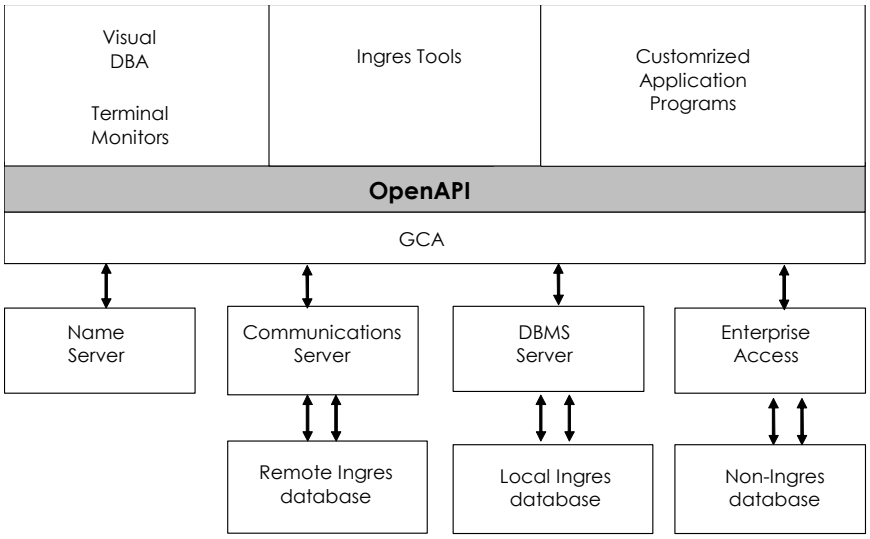
OpenAPI Communication

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 73)
IIapi_connect() Function—Connect to a DBMS Server or Name Server (see page 52)

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_QUERYPARM
{
```

Generic Parameters (containing input and output parameters):

```
        IIAPI_GENPARM    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_QUERYPARM;
```

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

More information:

Generic Parameters (see page 35)

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 15)

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 14)

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 19)

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().

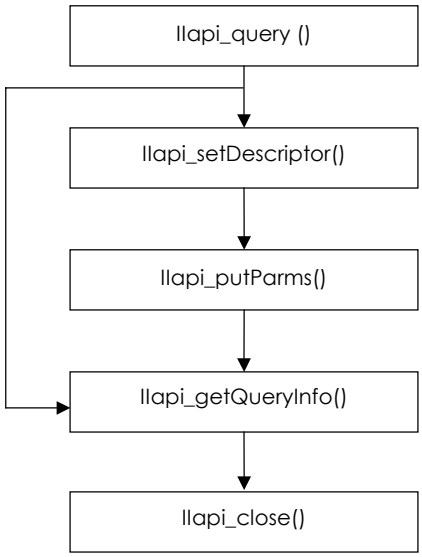
How Query Statements Work

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.

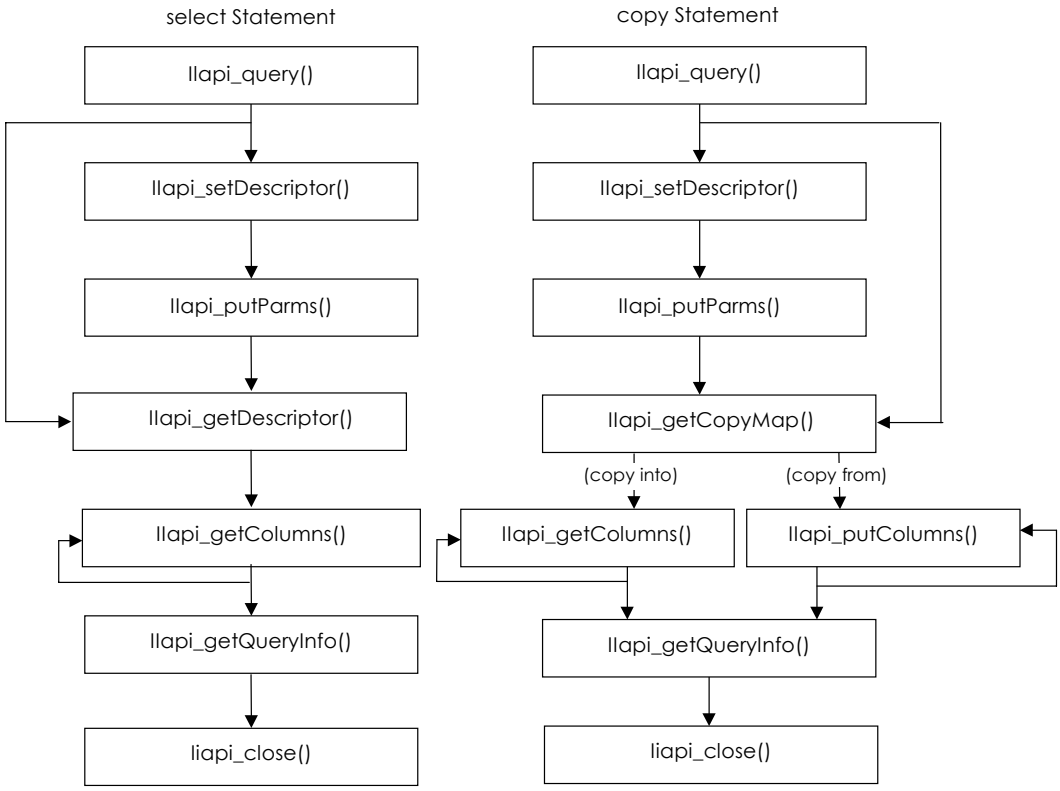
Typical Flow of Operations for SQL and Name Server Query Statement Processing



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.

Updateable 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 updateable 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 updateable (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.

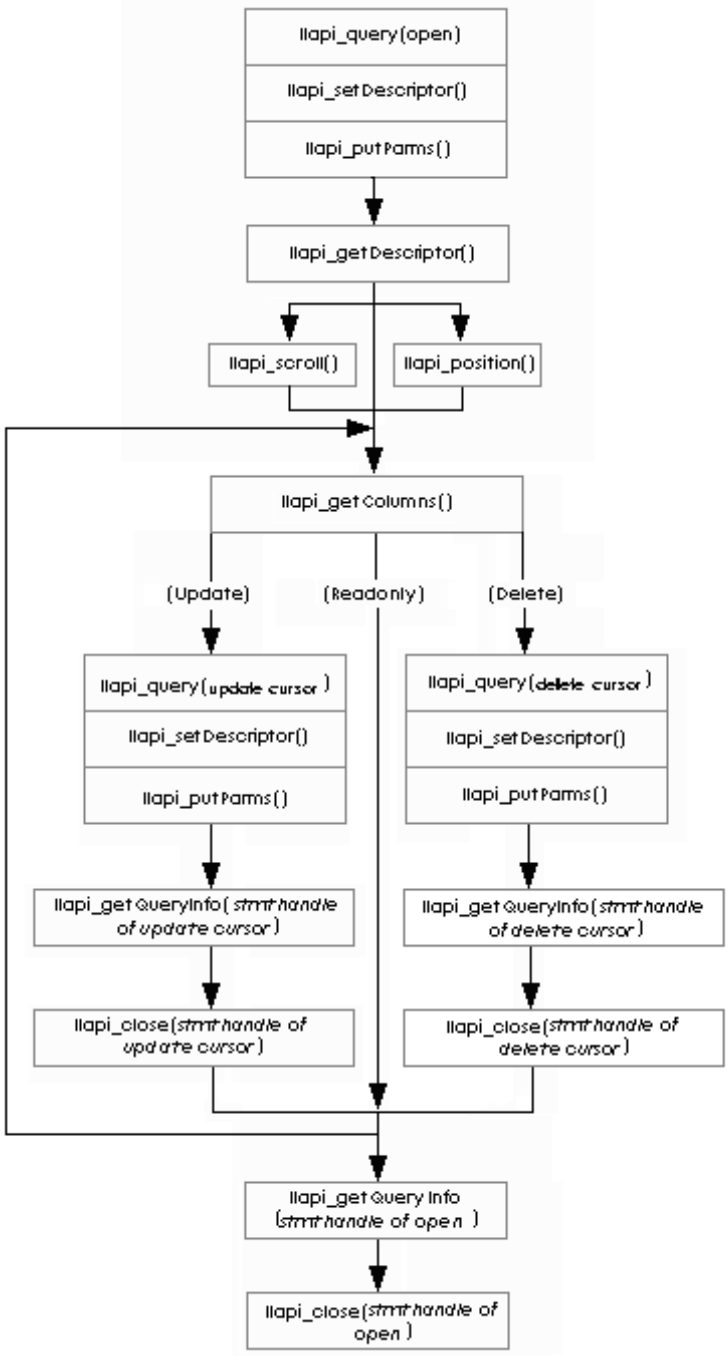
When an application calls IIapi_query() to open a cursor, it provides the name of the cursor as a parameter in subsequent calls to IIapi_setDescriptor() and IIapi_putParms(). *Cursor name* is the character string, unique within the application, that represents the cursor.

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.

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:

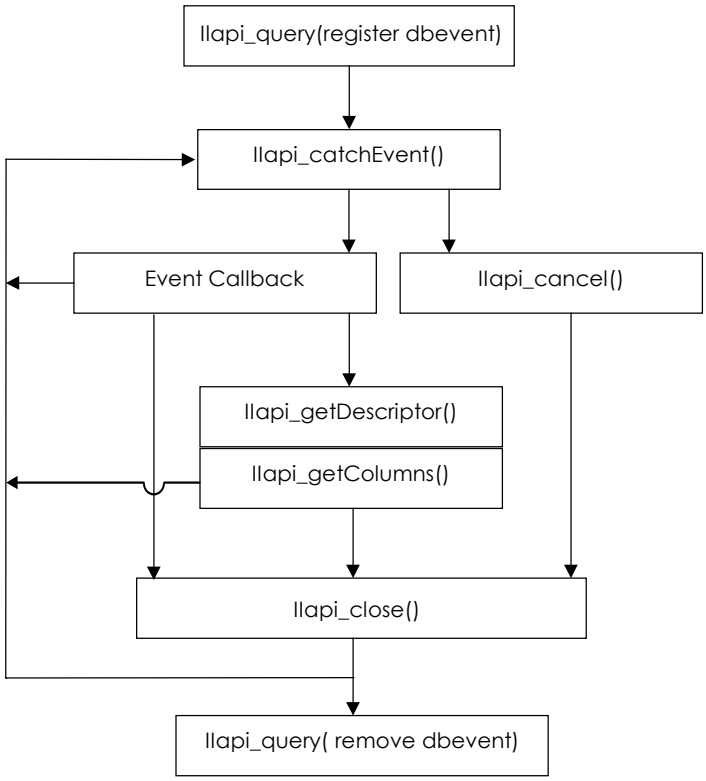
register dbevent

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 139)
SQL Syntax (see page 146)

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 153)

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 175)

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 187)
Data Type Descriptions (see page 189)
IIapi_initialize() Function (see page 73)
IIapi_connect() Function (see page 52)

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 66)

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 179)
SQLSTATE Values and Descriptions (see page 182)

Chapter 2: OpenAPI Function Reference

This chapter provides details about each OpenAPI function. It begins with a description of the generic parameters that are common to each OpenAPI function. It then lists the output parameters for which the OpenAPI allocates and manages memory. It also describes each OpenAPI function and provides its syntax and a description of its parameters.

Note: For a description of non-standard C data types used in OpenAPI parameter blocks, see OpenAPI Data Types (see page 121).

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_GENPARM
{
      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_GENPARM;
```

The generic parameters 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.

How Memory Is Managed for Data Input and Output

An application allocates memory for each OpenAPI function argument and its input parameters. The application also allocates memory for the output parameters, with the exception of the parameters listed in the following table. These parameters 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_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_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_ABORTPARM *abortParm );

typedef struct _IIAPI_ABORTPARM
{
      IIAPI_GENPARM   ab_genParm;
      II_PTR          ab_connHandle;
} IIAPI_ABORTPARM;
```

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 35).

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_GENPARM  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 35).

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_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 IIapi_cancel (IIAPI_CANCELPARM *cancelParm);

typedef struct _IIAPI_CANCELPARM
{
      IIAPI_GENPARM   cn_genParm;
      II_PTR          cn_stmtHandle;
} IIAPI_CANCELPARM;
```

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 35).

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 35).

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 How Memory is Managed for Data Input and Output (see page 37).

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 How Memory is Managed for Data Input and Output (see page 37).

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 How Memory Is Managed for Data Input and Output (see page 37).

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 How Memory Is Managed for Data Input and Output (see page 37).

ce_eventInfoAvail

Type: delayed output

Indicates whether the 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() 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 How Memory is Managed for Data Input and Output (see page 37).

For each IIapi_query() 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_CLOSEPARM *closeParm);

typedef struct _IIAPI_CLOSEPARM
{
      IIAPI_GENPARM cl_genParm;
      II_PTR        cl_stmtHandle;
} IIAPI_CLOSEPARM;
```

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 35).

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 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_COMMITPARM *commitParm);

typedef struct _IIAPI_COMMITPARM
{
      IIAPI_GENPARM  cm_genParm;
      II_PTR         cm_tranHandle;
} IIAPI_COMMITPARM;
```

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 35).

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() 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 35).

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.

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:

Data Type	Converted To
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 35).

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 56).

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 35).

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_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_GENPARM    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 35).

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_GETDESCRPARM *getDescrParm);

typedef struct _IIAPI_GETDESCRPARM
{
      IIAPI_GENPARM       gd_genParm;
      II_PTR              gd_stmtHandle;
      II_LONG             gd_descriptorCount;
      IIAPI_DESCRIPTOR   *gd_descriptor;
} IIAPI_GETDESCRPARM;
```

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 35).

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 How Memory Is Managed for Data Input and Output (see page 37).

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

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.

ge_message

Type: immediate output

Provides the text of the message.

The OpenAPI manages the memory for this parameter. For more information, see How Memory is Managed for Data Input and Output (see page 37).

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 35).

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.

This function has the following syntax:

```
II_VOID IIapi_getQueryInfo (IIAPI_GETQINFOPARM *getQInfoParm);

typedef struct _IIAPI_GETQINFOPARM
{
      IIAPI_GENPARM  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 35).

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_repeatQueryHande

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

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);

typdef 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 35).

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_POSPARM *posParm );
typedef struct _IIAPI_POSPARM
{
        IIAPI_GENPARM           po_genParm;
        II_PTR                  po_stmtHandle;
        II_UINT2                po_reference;
        II_INT4                 po_offset;
        II_INT2                 po_rowCount;
} IIAPI_POSPARM;
```

This function has the following parameters:

po_genParm

Type: input and delayed output

Specifies the generic parameters (see page 35).

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 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() or IIapi_connect() to create or restart a distributed transaction. The application used the transaction handle returned by IIapi_query() 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_PREPCMTPARM *prepCommitParm);

typedef struct _IIAPI_PREPCMTPARM
{
      IIAPI_GENPARM    pr_genParm;
      II_PTR           pr_tranHandle;
} IIAPI_PREPCMTPARM;
```

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 35).

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 provided by IIapi_query() 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 35).

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 35).

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 139).

When entering the syntax of SQL statements in the qy_queryText parameter, see SQL Syntax (see page 146) 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_queryText;
       II_CHAR             *qy_parameters;
       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 35).

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 139) 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 148).

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_RELENVPARM *relEnvParm);

typedef struct _IIAPI_RELENVPARM
{
       II_PTR             re_envHandle:
       IIAPI_STATUS       re_status;
} IIAPI_RELENVPARM;
```

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 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_ROLLBACKPARM *rollbackParm);

typedef struct _IIAPI_ROLLBACKPARM
{
       IIAPI_GENPARM   rb_genParm;
       II_PTR          rb_tranHandle;
       II_PTR          rb_savePointHandle;
} IIAPI_ROLLBACKPARM;
```

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 35).

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() 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 35).

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() 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_SCROLLPARM *scrollParm );

typedef struct _IIAPI_SCROLLPARM
{
        IIAPI_GENPARM           sl_genParm;
        II_PTR                  sl_stmtHandle;
        II_UINT2                sl_orientation;
        II_INT4                 sl_offset;
} IIAPI_SCROLLPARM;
```

This function has the following parameters:

sl_genParm

Type: input and delayed output

Specifies the generic parameters (see page 35).

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 35).

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_SETDESCRPARM *setDescrParm);

typedef struct _IIAPI_SETDESCRPARM
{
       IIAPI_GENPARM       sd_genParm;
       II_PTR              sd_stmtHandle;
       II_LONG             sd_descriptorCount;
       IIAPI_DESCRIPTOR    *sd_descriptor;
} IIAPI_SETDESCRPARM;
```

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 35).

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_EVENT_FUNC (see page 110).

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 110).

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 107).

More information:

Environment Parameter IIAPI_EP_CAN_PROMPT (see page 107)
Environment Parameter IIAPI_EP_TRACE_FUNC (see page 109)
Environment Parameter IIAPI_EP_EVENT_FUNC (see page 110)

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_PROMPTPARM
{
       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_PROMPTPARM;
```

The syntax of the application function is as follows:

```
II_VOID promptFunc(IIAPI_PROMPTPARM *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_PROMPTPARM 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_TRACEPARM
{
       II_PTR       tr_envHandle;
       II_PTR       tr_connHandle;
       II_INT4      tr_length;
       II_CHAR      *tr_message;
} IIAPI_TRACEPARM;
```

The syntax of the application function is as follows:

```
II_VOID traceFunc(IIAPI_PROMPTPARM *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_TRACEPARM 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.

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 occured, 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_WAITPARM *waitParm);

typedef struct _IIAPI_WAITPARM
{
       II_LONG        wt_timeout;
       II_API_STATUS  wt_status;
} IIAPI_WAITPARM;
```

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 35).

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 handled 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_XAENDPARM *endParm );

typedef struct _IIAPI_XAENDPARM
{
        IIAPI_GENPARM           xe_genParm;
        II_PTR                  xe_connHandle;
        IIAPI_TRAN_ID           xe_tranID;
        II_ULONG                xe_flags;
} IIAPI_XAENDPARM;
```

This function has the following parameters:

xe_genParm

Type: input and delayed output

Specifies the generic parameters (see page 35).

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_GENPARM           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 35).

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_XAROLLPARM *rollbackParm );

typedef struct _IIAPI_XAROLLPARM
{
        IIAPI_GENPARM           xr_genParm;
        II_PTR                  xr_connHandle;
        IIAPI_TRAN_ID           xr_tranID;
        II_ULONG                xr_flags;
} IIAPI_XAROLLPARM;
```

This function has the following parameters:

xr_genParm

Type: input and delayed output

Specifies the generic parameters (see page 35).

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() 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_XASTARTPARM *startParm );

typedef struct _IIAPI_XASTARTPARM
{
        IIAPI_GENPARM           xs_genParm;
        II_PTR                  xs_connHandle;
        IIAPI_TRAN_ID           xs_tranID;
        II_ULONG                xs_flags;
        II_PTR                  xs_tranHandle;
} IIAPI_XASTARTPARM;
```

This function has the following parameters:

xs_genParm

Type: input and delayed output

Specifies the generic parameters (see page 35).

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 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 187).

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 How Memory is Managed for Data Input and Output (see page 37).

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 How Memory is Managed for Data Input and Output (see page 37).

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 How Memory is Managed for Data Input and Output (see page 37).

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 How Memory is Managed for Data Input and Output (see page 37).

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
{
       II_BOOL         dv_null;
       II_UINT2        dv_length;
       II_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       de_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_PROCBYREFPARM
- IIAPI_COL_PROCPARM
- IIAPI_COL_SVCPARM
- IIAPI_COL_QPARM
- IIAPI_COL_PROCINPARM

Note: IIAPI_COL_PROCINPARM is a synonym for IIAPI_COL_PROCPARM.

- IIAPI_COL_PROCOUTPARM
- IIAPI_COL_PROCINOUTPARM

  Note: IIAPI_COL_PROCINOUTPARM is a synonym for IIAPI_COL_PROCBYREFPARM.

For a description of how IIapi_query() uses column types to describe its parameters, see Query Data Correlation (see page 150). 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_FDATADESCR 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_FDATADESCR
{
       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_FDATADESCR;
```

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 How Memory is Managed for Data Input and Output (see page 37).

svr_parmValue

Specifies the value of the message parameters. The memory for this parameter is managed by the OpenAPI.

Note: For more information, see How Memory is Managed for Data Input and Output (see page 37).

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 chapter provides information on how to use SQL statements with OpenAPI to access a DBMS. It also details the syntax of those SQL statements for which the syntax differs from that of embedded SQL. Lastly, it instructs how to enter SQL statement parameters using IIapi_query(), IIapi_setDescriptor(), and IIapi_putParms().

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 148). For the syntax of SQL statements, see SQL Syntax (see page 146) 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
<general SQL>	IIapi_query()	Invokes the SQL statement	IIAPI_QT_QUERY
	[IIapi_setDescriptor()]	Optional. Sends the information about the format of data to be sent with subsequent IIapi_putParms() calls	
	[IIapi_putParms()]	Optional. Sends data value for SQL statement parameters	
	IIapi_getQueryInfo()	Retrieves query results	
	IIapi_close()	Frees resources associated with the SQL statement	
close	IIapi_close()	Closes a cursor along with the statement handle	

SQL Statement	OpenAPI Function	Description	Query Type
commit	IIapi_commit()	Commits a transaction	
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

SQL Statement	OpenAPI Function	Description	Query Type
	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 repeat query ID	
	IIapi_close()	Frees resources associated with the SQL statement	
	IIapi_query()	Executes a repeated delete	Subsequent repeats: IIAPI_QT_EXEC_ 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 row count and query results	
	IIapi_close()	Frees resources associated with the SQL statement	
describe	IIapi_query()	Describes a previously prepared statement	IIAPI_QT_QUERY
	IIapi_getDescriptor()	Retrieves descriptors if a prepared statement is a select statement	
	IIapi_getQueryInfo()	Retrieves query results	
	IIapi_close()	Frees resources associated with the SQL statement	
disconnect	IIapi_disconnect()	Disconnects from a data source	
execute	IIapi_query()	Executes a previously-prepared statement	IIAPI_QT_EXEC
	[IIapi_setDescriptor()]	Optional. Sends information about the format of data to be sent with subsequent IIapi_putParms() calls	

SQL Statement	OpenAPI Function	Description	Query Type
	[IIapi_putParms()]	Optional. Sends data values for SQL statement parameters	
	IIapi_getQueryInfo()	Retrieves query results	
	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	

SQL Statement	OpenAPI Function	Description	Query Type
	IIapi_getQueryInfo()	Retrieves a repeat query ID	
	IIapi_close()	Frees resources associated with the SQL statement	
	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 sqlda" is used in prepare query text	
	IIapi_getQueryInfo()	Retrieves query results	
	IIapi_close()	Frees resources associated with the SQL statement	

SQL Statement	OpenAPI Function	Description	Query Type
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	
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

SQL Statement	OpenAPI Function	Description	Query Type
	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 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	

SQL Statement	OpenAPI Function	Description	Query Type
	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() 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	

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 173) 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_DEFINE_ 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 146).

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

Query Parameters	Descriptor Column Type	Data Type
Procedure parameters	IIAPI_COL_PROCPARM or IIAPI_COL_ PROCBYREFPARM IIAPI_COL_PROCINPARM IIAPI_COL_PROCOUTPARM IIAPI_COL_PROCINOUTPARM	Any of the Ingres data types in Ingres Data Types the "Data Types" appendix
Parameter values	IIAPI_COL_QPARM	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_QPARM	Any of the Ingres data types in Ingres Data Types the "Data Types" appendix
Dynamic parameter values	IIAPI_COL_QPARM	Any of the Ingres data types in Ingres Data Types the "Data Types" appendix

Chapter 5: Accessing the Name Server

This chapter provides the syntax and descriptions for using the Name Server query statements supported by OpenAPI.

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:

- wintcp
- lanman
- nvlspx
- decent
- tcp_ip

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:

- wintcp
- lanman
- nvlspx
- decent
- tcp_ip

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:

- wintcp
- lanman
- nvlspx
- decent
- tcp_ip

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 chapter describes the requirements for creating an OpenAPI application. It also describes how to use the synchronous and asynchronous sample code available with OpenAPI.

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_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.

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 are 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 sources 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.

apissell.c

Demonstrates retrieving data using a select loop and canceling a query.

apisselc.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 sources 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
apiademo -c dbname	apiademo -c dbname 5
apiademo -t dbname	apiademo -c -t db 0

Source File Descriptions

The asynchronous sample source files are as follows:

apiademo.c

Demonstrates main processing control.

apiaclnt.c

Demonstrates client execution unit.

apiasvr.c

Demonstrates server execution unit.

apiautil.c

Demonstrates utility functions.

Chapter 7: Using Repeat Queries with OpenAPI

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 174) 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 177)

Repeat Query Parameters

The query parameters (see page 175) 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 175). These extended parameter markers take the form:

$*n* = ~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 176)

Appendix A: Error Handling

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_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
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

SQLSTATE	Value	Description
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
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

SQLSTATE	Value	Description
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_PARMS	07004	Using clause required: params
II_SS07005_STMT_NOT_CURS_SPEC	07005	Prepared statement not cursor specification
II_SS07006_RESTR_DT_ATTR_ERR	07006	Restricted data type violation
II_SS07007_NEED_USING_FOR_RES	07007	Using clause required: result
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

SQLSTATE	Value	Description
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
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

SQLSTATE	Value	Description
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

Appendix B: Data Types

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:

Data Type	Length	Precision/ Scale	C Type	SQL Type
IIAPI_BYTE_TYPE	*n*	0/0	char*	byte(*n*)
IIAPI_CHA_TYPE	*n*	0/0	char*	char(*n*)
IIAPI_CHR_TYPE	*n*	0/0	char*	c(*n*)

Data Type	Length	Precision/ Scale	C Type	SQL Type
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)	*p*/*s*	none	decimal(*p*,*s*)
IIAPI_INTDS_TYPE	12	*p*/0	none	interval day to second
IIAPI_DTE_TYPE	12	0/0	none	ingresdate
IIAPI_FLT_TYPE	4 8	*p*/0	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	*p*/0	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	*n*	0/0	char*	null
IIAPI_MNY_TYPE	8	*p*/0	none	money
IIAPI_NCHA_TYPE	*n*	0/0	short*	nchar(n)
IIAPI_NVCH_TYPE	*n*	0/0	short*	nvarchar(n)
IIAPI_TABKEY_TYPE	8	0/0	char*	table_key
IIAPI_TIME_TYPE	10	*p*/0	none	time with local time zone
IIAPI_TMWO_TYPE	10	*p*/0	none	time without time zone
IIAPI_TMTZ_TYPE	10	*p*/0	none	time with time zone
IIAPI_TSWO_TYPE	14	*p*/0	none	timestamp without time zone
IIAPI_TSTZ_TYPE	14	*p*/0	none	timestamp with time zone

Data Type	Length	Precision/ Scale	C Type	SQL Type
IIAPI_TS_TYPE	14	*p*/0	none	timestamp with local time zone
IIAPI_TXT_TYPE	*n*	0/0	char*	text(*n*)
IIAPI_VBYTE_TYPE	*n*	0/0	char*	varbyte(*n*)
IIAPI_VCH_TYPE	*n*	0/0	char*	varchar(*n*)

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_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_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

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W