# Polyhedra®

## Call-back API

Enea Polyhedra Ltd

ENEA

## Copyright notice



## Disclaimer



## Trademarks



## Manual Details

| | |
|---|---|
| **Manual title** | **Call-back API** |
| **Document number** | **manuals/clntapi** |
| **Revision number** | **9.4.0** |
| **Revision Date** | **7th September 2018** |

## Software Version

This documentation corresponds to the following version of the software:

| | |
|---|---|
| **Version** | **9.4** |

# Document conventions

In this document some special conventions are followed to provide information in a concise manner.

| | |
|---|---|
| **<x>** | **x** to be further defined |
| **[ x ]** | Optional **x** |
| **{ x }** | Zero or more **x** |
| **x \| y** | **x** or **y** |
| **font** | A special bold font to indicate code |
| `font` | A special courier font to indicate keywords and built-in functions, classes and resources |

# CONTENTS

# 1. Introduction

This manual describes the Polyhedra Call-back Application Programmers Interface (Call-back API). The Polyhedra Call-back API contains a set of functions for implementing a Polyhedra client. It does this in a manner compatible with the non-blocking nature of Polyhedra and is fully integrated with the underlying scheduling mechanism employed by Polyhedra. Anyone implementing a Polyhedra client coded in C or C++ will need to use this interface or the ODBC API, which is separately documented. The interface consists of:

- A set of six header files, **appapi.h**, **clntapi.h**, **commandapi.h**, **queryapi.h**, **transapi.h** and **timerapi.h,** declaring the functions in the API.
- A set of library files containing the code implementing the functions in the API.

Each header file relates to a particular area of functionality. The header files entirely define the API. No further Polyhedra header files are required. The API is defined using standard C data types. It does not use any Polyhedra defined data types. This is done deliberately to avoid any conflicts that could occur between standard C data type and macro definitions, third party software data type and macro definitions and Polyhedra data type and macro definitions.
The library files are platform dependent. There is a different set of library files for each platform on which the API is supported.
The API supports the following facilities:

- Scheduler initialisation, starting and stopping.
- Multiple timers.
- Separate preparation and execution of SQL.
- Data retrieval using SQL.
- Data retrieval using object queries.
- Data retrieval using SQL procedure queries.
- Data modification using SQL.
- Active SQL queries.
- Active object queries.
- Active SQL procedure queries.
- Data modification via active queries
- Terminating a server

The Polyhedra Call-back API is supported on all platforms on which Polyhedra is released.

Users of the Call-back API may also be interested in the Socket API, which is documented in a separate reference manual. Amongst other features, the Socket API allows the Call-back API scheduler to monitor extra sockets, and thus provides a way for other threads or external events to interrupt the scheduler.

# 2. Using the API

This section describes how to use the API. It covers the following topics:

- Scheduler initialisation and start-up.
- Stopping the scheduler.
- Timers.
- Opening a client connection.
- Logging onto a server.
- Performing queries.
- Accessing row data.
- Performing active queries.
- Aborting active queries.
- Performing transactions.
- Logging out of a server.
- Closing a client connection.
- Terminating a server.

## 2.1 Initialising the Scheduler

The heart of a Polyhedra client is the Polyhedra scheduler. This manages the scheduling of I/O, timer events and internal messages. For a Polyhedra client to function correctly the scheduler must be running. The Call-back API provides functions to initialise and start-up the scheduler. These functions must be called before any other facilities provided by the API can be used. The **AppAPI::Init** function is provided by the API for scheduler initialisation. It has the prototype:

**static int Init();**

The equivalent C-callable function is:

**int AppAPI_Init (void);**

This function performs all necessary initialisation of the scheduler. It returns zero if initialisation is successful and non-zero if not. This must be the first API function to be called by your application and it must only be called once.

Once the scheduler has been successfully initialised, client application instances can then be created. The **AppAPI::Create** function is provided by the API for creating client application instances. It has the prototype:

**static AppAPI *Create();**

The equivalent C-callable function is:

**AppAPI *AppAPI_Create(void);**

In a multi-threaded client each thread must create its own instance of **AppAPI** using **AppAPI::Create** and must not share objects created from the **AppAPI** object with other threads. Thus each thread will operate on entirely separate objects**.**

Once the scheduler has been successfully initialised and created, it can then be started. When the scheduler is executing the only user-defined code that will be executed is that which is defined in callback functions. Essentially, the scheduler contains the *main loop* and calls user-defined functions (call-backs) that perform whatever user code is required and then return control back to the scheduler. The user-defined callback functions can of course include calls to the Call-back API.
The **AppAPI::Start** function is provided by the API for starting-up the scheduler. It has the prototype:

**static int Start(**

**AppAPI *app,**

**const int (*userFun)(void *),**

**const void*userdata);**

The equivalent C-callable function is:

**int AppAPI_Start(**

**AppAPI *app,**

**const int (*userFun)(void *),**

**const void*userdata);**

This function invokes the scheduler passing control to the scheduler's *main loop*. This function will only return when the scheduler has nothing more to do, the **AppAPI::Stop** function is called or starting-up the scheduler fails. Note that an open connection or a timer will occupy the scheduler and control will not return from the **AppAPI::Start** function. Similarly, if there is a callback function defined, either because the callback function passed into the **AppAPI::Start** function returned a non-zero value or if **AppAPI::SetFun** has been called, then control will not return from the **AppAPI::Start** function.
The application instance is specified by the *app* parameter. A user-defined callback function is specified by the *userFun* parameter. This is the first function that the scheduler calls. The *userdata* parameter is a pointer to an object or piece of memory. This pointer is passed unaltered to the callback function as its only parameter. The API does not check the validity of the pointer. If the pointer references an object or dynamically allocated piece of memory, this must not be deleted or freed before the callback function has been invoked.

This user-defined function is the first callback function called by the scheduler. The following fragment of code initialises, creates and starts-up the scheduler:

```
#include "appapi.h"

class MyApp
{
    AppAPI *App;
public:
    MyApp();
    int StartScheduler();
    static const int StartFun(void *userdata);
};

int main()
{
    if (AppAPI::Init()) exit(0);

    MyApp *app = new MyApp;
```

```
    return app->StartScheduler();
}

MyApp::MyApp()
{
    App = AppAPI::Create();

    if (App == NULL) exit(1);
}

int MyApp::StartScheduler()
{
    return AppAPI::Start(App, &MyApp::StartFun, this);
}

const int MyApp::StartFun(void *userdata)
{
    MyApp *app = (MyApp *)userdata;
    .
    .
    .
    return 0;
}
```

If the user-defined function returns zero, the scheduler will not call it again. If the function returns a non-zero value, the scheduler will call the function again as part of the normal scheduling process. Effectively a single user-defined callback function is registered with the scheduler. The scheduler repeatedly calls this until the function returns zero. The user-defined function registered with the scheduler is initially set by the **AppAPI::Start** function. Subsequently a different function can be registered. The **AppAPI::SetFun** function is provided by the API for registering a different user-defined callback function with the scheduler. It has the prototype:

**static int SetFun(**

**AppAPI *app,**

**const int (*userFun)(void *),**

**const void*userdata);**

The equivalent C-callable function is:

**int AppAPI_SetFun(**

**AppAPI *app,**

**const int (*userFun)(void *),**

**const void*userdata);**

The application instance is specified by the *app* parameter. The new function is specified by the *userFun* parameter and the user-defined pointer passed to it specified by the *userdata* parameter. The **SetFun** function returns zero if the function is registered successfully and non-zero if it not. Only one user-defined function can be registered with the scheduler at any give time and so calling **SetFun** will overwrite the existing registered function.

Again, the user-defined function returns non-zero to inform the scheduler that it needs to be called again, or zero if it does not. More precisely, it informs the scheduler that the currently registered function should be called. The following fragment of code illustrates the use of the **SetFun** function:

```
#include "appapi.h"

class MyApp
{
    AppAPI *App;
public:
    MyApp();
    int StartScheduler();
    static const int Fun1(void *userdata);
    static const int Fun2(void *userdata);
};

int main()
{
    if (AppAPI::Init()) exit(0);

    MyApp *app = new MyApp;

    return app->StartScheduler();
}

MyApp::MyApp()
{
    App = AppAPI::Create();

    if (App == NULL) exit(1);
}

int MyApp::StartScheduler()
{
    return AppAPI::Start(App, &MyApp::Fun1, this);
}

const int MyApp::Fun1(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    AppAPI::SetFun(&MyApp::Fun2, app);

    return 1;
}

const int MyApp::Fun2(void *userdata)
{
    MyApp *app = (MyApp *)userdata;
    .
    .
    .
    return 0;
}
```

## 2.2 Stopping the Scheduler

The Polyhedra scheduler can be stopped. The **AppAPI::Stop** function is provided by the API for stopping the scheduler. It has the prototype:

**static int Stop(**

**AppAPI *app);**

The equivalent C-callable function is:

**int AppAPI_Stop(**

**AppAPI *app);**

This function can be called from within any callback function. When control is returned to the scheduler's *main loop* after a call to **Stop()**, the *main loop* is immediately terminated. Control then returns to the users code immediately following the call to **Start()** which initiated the scheduler. It returns zero if successful and non-zero if not.

Once stopped the scheduler can be restarted by another call to **Start()**, in which case the scheduler continues with any work which it was doing when **Stop()** was called (e.g. timers). Calling **Stop()** when the scheduler is not running will invoke an error.

## 2.3 Timers

The Call-back API provides support for timers. These are a means of registering a callback function with the scheduler that is called after a specific timer period has elapsed. The function **TimerAPI::CreateOneShotTimer** is provided by the API for registering callback functions invoked after a timed interval. It has the prototype:

**static TimerAPI *CreateOneShotTimer(**

**AppAPI *app,**

**long dsecs,**

**const void (*timerFun)(void *),**

**const void *userdata);**

The equivalent C-callable function is:

**TimerAPI *TimerAPI_CreateOneShotTimer(**

**AppAPI *app,**

**long dsecs,**

**const void (*timerFun)(void *),**

**const void *userdata);**

The application instance is specified by the *app* parameter. The callback function is specified by the *timerFun* parameter and user-defined pointer passed to it is specified by the *userdata* parameter. The *dsecs* parameter specifies the number of tenths of a second that must elapse before the function is invoked.
The function returns a pointer to an instance of the **TimerAPI** class. This can subsequently be used for stopping the timer or obtaining the time it has left to run before it activates.
Multiple timers can be registered at one time. The scheduler manages invoking the correct callback function at the correct time.
The following fragment of code creates a timer that has a duration of ten seconds:

```
#include "appapi.h"
#include "timerapi.h"

class MyApp
{
    AppAPI *App;
    TimerAPI *Timer;
public:
    void CreateTimer();
    static const void TimerFun(void *userdata);
};

void MyApp::CreateTimer()
{
    Timer = TimerAPI::CreateOneShotTimer(App, 100, &MyApp::TimerFun,
this);
}

const void MyApp::TimerFun(void *userdata)
{
    MyApp *app = (MyApp *)userdata;
    .
    .
    .
}
```

When the callback function terminates and control returns to the scheduler, the timer object is deleted.
The **TimerAPI::StopTimer** function is provided by the API for stopping timers. It has the prototype:

**static void StopTimer (**

**TimerAPI *timer);**

The equivalent C-callable function is:

**void TimerAPI_StopTimer(**

**TimerAPI *timer);**

The *timer* parameter specifies the particular timer to be stopped. Calling **StopTimer** will delete the timer object referred to in the second argument of both prototypes and will mean the callback function for the timer is never called. Calling **StopTimer** from within the actual callback function for the timer has no effect.

The **TimerAPI::ReadTimer** function is provided by the API for obtaining the number of tenths of a second remaining on a timer before it activates. It has the prototype:

**static long ReadTimer(**

**const TimerAPI *timer);**

The equivalent C-callable function is:

**long TimerAPI_ReadTimer(**

**const TimerAPI *timer);**

The *timer* parameter specifies the timer for which the request is made. This function can be called any number of times on a timer object. It does not affect the operation of the timer.

## 2.4 Opening a Client Connection

A client connection to the database provides a means of performing queries and transactions upon the database. An initial connection is established by specifying the name of the service on which the database is listening for connections. All queries and transactions are subsequently performed through this connection. When all the required operations have been performed on the database, the connection is closed.
The function **ClientAPI::StartConnect** is provided by the API for establishing a client connection. It has the prototype:

**static ClientAPI *StartConnect(**

**AppAPI *app,**

**const char *serverName,**

**const void (*connectedCallBack)(void *),**

**const void *userdata,**

**const void (*disconnectedCallBack)(void *) = NULL,**

**const void *disconnectedUserData = NULL**

**const void (*modeChangeCallBack)(void *) = NULL,**

**const void *modeChangeUserData = NULL);**

The equivalent C-callable function is:

**ClientAPI *ClientAPI_StartConnect(**

**AppAPI *app,**

**const char *serverName,**

**const void (*connectedCallBack)(void *),**

**const void *userdata,**

**const void (*disconnectedCallBack)(void *),**

**const void *disconnectedUserData**

**const void (*modeChangeCallBack)(void *),**

**const void *modeChangeUserData);**

The application instance is specified by the *app* parameter. The host and port number are specified by the **serverName** parameter. This is a string containing the IP address of the host and the port number separated by a colon. For example, to connect to the port *5000* on the host with address *1.2.3.4*, **serverName** should contain the string "*1.2.3.4:5000*".
The function is non-blocking. That is, it returns immediately rather than waiting for the connection to be attempted (via underlying communications between the two machines). Instead, notification of completion of the operation, whether successful or unsuccessful, is done by invoking a callback function. This is a general mechanism employed by the API.
A callback function is a static function with a prototype matching that specified for the particular operation it is used for, the address of which is passed as a parameter to the function initiating the operation. The callback function is called when the operation completes.
The callback function for the connect operation is specified by the *connectedCallBack* parameter. The *userdata* parameter specifies a pointer to an object or piece of memory to be specified. This is termed user-defined data. This pointer will be passed unaltered to the callback function as its only parameter.
The API does not check the validity of this pointer. If the pointer references a dynamically allocated object or piece of memory, this must not be deleted or freed before the callback function has been invoked.
The *connectedCallback* callback function is called whether or not the operation succeeds. The status of the operation can be determined by using the **ClientAPI::GetError** API function. This retrieves the error code associated with the last completed operation. A zero value returned indicates the operation has succeeded. A non-zero value indicates the operation has failed.

**Note:** If there are multiple operations outstanding on a single socket, the error must be retrieved from inside the callback function. This is because returning from the callback function may be followed by the processing of another operation completion and subsequent overwriting of the error value before you have chance to access the error value for the first operation.

The *disconnectedCallBack* and *disconnectedUserdata* parameters are optional. They specify a callback function and a pointer to user-data to be passed to the function. The function is invoked whenever the connection to the server is unexpectedly lost after previously being successfully connected. This callback function is not invoked when the connection to the server is explicitly closed by the client, (see section 2.11).
The *modeChangeCallBack* and *modeChangeUserdata*are optional. The specify a callback function and a pointer to user-data to be passed to the function. The function is invoked whenever the fault tolerant mode of the connection of the server changes.

The following fragment of code attempts to establish a connection to a database listening to the port *5000* on the host specified by the IP address *1.2.3.4*:

```
#include "appapi.h"
#include "clntapi.h"

class MyApp
{
    AppAPI *App;
    ClientAPI *Client;
    static const void ConnectCallback(void *userdata);
    static const void DisconnectCallback(void *userdata);
public:
    void StartConnect();
};

void MyApp::StartConnect()
{
    // Initiate the connect operation
    Client = ClientAPI::StartConnect(App,
                                     "1.2.3.4:5000",
                                     &MyApp::ConnectCallback,
                                     this,
                                     &MyApp::DisconnectCallback,
                                     this);
}

const void MyApp::ConnectCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    if (ClientAPI::GetError(app->Client))
    {
        // The connection attempt failed.
    }
}

const void MyApp::DisconnectCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // The connection has been lost.
    // Use ClientAPI::GetError(app->Client) to obtain the error
}
```

Note the technique of passing a pointer to the instance of the class invoking the **StartConnect** call and containing a member variable holding a pointer to the create instance of **ClientAPI**. This is done because the callback function must be static. It therefore has no valid *this* pointer. The only way to access the created client object (unless it stored in a global variable) is to pass the pointer to the instance storing the pointer to the socket into the callback .
The pointer to the instance of **ClientAPI** returned by **StartConnect** can subsequently be used for performing queries and transactions and will eventually be closed once the callback function has been invoked and no error has occurred.

## 2.5 Logging onto a Server

If security is enabled in the server, it may necessary to log-on as a particular user to obtain sufficient privileges to perform subsequent operations. Logging on to the server involves supplying a user name and password for authentication by the server. If the server authenticates the user name as specifying a valid user registered in the system, and the password matches that specified for the user, the client is then successfully logged on as that user. All subsequent operations performed on the same connection are done as that user.
The function **ClientAPI::StartLogin** is provided by the API for logging onto a server. It has the prototype:

```
static void StartLogin(
                    ClientAPI *client,
                    const char *username,
                    const char *password,
                    const void (*loginCallBack)(void *),
                    const void *userdata,
                    const char *env = 0);
```

The equivalent C-callable function is:

```
void ClientAPI_StartLogin(
                    ClientAPI *client,
                    const char *username,
                    const char *password,
                    const void (*loginCallBack)(void *),
                    const void *userdata,
                    const char *env);
```

The function authenticates the user name and password, specified by *username* and *password* respectively, through the client connection specified by the *client* parameter.
This function is non-blocking. It returns when the authentication request has been initiated. The response from the server is communicated to the application via the callback function *loginCallBack*. The *userdata* parameter specifies a user-defined pointer to be passed into the *loginCallBack* callback function. For details of the optional *env* parameter see the description of this function in the API Reference section.

The following code fragment attempts to log onto the server as the user "SYSTEM" with the password "SCOTT":

```
#include "appapi.h"
#include "clntapi.h"

class MyApp
{
    ClientAPI *Client;
    static const void LoginCallback(void *userdata);
public:
    void StartLogin();
};

void MyApp::StartLogin()
{
    // Initiate the login operation
    ClientAPI::StartLogin(Client,
                          "SYSTEM",
                          "SCOTT",
                          &MyApp::LoginCallback,
                          this);
}

const void MyApp::LoginCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    if (ClientAPI::GetError(app->Client))
    {
        // The log-in attempt failed.
    }
    else
    {
        // The log-in attempt succeeded.
    }
}
```

## 2.6 Performing Queries

The purpose of establishing a client connection is to retrieve and modify data held in the database. This is achieved using queries and transactions. There are three ways to specify a query in Polyhedra. The first uses an SQL select statement to define the query. The second uses a specialised form of query for retrieving a single row from a table (or view) called an object query, and the third uses a SQL procedure already defined in the server. The Call-back API supports all forms of query. The API function **QueryAPI::StartQuery** is provided for executing SQL queries. It has the prototype:

```
static QueryAPI *StartQuery(
                    const ClientAPI *client,
                    const char *sqlText,
                    const void (*gotRow)(void *),
                    const void (*queryComplete)(void *),
                    const void *userdata,
                    const unsigned int maxRows = POLY_MAX_ROWS_UNLIMITED /*implies
                    no row limitation */);
```

The equivalent C-callable function is:

```
void QueryAPI*QueryAPI_StartQuery(
                    const ClientAPI *client,
                    const char *sqlText,
                    const void (*gotRow)(void *),
                    const void (*queryComplete)(void *),
                    const void *userdata,
                    const unsigned int maxRows);
```

The function executes the SQL query specified by the *sqlText* parameter through the client connection specified by the *client* parameter.
This function is non-blocking. It returns a pointer to an instance of the QueryAPI class when the request has been initiated. Data retrieved by the query is communicated to the application via two callback functions, *gotRow* and *queryComplete*.
The *gotRow* callback function is invoked once for each row of data retrieved by the query. If no data is retrieved, the function is not called. The API provides functions specifically for use within this callback function for obtaining the actual data contained in the row retrieved.
When all rows have been retrieved and the associated invocation of the *gotRow* callback function completed, the callback function *queryComplete* is invoked. This is an indication that execution of the query has completed and that no more rows will be returned by the query. The *userdata* parameter specifies a user-defined pointer to be passed into both the *gotRow* and *queryComplete* callback functions.
The *maxRows* optional parameter can be used to limit the number of rows retrieved by the query. The default value is POLY_MAX_ROWS_UNLIMITED (0xFFFFFFFF or 4294967295), i.e. effectively no limitation.

The following code fragment retrieves data using the SQL query, "select name, address from person":

```
#include "appapi.h"
#include "clntapi.h"
#include "queryapi.h"

class MyApp
{
    ClientAPI *Client;
    QueryAPI *Query;
    static const void GotRowCallback(void *userdata);
    static const void QueryCompleteCallback(void *userdata);
public:
    void StartQuery();
};

void MyApp::StartQuery()
{
    // Initiate the query
    Query = QueryAPI::StartQuery(Client,
                                 "select name, address from person",
                                 &MyApp::GotRowCallback,
                                 &MyApp::QueryCompleteCallback,
                                 this);
}

const void MyApp::GotRowCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // Obtain the data for a row here
}

const void MyApp::QueryCompleteCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // No more records
}
```

The **StartQuery** function returns a pointer to an instance of the **QueryAPI** class. This can be used to retrieve the error code set by the function to indicate whether the query executed successfully. For instance, it may fail if the SQL specified was illegal.
When the *queryComplete* callback function terminates, the instance of **QueryAPI** is deleted.

The API function **QueryAPI::StartObjectQuery** is provided for executing object queries. It has the prototype:

**static QueryAPI *StartObjectQuery(**

**const ClientAPI *client,**

**const char *tableName,**

**const void (*gotRow)(void *),**

**const void (*queryComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(QueryAPI *, void *));**

The equivalent C-callable function is:

**void QueryAPI*QueryAPI_StartObjectQuery(**

**const ClientAPI *client,**

**const char *tableName,**

**const void (*gotRow)(void *),**

**const void (*queryComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(QueryAPI *, void *));**

The function executes an object query specified by the *tableName* parameter through the client connection specified by the *client* parameter. Results from the query are communicated to the client in an identical manner to that described for the **QueryAPI::StartQuery** function using the *gotRow* and *queryComplete* callback functions.
The **StartObjectQuery** function returns a pointer to an instance of the **QueryAPI** class. This can be used to retrieve the error code set by the function to indicate whether the query executed successfully. For instance, it may fail if the table name specified does not exist.
When the *queryComplete* callback function terminates, the instance of **QueryAPI** is deleted.
An object query requires the following information:

- the name of the table (or view) being queried.
- the names and values of the primary key columns in the table specifying the particular row.
- optionally, a set of column names of the table specifying the set of columns retrieved by the query. If none are specified, all columns are retrieved.

The first of these pieces of information, the table name, is passed as a parameter to the **QueryAPI::StartObjectQuery** function. The remaining information is specified by calling one of two functions provided by the API from within the *collectArgs* callback function.

The API function **QueryAPI::AddName** is provided for specifying the name of a column to be retrieved by an object query. It has the prototype:

**static int AddName(**

**const QueryAPI *query,**

**const char *columnName);**

The equivalent C-callable function is:

**int QueryAPI_AddName(**

**const QueryAPI *query,**

**const char *columnName);**

This function can only be called from within a *collectArgs* callback function. The *query* parameter specifies the query with which the argument is associated, and is identical to the first parameter passed to the *collectArgs* callback function. The column is specified by supplying its name in the *columnName* parameter. The name supplied must be all lower case.

The API function **QueryAPI::AddArg** is provided for specifying a query argument. It has the prototype:

**static int AddArg(**

**const QueryAPI *query,**

**const char *name,**

**int type,**

**const void *buffer,**

**int bufferLength);**

The equivalent C-callable function is:

**int QueryAPI_AddArg(**

**const QueryAPI *query,**

**const char *name,**

**int type,**

**const void *buffer,**

**int bufferLength);**

This function can only be called from within a *collectArgs* callback function. The *query* parameter specifies the query with which the argument is associated, and is identical to the first parameter passed to the *collectArgs* callback function. The query argument is specified by supplying its name in the *name* parameter, its type in the *type* parameter and its value in the *buffer* parameter. The length of the values is specified by the *bufferLength* parameter. For all non-object query uses, the *name* parameter may be a NULL pointer. In this case the arguments are positional, rather than named. Whereas a named parameter is referred to in the SQL text as ":<type>name" a positional parameter is simply "?". Named and positional parameters may not be mixed in SQL text or calls to *QueryAPI::AddArg* within a *collectArgs* function. Possible values for the type are given in section 5.
The following code fragment retrieves data using an object query on the person table:

```
#include "appapi.h"
#include "clntapi.h"
#include "queryapi.h"

class MyApp
{
    ClientAPI *Client;
    QueryAPI *Query;
    static const void GotRowCallback(void *userdata);
    static const void QueryCompleteCallback(void *userdata);
    static const void CollectArgsCallback(QueryAPI *, void
*userdata);
public:
    void StartObjectQuery();
};

void MyApp::StartObjectQuery()
{
    // Initiate the query
    Query = QueryAPI::StartObjectQuery(Client,
                                       "person",
                                       &MyApp::GotRowCallback,
```

```
                                        &MyApp::QueryCompleteCallback,
                                        this,
                                        &MyApp::CollectArgsCallback);
}
```

```
const void MyApp::GotRowCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // Obtain the data for a row here
}

const void MyApp::QueryCompleteCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // No more records
}

const void MyApp::CollectArgsCallback(QueryAPI *query, void
*userdata)
{
    MyApp *app = (MyApp *)userdata;

    // Define output columns
    QueryAPI::AddName(query, "name");
    QueryAPI::AddName(query, "address");

    // Define argument
    int id = 1;
    QueryAPI::AddArg(query, "id", 1, &id, sizeof(int));
}
```

The API function **QueryAPI::StartProcedureQuery** is provided for executing SQL procedure queries. It has the prototype:

**static QueryAPI *StartProcedureQuery(**

**const ClientAPI *client,**

**const char *procedureName,**

**const void (*gotRow)(void *),**

**const void (*queryComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(QueryAPI *, void *),**

**const unsigned int maxRows = POLY_MAX_ROWS_UNLIMITED /*implies no row limitation */);**

The equivalent C-callable function is:

**QueryAPI *QueryAPI_StartProcedureQuery(**

**const ClientAPI *client,**

**const char *procedureName,**

**const void (*gotRow)(void *),**

**const void (*queryComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(QueryAPI *, void *),**

**const unsigned int maxRows);**

The function executes the SQL procedure query specified by the *procedureName* parameter through the client connection specified by the *client* parameter. Results from the query are communicated to the client in an identical manner to that described for the **QueryAPI::StartQuery** function using the *gotRow* and *queryComplete* callback functions.
The **StartProcedureQuery** function returns a pointer to an instance of the **QueryAPI** class. This can be used to retrieve the error code set by the function to indicate whether the query executed successfully. For instance, it may fail if the procedure name specified does not exist.
When the *queryComplete* callback function terminates, the instance of **QueryAPI** is deleted.
The *maxRows* optional parameter can be used to limit the number of rows retrieved by the query. The default value is POLY_MAX_ROWS_UNLIMITED (0xFFFFFFFF or 4294967295), i.e. effectively no limitation.
An SQL procedure requires the following information:

- the name of the procedure being executed
- optionally, a set of named parameter values

The procedure name, is passed as a parameter to the **QueryAPI::StartProcedureQuery** function. The parameters are specified by calling the **QueryAPI::AddArg** function described earlier.
The following code fragment retrieves data using an SQL procedure query:

```
#include "appapi.h"
#include "clntapi.h"
#include "queryapi.h"

class MyApp
{
    ClientAPI *Client;
    QueryAPI *Query;
    static const void GotRowCallback(void *userdata);
    static const void QueryCompleteCallback(void *userdata);
    static const void CollectArgsCallback(QueryAPI *, void
*userdata);
public:
    void StartProcedureQuery();
};

void MyApp::StartProcedureQuery()
{
    // Initiate the query
    Query = QueryAPI::StartProcedureQuery(Client,
                                          "GetPerson",
                                          &MyApp::GotRowCallback,
                                          &MyApp::QueryCompleteCallback,
                                          this,
                                          &MyApp::CollectArgsCallback);
}

const void MyApp::GotRowCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // Obtain the data for a row here
}

const void MyApp::QueryCompleteCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // No more records
}
```

```
const void MyApp::CollectArgsCallback(QueryAPI *query, void
*userdata)
{
    MyApp *app = (MyApp *)userdata;

    // Supply positional argument
    int id = 1;
    QueryAPI::AddArg(query, NULL, 1, &id, sizeof(int));
}
```

## 2.7 Accessing Row Data

When the *gotRow* callback function is called it needs to obtain the values of the data contained in the row. It does this by calling one of the API functions for copying data. The API function **QueryAPI::CopyColumn** is provided by the API for copying data for a row into a user supplied buffer. It has the prototype:

**static int CopyColumn(**

**const QueryAPI *query,**

**const char *columnName,**

**void *variable,**

**int bytesToCopy);**

The equivalent C-callable function is:

**int QueryAPI_CopyColumn(**

**const QueryAPI *query,**

**const char *columnName,**

**void *variable,**

**int bytesToCopy);**

This function can only be called from within a *gotRow* callback function. The *query* parameters specify the query from which the row data is to be obtained. This must match the query on which the *gotRow* callback function is defined. The **CopyColumn** function returns data for the value of a single field within the row. The field required is specified by supplying its name in the *columnName* parameter. The name of a column for an SQL query is specified in the project list for the query. For instance, the SQL query:

```
select a, b from t;
```

has two columns, the first called *a* and the second called *b*.
The value of the field is copied into a user-supplied buffer specified by the *variable* parameter. The number of bytes to be copied is specified by the *bytesToCopy* parameter. This should obviously be no larger than the number of bytes available in the buffer. The size of the actual data value to be copied depends on the data type of the column. For example, an integer column will be four bytes and an integer64 column eight bytes. The user-supplied buffer should be large enough to hold the data retrieved. Some types of columns are of fixed size - that is, all the non-null values in the column are of the same size, and some columns are of variable size - that is, the size of each value may be different. Fixed sized columns can be copied into variables of specific types dependent on the data type of the column. Variable sized columns should be copied into buffers. The following table specifies the size of each column type and the C++ data type that should be used to hold the value copied:

| Data Type | Size | C++ Data Type |
|---|---|---|
| Binary | Variable | unsigned char[] |
| Boolean | 4 | int |
| DateTime | 8 | * |
| Float | 8 | double |
| Float32 | 8 | double |
| Integer | 4 | int |
| Integer8 | 4 | int |
| Integer16 | 4 | int |
| Integer64 | 8 | long long |
| Varchar | Variable | char[] |

* The *DateTime* data type is a structured value of size eight bytes. No support is currently provided by the API for manipulating values of this type.

If the size of the data requested is more than user supplied buffer, only the amount requested is copied, with the amount copied being returned as the result. In this case the error code is also set, which can be retrieved by **QueryAPI::GetError,** which indicates that the data has been truncated. In the case of a string the truncated data will not contain a null terminator If the size of the data requested is less than the user supplied buffer, only the data available is copied and the amount copied returned as the result with no error code set.
The following code fragment executes the query, "select name, age from person" and copies the data for each row:

```
#include "appapi.h"
#include "clntapi.h"
#include "queryapi.h"

class MyApp
{
    ClientAPI *Client;
    QueryAPI *Query;
    static const void GotRowCallback(void *userdata);
    static const void QueryCompleteCallback(void *userdata);
public:
    void StartQuery();
};

void MyApp::StartQuery()
{
    // Initiate the query
    Query = QueryAPI::StartQuery(Client,
                                 "select name, age from person",
                                 &MyApp::GotRowCallback,
                                 &MyApp::QueryCompleteCallback,
                                 this);
}

const void MyApp::GotRowCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;
    char name[100];
    int age;

    // Obtain the data for a row here
    if (QueryAPI::CopyColumn(app->Query, "name", name, 100) == 0)
    {
        // Copy failed
    }

    if (QueryAPI::CopyColumn(app->Query, "age", &age, sizeof(int)) !=
    sizeof(int))
    {
        // Copy failed
    }
}
```

```
const void MyApp::QueryCompleteCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // No more records
}
```

Not all columns available need to be copied and the value of a column can be copied any number of times.
There is an alternative, more efficient way of copying column values supported by the API. This involves specifying the column by number rather than by its name. The API function **QueryAPI::CopyNColumn** is provided by the API for copying data for a column by column number. It has the prototype:

**static int CopyNColumn(**

**const QueryAPI *query,**

**const int columnNumber,**

**void *variable,**

**int bytesToCopy);**

The equivalent C-callable function is:

**int QueryAPI_CopyNColumn(**

**const QueryAPI *query,**

**const int columnNumber,**

**void *variable,**

**int bytesToCopy);**

The behaviour of this function is identical to that of **CopyColumn** except that the column requested is specified by its column number by the parameter *columnNumber* rather than by its name. The column number for a column is its position in the project list of the query. For example, in the query,

```
select a, b from t;
```

the column *a* has the column number 1 and the column *b* has the column number 2. Using column number is more efficient than using column names since the API has to perform a *look-up* when a name is specified whereas when a number is specified it can merely indirect into an array.
The **CopyNColumn** function will error if the column number supplied is out of range.

The column number for a particular column name can be obtained using the API. The API function **QueryAPI::GetColNum** is provided for obtaining the column number for a column. It has the prototype:

**static int GetColNum(**

**const QueryAPI *query,**

**const char *columnName);**

The equivalent C-callable function is:

**int QueryAPI_GetColNum(**

**const QueryAPI *query,**

**const char *columnName);**

This function returns the column number for the column with the name specified by the `columnName` parameter for the query specified by the *query* parameter. If no column exists with the specified name, the function will return an error.

The length of the value of a particular column can be obtained using the API function **QueryAPI::GetColumnLength**. It has the prototype:

**static int GetColumnLength(**

**const QueryAPI *query,**

**const int columnNumber);**

The equivalent C-callable function is:

**int QueryAPI_GetColumnLength(**

**const QueryAPI *query,**

**const int columnNumber);**

This function returns the length in bytes of the value for the column specified by the *columnNumber* parameter for the query specified by the *query* parameter.

The API also supports the retrieval of column values as strings. The API function **QueryAPI::CopyColumnToString** is provided by the API for copying data for a column by column number translating the value to an appropriate string. It has the prototype:

**static int CopyColumnToString(**

**const QueryAPI *query,**

**const int columnNumber,**

**char *buffer,**

**int bufferLength);**

The equivalent C-callable function is:

**int QueryAPI_CopyColumnToString(**

**const QueryAPI *query,**

**const int columnNumber,**

**char *buffer,**

**int bufferLength);**

The behaviour of this function is identical to that of the **CopyNColumn** function except the value is converted to a string and placed in the character buffer pointed to by **buffer**. The length of the buffer is specified by **bufferLength**.
If the size of the data requested is more than the size available, only the amount requested is copied with the amount copied being returned as the result. In this case the error code is also set, which can

be retrieved by **QueryAPI::GetError,** which indicates that the data has been truncated. In the case of a string the truncated data will not contain a null terminator If the size of the data requested is less than the user supplied buffer, only the data available is copied and the amount copied returned as the result with no error code set.

How the value is converted to a string is dependent on its type. The following table shows the conversion that takes place:

| Data Type | String Format |
|---|---|
| Binary | Hexadecimal values |
| Boolean | The string true or false |
| DateTime | dd-mon-yyyy hh:mm:ss |
| Float | Float value |
| Float32 | Float value |
| Integer | Integer value |
| Integer8 | Integer value |
| Integer16 | Integer value |
| Integer64 | Integer value |
| Varchar | String not enclosed in quotes |

The type of a particular column can be obtained using the API. The API function **QueryAPI::GetColumnType** is provided for obtaining the column type for a column. It has the prototype:

**static int GetColumnType(**

**const QueryAPI *query,**

**const int columnNumber);**

The equivalent C-callable function is:

**int QueryAPI_GetColumnType(**

**const QueryAPI *query,**

**const int columnNumber);**

This function returns the type for the column specified by the *columnNumber* parameter for the query specified by the *query* parameter.

Additional information about a particular column can be obtained using the API function **QueryAPI::GetColumnFlags**. It has the prototype:

**static int GetColumnFlags(**

**const QueryAPI *query,**

**const int columnNumber);**

The equivalent C-callable function is:

**int QueryAPI_GetColumnFlags(**

**const QueryAPI *query,**

**const int columnNumber);**

This function returns flags associated with the column specified by the *columnNumber* parameter for the query specified by the *query* parameter. The flags indicate whether the column is a primary key column, can contain null values and can be updated. The supported flags are listed in section 6.

## 2.8 Active Queries

So far we have described how to perform fixed queries and access the data values they retrieve. This section describes how to implement active queries using the API. It is assumed that the user is familiar with the term *active query* and fully understands the concept and the facilities it provides.
The API function **QueryAPI::StartActiveQuery** is provided for creating active SQL queries. It has the prototype:

```
static QueryAPI *StartActiveQuery(
                    const ClientAPI *client,
                    const char *sqlText,
                    const void (*insertedRow)(void *),
                    const void (*updatedRow)(void *),
                    const void (*deletedRow)(void *),
                    const void (*deltaComplete)(void *),
                    const void *userdata,
                    const unsigned int maxRows = POLY_MAX_ROWS_UNLIMITED, /*
                    implies no row limitation */
                    const unsigned int secs = 0,
                    const unsigned int microsecs = 0);
```

The equivalent C-callable function is:

```
QueryAPI *QueryAPI_StartActiveQuery(
                    const ClientAPI *client,
                    const char *sqlText,
                    const void (*insertedRow)(void *),
                    const void (*updatedRow)(void *),
                    const void (*deletedRow)(void *),
                    const void (*deltaComplete)(void *),
                    const void *userdata,
                    const unsigned int maxRows,
                    const unsigned int secs,
                    const unsigned int microsecs);
```

Similarly to the **StartQuery** function, this function executes the SQL query specified by the *sqlText* parameter through the client connection specified by the *client* parameter. It is also non-blocking and returns a pointer to an instance of the QueryAPI class immediately the request has been initiated. Data retrieved by the query and data modifications sent by the database during the queries life-time are communicated to the application via a set of four callback functions, *insertedRow*, *updatedRow*, *deletedRow* and *deltaComplete*.

Data is sent from the database to the client as a sequence of deltas. Each delta can contain a collection of new inserted rows, modifications to existing rows and deletions of existing rows.
For an ordinary active query, a delta corresponds to a transaction performed on the database. The changes sent in a single delta correspond to the effect of that transaction upon the data associated with the active query. The optional *secs* and *microsecs* parameters specify a minimum interval between deltas. Use of these parameters allows the processing load on the server and the network traffic between server and client to be reduced.
In either case, the first delta sent holds the initial set of rows retrieved by the query. For the first delta, the *insertedRow* callback function is invoked once for each row of data retrieved by the query. If no data is retrieved, the function is not called. The same functions provided for accessing the actual data values in a fixed query can be used to access the data contain in the row.
When all rows have been retrieved and the associated invocation of the *insertedRow* callback function completed, the callback function *deltaComplete* is invoked. This is an indication that execution of the query has completed and that no more rows will be returned by the query.
Subsequent deltas can contain updates and deletes in addition to further inserts. The *insertedRow* callback function is invoked each time a row is inserted into the active query. The *updatedRow* callback function is invoked each time a row is updated and the *deletedRow* callback function is invoked each time a row is deleted. The *userdata* parameter specifies a user-defined pointer to be passed into all the callback functions.
The *maxRows* optional parameter can be used to limit the number of rows retrieved by the query. The default value is POLY_MAX_ROWS_UNLIMITED (0xFFFFFFFF or 4294967295), i.e. effectively no limitation
A row update may not involve modifications to all the columns in a row. To obtain the new value for a column in an updated row, the **CopyColumn** and **CopyNColumn** functions are still used. These return an error if the value requested has not changed and do not modify the contents of the buffer they are supplied. Applications therefore have to maintain their own data structures containing the data. When *updatedRow* is invoked, the application typically requires to know which row has been updated so that it can locate the corresponding portion of its data structures and modify the value it contains. Row Ids are supported by the API for identifying rows within an active query. Each row returned by an active query is allocated a unique row Id. The API function **QueryAPI::GetRowId** is provided for obtaining the row Id of the current row. It has the prototype:

**static long GetRowId (**

**const QueryAPI *query);**

The equivalent C-callable function is:

**long QueryAPI_GetRowId (**

**const QueryAPI *query);**

This function can be called from within any of the three row manipulation callback functions, *insertedRow*, *updatedRow* or *deletedRow*. It returns the row Id for the current row being processed. Using this function updated (and deleted) rows can be mapped to their application data structure.
The *deltaComplete* callback function is invoked at the end of each delta. The current row may be one inserted through the active query. In this case, the user needs a way to map client row Ids to (server) row Ids. The API function **QueryAPI::GetClientRowId** is provided for obtaining the client row Id of the current row.
The following code fragment illustrates the use of the **StartActiveQuery** function:

```
#include "appapi.h"
#include "clntapi.h"
#include "queryapi.h"

class AppRowData;

class MyApp
{
    ClientAPI *Client;
    QueryAPI *Query;

    // Functions for manipulating application data structures
    // Implementation of these functions is not provided in this
    // example
    static AppRowData *NewRow(long rowId, char *name, int age);
    static AppRowData *FindRow(long rowId);
    void UpdateRow(AppRowData *data, char *name, int age);
    static void UpdateName(AppRowData *data, char *name);
    static void UpdateAge(AppRowData *data, int age);
    static void DeleteRow(AppRowData *data);

    static const void InsertRowCallback(void *userdata);
    static const void UpdateRowCallback(void *userdata);
    static const void DeleteRowCallback(void *userdata);
    static const void DeltaCompleteCallback(void *userdata);
public:
    void StartActiveQuery();
};

void MyApp::StartActiveQuery()
{
    // Initiate the query
    Query = QueryAPI::StartActiveQuery(Client,
                                       "select name, age from person",
                                        &MyApp::InsertRowCallback,
                                        &MyApp::UpdateRowCallback,
                                        &MyApp::DeleteRowCallback,
                                        &MyApp::DeltaCompleteCallback,
                                        this);
}
```

```
const void MyApp::InsertRowCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;
    char name[100];
    int age;

    // Obtain the data for a row here
    if (QueryAPI::CopyColumn(app-Query, "name", name, 100) == 0)
    {
        // Copy failed
    }

    if (QueryAPI::CopyColumn(app->Query, "age", &age, sizeof(int)) !=
    sizeof(int))
    {
        // Copy failed
    }

    NewRow(QueryAPI::GetRowId(app->Query), name, age);
}

const void MyApp::UpdateRowCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;
    char name[100];
    int age;

    // Obtain the data for a row here
    if (QueryAPI::CopyColumn(app->Query, "name", name, 100))
    {
        // Copy failed
    }
    else UpdateName(FindRow(QueryAPI::GetRowId(app->Query)), name);

    if (QueryAPI::CopyColumn(app->Query, "age", &age, sizeof(int)))
    {
        // Copy failed
    }
    else UpdateAge(FindRow(QueryAPI::GetRowId(app->Query)), age);
}

const void MyApp::DeleteRowCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    DeleteRow(FindRow(QueryAPI::GetRowId(app-Query)));
}

const void MyApp::DeltaCompleteCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // No more records
}
```

The API function **QueryAPI::StartActiveObjectQuery** is provided for creating active object queries. It has the prototype:

```
static QueryAPI *StartActiveObjectQuery(
                    const ClientAPI *client,
                    const char *tableName,
                    const void (*insertedRow)(void *),
                    const void (*updatedRow)(void *),
                    const void (*deletedRow)(void *),
                    const void (*deltaComplete)(void *),
                    const void *userdata,
                    const void (*collectArgs)(QueryAPI *, void *),
                    const unsigned int secs = 0,
                    const unsigned int microsecs = 0);
```

The equivalent C-callable function is:

```
QueryAPI *QueryAPI_StartActiveObjectQuery(
                    const ClientAPI *client,
                    const char *tableName,
                    const void (*insertedRow)(void *),
                    const void (*updatedRow)(void *),
                    const void (*deletedRow)(void *),
                    const void (*deltaComplete)(void *),
                    const void *userdata,
                    const void (*collectArgs)(QueryAPI *, void *),
                    const unsigned int maxRows,
                    const unsigned int secs,
                    const unsigned int microsecs);
```

This function executes an active object query on the table (or view) specified by the *tableName* parameter through the client connection specified by the *client* parameter. Parameters for the query and columns to be retrieved are collected within the *collectArgs* callback function (in a similar fashion to **StartObjectQuery**). Row limitation is not supported for active object queries. In all other respects the behaviour of this function is identical to the **StartActiveQuery** function.

The API function **QueryAPI::StartActiveProcedureQuery** is provided for creating active SQL procedure queries. It has the prototype:

```
static QueryAPI *StartActiveProcedureQuery(
                    const ClientAPI *client,
                    const char *procedureName,
                    const void (*insertedRow)(void *),
                    const void (*updatedRow)(void *),
                    const void (*deletedRow)(void *),
                    const void (*deltaComplete)(void *),
                    const void *userdata,
                    const void (*collectArgs)(QueryAPI *, void *),
                    const unsigned int maxRows = POLY_MAX_ROWS_UNLIMITED /*
                    implies no row limitation */,
                    const unsigned int secs = 0,
                    const unsigned int microsecs = 0);
```

This function executes an active SQL procedure query specified by the *procedureName* parameter through the client connection specified by the *client* parameter. Parameters for the query are collected within the *collectArgs* callback function. In all other respects the behaviour of this function is identical to the **StartActiveQuery** function.

## 2.9 Accessing Active Query Row Data

Data contained in the rows returned by an active query can be obtained from the *insertedRow* and *updatedRow* callback functions. The API functions **QueryAPI::CopyColumn**, **QueryAPI::CopyNColumn** and **QueryAPI::CopyColumnToString** described in section 2.7, can be used to obtain values in the same way as for fixed queries. When trying to obtain the value of column from the *updateRow* callback function, the value may be unavailable. The default behaviour for an active query is to make values that have changed available from the *updateRow* callback. This behaviour can be modified using resources specified for the RTRDB (See the *RTRDB* manual for details). If a value is unavailable the API copy functions will return 0 and set the error code obtained by calling **QueryAPI::GetError** to **POLY_ENOVALUE.**
To detect whether a value has changed the API provides the function **QueryAPI::GetColumnChange**. It has the prototype:

```
static int GetColumnChange(
                    const QueryAPI*query,
                    const int columnNumber);
```

The C callable equivalent function is:

```
int QueryAPI_GetColumnChange(
                    const QueryAPI*query,
                    const int columnNumber),
```

The function returns 1 if the value has changed and 0 if it has not. If it has changed, the value will be available. If it is unchanged, the value may or may not be available.

## 2.10 Aborting an Active Query

When an active query is no longer required it should be deleted. All instances of **QueryAPI** should be deleted. The API function **QueryAPI::StartAbort** is provided for deleting instances of **QueryAPI** and aborting queries. It has the prototype:

**static void StartAbort(**

**QueryAPI *query,**

**const void (*queryAborted)(void *),**

**const void *userdata);**

The equivalent C-callable function is:

**void QueryAPI_StartAbort(**

**QueryAPI *query,**

**const void (*queryAborted)(void *),**

**const void *userdata);**

This function deletes the instance of **QueryAPI** specified by the *query* parameter. The query is first aborted before deleting it. This function is non-blocking. It initiates the abort and then returns immediately. The callback function specified by the *queryAborted* parameter is invoked when the query has been aborted and the user-defined pointer specified by the parameter *userdata* is passed to it. **QueryAPI::StartAbort** should be called only after the first delta has been successfully received.

## 2.11 Commands

Commands allow parameterised SQL statements to be prepared and then executed multiple times with different parameter values. A prepared statement is more efficient to execute than one that has not been prepared as the compilation and optimisation of the statement is performed once when it is prepared.

A command is created using the API function **CommandAPI::CreateCommand**, which has the prototype:

**static CommandAPI *CreateCommand(**

**const ClientAPI *client,**

**const char *sqlText);**

The equivalent C-callable function is:

**CommandAPI*CommandAPI_CreateCommand(**

**const ClientAPI *client,**

**const char *sqlText);**

The function creates an instance of the CommandAPI class to represent the command using the SQL statement specified by the *sqlText* parameter. The client connection that the command will use is specified by the *client* parameter. It should be noted that this function does not prepare the SQL statement and that commands can be executed whether or not they have been prepared.
The SQL statement may contain named or unnamed arguments whose values will be provided each time the command is executed.

Once created a command can be prepared using the API function **CommandAPI::StartPrepare**, which has the prototype:

**static int StartPrepare(**

**CommandAPI *command,**

**const void (*prepareComplete)(void *),**

**const void *userdata);**

The equivalent C-callable function is:

**int CommandAPI_StartPrepare(**

**CommandAPI *command,**

**const void (*prepareComplete)(void *),**

**const void *userdata);**

The function prepares the command specified by *command* through the client connection associated with the command when it was created.
The function is non-blocking and returns when it has initiated or failed to initiate preparation of the command. It returns zero to indicate the preparation has been successfully initiated and non-zero to indicate failure. The API function **QueryAPI::GetError** can be used to obtain a more detailed error

if the function fails. Note that the **CommandAPI** class is derived from the **QueryAPI** class and so **QueryAPI** functions that take a pointer to a **QueryAPI** instance as a parameter can also be applied to **CommandAPI** instances. **QueryAPI::GetError** is an example of this.
The *prepareComplete* callback function is invoked when preparation of the command is completed. The *userdata* parameter specifies a user-defined pointer to be passed into *prepareComplete* callback function. The API function **QueryAPI::GetError** must be used in this callback to determine if the preparation was successful.
If preparing a command fails, that command cannot be executed. Either it must be deleted or prepared successfully.

The following code fragment creates and prepares a command with an SQL query to retrieve the name of a person based on their id, which is supplied as a parameter:

```
#include "appapi.h"
#include "clntapi.h"
#include "commandapi.h"

class MyApp
{
    ClientAPI *Client;
    CommandAPI *Command;
    static const void PrepareCallback(void *userdata);
public:
    void StartPrepare();
};

void MyApp::StartPrepare()
{
    // Create the command
    Command = Command::CreateCommand(Client
                "select name from person where id=:<integer>id");

    // Initiate the prepare operation
    int res = CommandAPI::StartPrepare(Command,
                                       &MyApp::PrepareCallback,
                                       this);
}

const void MyApp::PrepareCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    if (QueryAPI::GetError(app->Client))
    {
        // The preparation attempt failed.
    }
    else
    {
        // The preparation attempt succeeded.
    }
}
```

A command can be executed as a fixed query using the API function **CommandAPI::StartQuery,** which has the prototype:

**static int StartQuery(**

**CommandAPI *command,**

**const void (*gotRow)(void *),**

**const void (*queryComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(CommandAPI *, void *),**

**const unsigned int maxRows = POLY_MAX_ROWS_UNLIMITED);**

The equivalent C-callable function is:

**int CommandAPI_StartQuery(**

**CommandAPI *command,**

**const void (*gotRow)(void *),**

**const void (*queryComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(CommandAPI *, void *),**

**const unsigned int maxRows);**

The function executes the command specified by the *command* parameter as a fixed query through the client connection associated with the command when it was created.
It is non-blocking and returns when it has either initiated or failed to initiate execution of the query. It returns zero to indicate the execution has been successfully initiated and non-zero to indicate failure. The API function **QueryAPI::GetError** can be used to obtain a more detailed error if the function fails.
The *collectArgs* callback function is invoked to obtain any argument values that may be required to execute the command. The *collectArgs* callback is passed a pointer to the **CommandAPI** instance and the *userdata* parameter. Argument values are supplied by calling the **QueryAPI::AddArg** function from within the *collectArgs* function.
Data retrieval operates identically to the **QueryAPI::StartQuery** function using the *gotRows* and *queryComplete* callback function and the *userdata* parameter and can use the same API function to obtain the data contained in the rows retrieved by the query. The *maxRows* parameter also operates identically.
When the query completes, the command is not deleted. Instead it is returned to its pre-execution state ready to be re-executed, or explicitly deleted.

The following code fragment executes the previously prepared command as a fixed query:

```
#include "appapi.h"
#include "clntapi.h"
#include "commandapi.h"
#include "queryapi.h"

class MyApp
{
    ClientAPI *Client;
    CommandAPI *Command;
    static const void CollectArgsCallback(QueryAPI *,
                                          void *userdata);
```

```
    static const void GotRowCallback(void *userdata);
    static const void QueryCompleteCallback(void *userdata);
public:
    void StartQuery();
};

void MyApp::StartQuery()
{
    // Initiate the query
    int res = CommandAPI::StartQuery(Command,
                                     &MyApp::GotRowCallback,
                                     &MyApp::QueryCompleteCallback,
                                     this,
                                     &MyApp::CollectArgsCallback);
}

const void MyApp::CollectArgsCallback(CommandAPI *command,
                                      void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // Supply named argument
    int id = 1;
    QueryAPI::AddArg(command, "id", 1, &id, sizeof(int));
}

const void MyApp::GotRowCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // Obtain the data for a row here
}

const void MyApp::QueryCompleteCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // No more records
}
```

A command can be executed as an active query using the API function **CommandAPI::StartActiveQuery,** which has the prototype:

**static int StartActiveQuery(**

**CommandAPI *command,**

**const void (*insertedRow)(void *),**

**const void (*updatedRow)(void *),**

**const void (*deletedRow)(void *),**

**const void (*deltaComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(CommandAPI *, void *),**

**const unsigned int maxRows = POLY_MAX_ROWS_UNLIMITED,**

**const unsigned int secs = 0,**

**const unsigned int microsecs = 0);**

The equivalent C-callable function is:

**int CommandAPI_StartActiveQuery(**

**CommandAPI *command,**

**const void (*insertedRow)(void *),**

**const void (*updatedRow)(void *),**

**const void (*deletedRow)(void *),**

**const void (*deltaComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(CommandAPI *, void *),**

**const unsigned int maxRows = POLY_MAX_ROWS_UNLIMITED,**

**const unsigned int secs = 0,**

**const unsigned int microsecs = 0);**

The function executes the command specified by the *command* parameter as an active query through the client connection associated with the command when it was created.
It is non-blocking and returns when it has either initiated or failed to initiate execution of the query. It returns zero to indicate the execution has been successfully initiated and non-zero to indicate failure. The API function **QueryAPI::GetError** can be used to obtain a more detailed error if the function fails.
The *collectArgs* callback function is invoked to obtain any argument values that may be required to execute the command. The *collectArgs* callback is passed a pointer to the **CommandAPI** instance and the *userdata* parameter. Argument values are supplied by calling the **QueryAPI::AddArg** function from within the *collectArgs* function.
Data retrieval and the handling data modifications (deltas) sent by the database during the life-time of the active query operates identically to the **QueryAPI::StartActiveQuery** function using the *insertedRow*, *updatedRow*, *deletedRow* and *deltaComplete* callback functions and the *userdata* parameters and can use the same API function to obtain the data contained in the rows retrieved by the query. The *maxRows*, *secs* and *microsecs* parameters also operates identically.
Updating via the active query, which is described in section 2.12.2, also operates the same whether the query is created using this function or the **QueryAPI::StartActiveQuery** function. All the same functions used to update via active queries can be used.
The only API function that operates differently when applied to a **CommandAPI** instance rather than a **QueryAPI** instance is the **QueryAPI::StartAbort**. When applied to a **CommandAPI** instance it does not delete the command. Instead it returns the command to its pre-execution state ready to be re-executed, or explicitly deleted.

The following code fragment executes the previously prepared command as an active query:

```
#include "appapi.h"
#include "clntapi.h"
#include "commandapi.h"
#include "queryapi.h"

class MyApp
{
    ClientAPI *Client;
    CommandAPI *Command;
    static const void CollectArgsCallback(QueryAPI *,
                                          void *userdata);
    static const void InsertRowCallback(void *userdata);
    static const void UpdateRowCallback(void *userdata);
    static const void DeleteRowCallback(void *userdata);
```

```
    static const void DeltaCompleteCallback(void *userdata);
public:
    void StartActiveQuery();
};

void MyApp::StartActiveQuery()
{
    // Initiate the query
    res = CommandAPI::StartActiveQuery(Command,
                                       &MyApp::InsertRowCallback,
                                       &MyApp::UpdateRowCallback,
                                       &MyApp::DeleteRowCallback,
                                       &MyApp::DeltaCompleteCallback,
                                       this,
                                       &MyApp::CollectArgsCallback);
}

const void MyApp::CollectArgsCallback(CommandAPI *command,
                                      void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // Supply named argument
    int id = 1;
    QueryAPI::AddArg(command, "id", 1, &id, sizeof(int));
}

const void MyApp::InsertRowCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // New row received
}

const void MyApp::UpdateRowCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // Existing row updated
}

const void MyApp::DeleteRowCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // Existing row deleted
}

const void MyApp::DeltaCompleteCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // No more rows in this delta
}
```

A command is deleted using the API function **CommandAPI::DeleteCommand**, which has the following prototype:

**static int DeleteCommand (**

**CommandAPI *command,**

**const void (*deleteComplete)(void *),**

**const void *userdata);**

The equivalent C-callable function is:

**int CommandAPI_DeleteCommand(**

**CommandAPI *command,**

**const void (*deleteComplete)(void *),**

**const void *userdata);**

The function prepares the command specified by *command* through the client connection associated with the command when it was created.
The function is non-blocking and returns when it has initiated or failed to initiate deletion of the command. It returns zero to indicate the deletion has been successfully initiated and non-zero to indicate failure. The API function **QueryAPI::GetError** can be used to obtain a more detailed error if the function fails.
The *deleteComplete* callback function is invoked when deletion of the command is completed. The *userdata* parameter specifies a user-defined pointer to be passed into *deleteComplete* callback function. The API function **QueryAPI::GetError** must be used in this callback to determine if the deletion was successful.
If deletion is successful the command **CommandAPI** instance is deleted after the *deleteComplete* callback has executed.

The following code fragment deletes a command:

```
#include "appapi.h"
#include "clntapi.h"
#include "commandapi.h"

class MyApp
{
    ClientAPI *Client;
    CommandAPI *Command;
    static const void DeleteCallback(void *userdata);
public:
    void DeleteCommand();
};

void MyApp::DeleteCommand()
{
    // Delete the command
    int res = CommandAPI::DeleteCommand(Command,
                                        &MyApp::DeleteCallback,
                                        this);
}

const void MyApp::DeleteCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // Command deleted
}
```

## 2.12 Transactions

Two forms of transaction are supported by the API. The first allows SQL DML statements to be applied directly to the database. The second allows changes to be made via active queries and commands to be executed.

### 2.12.1 Direct SQL Transactions

SQL DML statements are performed in individual SQL transactions. The API function **TransAPI::StartTrans** is provided for executing SQL transactions. It has the prototype:

**static TransAPI *StartTrans(**

**const ClientAPI *client,**

**const char *sqlText,**

**const void (*transactionComplete)(void *),**

**const void *userdata,**

**int safeCommitMode = false);**

The equivalent C-callable function is:

**TransAPI *TransAPI_StartSQLTrans(**

**const ClientAPI *client,**

**const char *sqlText,**

**const void (*transactionComplete)(void *),**

**const void *userdata,**

**int safeCommitMode);**

This function performs the SQL statements specified by the *sqlText* parameter through the client connection specified by the *client* parameter. It is non-blocking and returns a pointer to an instance of the **TransAPI** class immediately after the transaction has been initiated: that is, the statements have been sent to the database. The callback function specified by the *transactionComplete* parameter is invoked when the transaction completes. It is passed by the user-defined pointer specified by the *userdata* parameter. The callback function is invoked whether or not the transaction succeeds. A private member variable of the **TransAPI** instance is set to record any error that may have occurred. This can be accessed using the **TransAPI::GetError** function.
The **StartTrans** function returns a pointer to an instance of **TransAPI**. Once the callback function has terminated, this instance is automatically deleted by the API. The following code fragment performs the SQL statement "update person set age to 65 where name = 'Lewis England'":

```
#include "appapi.h"
#include "clntapi.h"
#include "transapi.h"

class MyApp
{
    ClientAPI *Client;
    TransAPI *Trans;

    static const void TransactionCompleteCallback(void *userdata);
public:
    void StartTrans();
};

void MyApp::StartTrans()
{
    // Initiate the transaction
    Trans = TransAPI::StartTrans(Client,
```

```
            "update person set age = 65 where name = 'Lewis England'",
               &MyApp::TransactionCompleteCallback,
               this);
}

const void MyApp::TransactionCompleteCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    if (TransAPI::GetError(app->Trans) == 0)
    {
        // Transaction succeeded
    }
    else
    {
        // Transaction failed
    }
}
```

### 2.12.2 Updating via Active Queries

The Call-back API provides support for updating via active queries. All such updates are made in transactions using an instance of the **TransAPI** class. Changes to multiple active queries may be included in the same transaction.
Performing a transaction containing updates on active queries (with conflict detection) consists of the following steps:

1. Creating the transaction object. The API function **TransAPI::CreateTrans** is used to create transaction objects.

2. The active queries to be modified are added to the transaction. The API function **TransAPI::AddQuery** adds a query to a transaction object.

3. Optionally performing a start transaction operation. This disables changes being sent by the server to the active queries included in the transaction and sets the point at which the server's conflict detection is based. The API function **TransAPI::StartTrans** initiates a start transaction operation. A user supplied callback function is used to indicate completion of the operation.

4. Performing a commit transaction operation. This involves specifying the updates to be performed on the active queries and then issuing them to the database. The API function **TransAPI::Commit** is provided to perform the commit operation. This requires two user supplied callback functions. The first is called once for each query included in the transaction in order to collect the updates to be made on that query. The three QueryAPI functions, **InsertColumn**, **UpdateColumn** and **DeleteRow** are provided for specifying these updates. These functions can only be called from within this callback function. The second callback function is called when the commit operation has completed and can detect whether the transaction has succeeded or failed.

5. Finally the transaction object is deleted using the API function **TransAPI::DeleteTrans**. A user supplied callback function is called to indicate completion. Transaction objects can be re-used by repeating steps 3 and 4 any number of times before eventually deleting the transaction.

The following code fragment is split into contiguous parts with detailed explanations of the operations between them. The **MyTrans** function which initiates the process by creating a query ("select * from mytable"). When the query has completed, the *deltaCompleteCallback* function is called, and it is from within this function that the transaction is created. In this example, the table *mytable* has two columns (id integer, char name).

```
#include "appapi.h"
#include "clntapi.h"
#include "transapi.h"
#include "queryapi.h"

class MyApp
{
    ClientAPI *Client;
    TransAPI *Trans;
    QueryAPI *Query;

    static const void RowInsertCallback(void *userdata);
    static const void RowUpdateCallback(void *userdata);
    static const void RowDeleteCallback(void *userdata);

    static const void DeltaCompleteCallback(void *userdata);
    static const void StartTransCompleteCallback(void *userdata);
    static const void UpdateCallback(QueryAPI *query, void
*userdata);
    static const void CommitCallback(void *userdata);
    static const void DeleteTransCompleteCallback(void *userdata);

  public:
    void MyTrans();
};

void MyApp::MyTrans()
{
    Query = QueryAPI::StartActiveQuery (Client,
                                "select * from mytable",
                                MyApp::RowInsertCallback,
                                MyApp::RowUpdateCallback,
                                MyApp::RowDeleteCallback,
                                MyApp::DeltaCompleteCallback,
                                this);
}
```

The API function **TransAPI::CreateTrans** is provided for creating SQL transactions. It has the prototype:

**static TransAPI *CreateTrans(**

**const ClientAPI *client);**

The equivalent C-callable function is:

**TransAPI *TransAPI_CreateTrans(**

**const ClientAPI *client);**

This function creates an instance of **TransAPI** on a database which has a client identified by *client*. A pointer to the created **TransAPI** instance is returned to the calling function.

The API function **TransAPI::AddQuery** is provided to specify which queries are included in the transaction. It has the prototype:

**static int AddQuery(**

**TransAPI *trans,**

**const QueryAPI *query);**

The equivalent C-callable function is:

**int TransAPI_AddQuery(**

**TransAPI *trans,**

**const QueryAPI *query);**

This function adds the query instance, *query*, to the transaction *trans*. When the operation is complete, the error code is written to a private member variable. The **TransAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

The API function **TransAPI::StartTrans** is provided to perform the start transaction operation. It has the prototype:

**static int StartTrans(**

**const TransAPI *trans,**

**const void (*callback)(void *),**

**void *userdata);**

The equivalent C-callable function is:

**int TransAPI_StartTrans(**

**const TransAPI *trans,**

**const void (*callback)(void *),**

**void *userdata);**

This function starts a transaction, already created by calling **TransAPI::CreateTrans**. When the operation is complete, the error code is written to a private member variable, and the function pointed to by *callback* is invoked on the *userdata* instance.
The **TransAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

The following code fragment shows the transaction being created, in the *deltaCompleteCallback* function.

```
const void MyApp::DeltaCompleteCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    if ((app->Trans = TransAPI::CreateTrans(app->Client)) != 0)
    {
        // CreateTrans failed
    }

    // CreateTrans succeeded, add the query to the transaction.

    if (TransAPI::AddQuery(app->Trans, app->Query) != 0)
    {
        // AddQuery failed
    }

    // Now start the transaction

    TransAPI::StartTrans(app->Trans,
                         MyApp::StartTransCompleteCallback,
                         app);
}

const void MyApp::RowInsertCallback(void *userdata)
{
}

const void MyApp::RowUpdateCallback(void *userdata)
{
}

const void MyApp::RowDeleteCallback(void *userdata)
{
}
```

The API function **TransAPI::Commit** is provided to perform the commit transaction operation. It has the prototype:

**static int Commit(**

**const TransAPI *trans,**

**const void (*UpdateCallback) (QueryAPI *, void *),**

**const void (*CommitCallback) (void *),**

**void *userdata,**

**bool safeCommitMode = false);**

The equivalent C-callable function is:

**int TransAPI_Commit(**

**const TransAPI *trans,**

**const void (*UpdateCallback) (QueryAPI *, void *),**

**const void (*CommitCallback) (void *),**

**void *userdata,**

**int safeCommitMode);**

This function is used to initiate the commit operation on a transaction, passed as *trans*. The function takes two callback functions, the first called when the commit has been started, and is waiting for data modification on the queries added to the transaction. It is invoked once for each query included in the transaction, passing a pointer to the relevant **QueryAPI** object as its first parameter. The second callback function is called when the commit has been completed. When the commit is initiated, the error code is written to a private member variable, and the function pointed to by *UpdateCallback* is invoked on the *userdata* instance. When the commit operation is complete, the error code is written to a private member variable, and the function pointed to by *CommitCallback* is invoked on the *userdata* instance.
The **TransAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

The following code fragment shows the commit procedure being started, in the *startTransCompleteCallback* function.

```
const void MyApp::StartTransCompleteCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    if (TransAPI::Commit(app->Trans,
                         MyApp::UpdateCallback,
                         MyApp::CommitCallback,
                         app) != 0)

    {
        // Commit failed
    }
}
```

There are only three API functions, which can be used to modify data in the context of the ***UpdateCallback*** function from an active query. These are **QueryAPI::InsertColumn**, **QueryAPI::UpdateColumn** and **QueryAPI::DeleteRow**.

**static int InsertColumn(**

**const QueryAPI *query,**

**long clientRowId,**

**int columnNumber,**

**void *buffer,**

**int length);**

The equivalent C-callable function is:

**int QueryAPI_InsertColumn(**

**const QueryAPI *query,**

**long clientRowId,**

**int columnNumber,**

**void *buffer,**

**int length);**

Used in groups, one call for each column of each row to be inserted, this function inserts a value to the given *clientRowId* and *columnNumber*.
When the operation is complete, the error code is written to a private member variable, and the function pointed to by *callback* is invoked on the *userdata* instance.

The **QueryAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

**static int UpdateColumn(**

**const QueryAPI *query,**

**long rowId,**

**int columnNumber,**

**void *buffer,**

**int length);**

The equivalent C-callable function is:

**int QueryAPI_UpdateColumn(**

**const QueryAPI *query,**

**long rowId,**

**int columnNumber,**

**void *buffer,**

**int length);**

This function updates a value to the given *rowId* and *columnNumber*.
When the operation is complete, the error code is written to a private member variable, and the function pointed to by *callback* is invoked on the *userdata* instance.
The **QueryAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

**static int DeleteRow(**

**const QueryAPI *query,**

**long rowId);**

The equivalent C-callable function is:

**int QueryAPI_DeleteRow(**

**const QueryAPI *query,**

**long rowId);**

This function causes the record specified by the *rowId* parameter to be deleted.
When the operation is complete, the error code is written to a private member variable, and the function pointed to by *callback* is invoked on the *userdata* instance.
The **QueryAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

The following code fragment here shows the data modification procedure, in the *UpdateCallback* function.

```
const void MyApp::UpdateCallback(QueryAPI *query, void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    int idVal = 10;
    char nameVal[3] = "Hi";

    // Insert a row with client row Id 1
    // Column 1 has value 10
    if (QueryAPI::InsertColumn(query, 1, 1,
                               &idVal, sizeof(int)) != 0)
    {
        // Failure
    }

    // Column 2 has value "Hi"
    if (QueryAPI::InsertColumn(query, 1, 2,
                               &nameVal, sizeof(nameVal)) != 0)
    {
        // Failure
    }

    // Update column 2 of row 2 to "Hi"
    if (QueryAPI::UpdateColumn(query, 2, 2,
                               &nameVal, sizeof(nameVal)) != 0)
    {
        // Failure
    }

    // Delete row 3
    if (QueryAPI::DeleteRow(query, 3) != 0)
    {
        // Failure
    }
}
```

The API function **TransAPI::DeleteTrans** is provided for deleting transaction objects. It has the following prototype:

**static void DeleteTrans(**

**const TransAPI *trans,**

**const void (*deleteCallback) (void *),**

**void *userdata);**

The equivalent C-callable function is:

**void TransAPI_DeleteTrans(**

**const TransAPI *trans,**

**const void (*deleteCallback) (void *),**

**void *userdata);**

This function deletes the transaction pointed to by trans. When the operation is complete, the function pointed to by *deleteCallback* is invoked on the *userdata* instance.
The **TransAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

The code example here shows the delete transaction procedure, in the commit complete callback function, followed by the *deleteTransCompleteCallback* which aborts the query.

```
const void MyApp::CommitCompleteCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    TransAPI::DeleteTrans(app->Trans,
                          MyApp::DeleteTransCompleteCallback,
                          app);
}

const void MyApp::DeleteTransCompleteCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    QueryAPI::StartAbort(app->Query, NULL, app);
}
```

A transaction object can be used for multiple updates through an active query. This is done in the natural way. One can simply do TransAPI::AddQuery and TransAPI::Commit for the first transaction, then TransAPI::AddQuery and TransAPI::Commit for the second transaction and so forth.

### 2.12.3 Executing Commands in a Transaction

A command can be executed in a transaction using the API function **TransAPI::AddCommand**, which has the prototype:

**static int AddCommand(**

**TransAPI *trans,**

**CommandAPI *command,**

**const void (*collectArgs)(Command API, void *),**

**const void *userdata);**

The equivalent C-callable function is:

**int TransAPI_AddCommand(**

**TransAPI *trans,**

**CommandAPI *command,**

**const void (*collectArgs)(Command API, void *),**

**const void *userdata);**

This function adds the command, *command*, to the transaction *trans*. It returns zero to indicate success and non-zero to indicate failure. The API function **QueryAPI::GetError** can be used to obtain a more detailed error if the function fails.
The *collectArgs* callback function is invoked to obtain any argument values that may be required to execute the command. It is invoked immediately when this function is called and not when the transaction is committed. The *collectArgs* callback is passed a pointer to the **CommandAPI** instance and the *userdata* parameter. Argument values are supplied by calling the **QueryAPI::AddArg** function.
The same command can be added multiple times to the same transaction, each time with different argument values, and it will be executed that many times when the transaction is committed.
A command cannot be added to a transaction if it is already added to a different transaction. Also, a command cannot be added to a transaction if it has been added as a query using the API function **TransAPI::AddQuery** and likewise a command cannot be added as a query if it has been added to a transaction. A command is automatically removed from a transaction when the transaction is committed or rolled-back.
The API function **TransAPI::StartTrans** will fail if any commands have been already added to the transaction.

The following code fragment executes the a command in a transaction

```
#include "appapi.h"
#include "clntapi.h"
#include "commandapi.h"
#include "queryapi.h"
#include "transapi.h"

class MyApp
{
    ClientAPI *Client;
    CommandAPI *Command;
    TransAPI *Trans;
    static const void CollectArgsCallback(QueryAPI *,
                                          void *userdata);
    static const void UpdateCallback(void *userdata);
    static const void CommitCallback(void *userdata);
public:
    void CommitTrans();
};

void MyApp::CommitTrans()
{
    // Create transaction
    Trans = TransAPI::CreateTrans(Client);

    // Add command to transaction
    // Assumes command has previously been created with the SQL
    // "update person set age=:<integer>age where id=:<integer>id"
    res = TransAPI::AddCommand(Command,
                               &MyApp::CollectArgsCallback);
                               this);


    // Commit the transaction
    res = TransAPI::Commit(Trans,
                           &MyApp::UpdateCallback,
                           &MyApp::CommitCallback,
                           this);
}

const void MyApp::CollectArgsCallback(CommandAPI *command,
                                      void *userdata)
```

```
{
    MyApp *app = (MyApp *)userdata;

    // Supply arguments
    int id = 1;
    QueryAPI::AddArg(command, "id", 1, &id, sizeof(int));
    int age = 82;
    QueryAPI::AddArg(command, "age", 2, &age, sizeof(int));
}

const void MyApp::UpdateCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // No updates to apply
}

const void MyApp::CommitCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    // Commit completed
    // Check result with TransAPI::GetError
}
```

It is possible to combine updating via active queries and executing commands in the same transaction. It is important to note that transaction objects can be re-used for successive transactions avoiding the overhead of creating a new transaction object for each transaction.

## 2.13 Logging out of a Server

Logging out of a server with the StartLogout function gives the user the ability to log off as a particular user and return the connection to an open connection.

The function **ClientAPI::StartLogout** is provided by the API for logging off from a server. It has the following prototype:

**static void StartLogout(**

**ClientAPI*client,**

**const void(*logoutcallback)(void*),**

**const void*userdata);**

The equivalent C-callable function is:

**void ClientAPI_StartLogout(**

**ClientAPI*client,**

**const void(*logoutCallBack)(void*),**

**const void*userdata);**

This function is non-blocking. It returns when the logout request has been processed. The response from the server is communicated to the application via the callback function *logoutCallBack*. The *userdata* parameter specifies a user-defined pointer to be passed into the *logoutCallBack* callback function.

## 2.14 Closing a Client Connection

All instances of **ClientAPI** created using the API should be deleted and the connection to the database associated with it closed. The API function **ClientAPI::DeleteClient** is provided for deleting instances of **ClientAPI** and closing connections. It has the prototype:

**static void DeleteClient (**

**ClientAPI *client,**

**const void (*deletedClient)(void *),**

**const void *userdata);**

The equivalent C-callable function is:

**void ClientAPI_DeleteClient (**

**ClientAPI *client,**

**const void (*deletedClient)(void *),**

**const void *userdata);**

This function deletes the instance of ClientAPI specified by the *client* parameters, providing there are no outstanding queries or transactions on process on the connection. The following fragment of code closes a client connection:

```
#include "appapi.h"
#include "clntapi.h"

class MyApp
{
    ClientAPI *Client;
    static const void DeleteCallback(void *userdata);
public:
    void DeleteClient();
};

void MyApp::DeleteClient()
{
    // Initiate the delete operation
    ClientAPI::DeleteClient(Client,
                            &MyApp::DeleteCallback,
                            this);
}

const void MyApp::DeleteCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;
    .
    .
    .
}
```

## 2.15 Terminating a Server

It is possible to terminate the server by issuing a shutdown control message from the client. The API function **ClientAPI::StartShutdown** is provided for deleting instances of **ClientAPI**, closing connections and closing the server to which the client connection is established. It has the prototype:

**static void StartShutdown (**

**ClientAPI *client,**

**const void (*shutdownServer)(void *),**

**const void *userdata);**

The equivalent C-callable function is:

**void ClientAPI_StartShutdown (**

**ClientAPI *client,**

**const void (*shutdownServer)(void *),**

**const void *userdata);**

This function issues a shutdown control message to the server. It is non-blocking and returns immediately the message has been sent. The callback function specified by the *shutdownServer* parameter is invoked when the server has shutdown. It is passed the user-defined pointer specified by the *userdata* parameter. A private member variable of the **ClientAPI** instance is set to record any errors that may have occurred. This can be accessed by the **ClientAPI::GetError** function. When the callback function returns, and if the server was successfully shutdown, the instance of the ClientAPI is deleted.

The following fragment of code closes a client connection, terminating the server.

```
#include "appapi.h"
#include "clntapi.h"

class MyApp
{
    ClientAPI *Client;
    static const void ShutdownCallback(void *userdata);
public:
    void ShutdownServer();
};

void MyApp::ShutdownServer()
{
    // Initiate the shutdown operation
    ClientAPI::StartShutdown(Client,
                             &MyApp::ShutdownCallback,
                             this);
}

const void MyApp::ShutdownCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;
    .
    .
    .
}
```

## 2.16 Setting Options

The API function **ClientAPI::SetOption** is provided for setting options in a ClientAPI application. It has the prototype:

**static void SetOption (**

**ClientAPI *client,**

**int option,**

**int optionValue);**

The equivalent C-callable function is:

**void ClientAPI_SetOption (**

**ClientAPI *client,**

**int option,**

**int optionValue);**

The *client* parameter specifies the client connection for which the option is to be set. If *client* is null, the option is set globally for client connections. Setting an option for a specific client connection overrides the global setting of that option.
By default all fault tolerant features are disabled. They may be enabled either globally for all client connections or on an individual connection basis, using the POLY_FT_OPTION_ENABLE option. The following is the complete list of options supported:

- POLY_FT_OPTION_ENABLE
  This option must be specified to enable fault tolerance for a client. The *optionValue* is TRUE to enable fault tolerance, FALSE to disable. By default, fault tolerance is disabled.

- POLY_FT_OPTION_RECONNECTION_RETRIES
  This specifies the number times to attempt to connect to each service during fail-over. The *optionValue* is interpreted as the number of retries.
- POLY_FT_OPTION_RECONNECTION_INTERVAL
  This specifies the interval between each attempt to connect to a service during fail-over. The *optionValue* is interpreted as the interval.
- POLY_FT_OPTION_RECONNECTION_TIMEOUT
  This specifies the time-out period associated with an attempt to connect to each service during fail-over. The *optionValue* is interpreted the connection time-out.
- POLY_FT_OPTION_HEALTHCHECK_INTERVAL
  This specifies the interval between heartbeat messages sent from the client to the RTRDB. The *optionValue* is interpreted as the number of milliseconds between each heartbeat message being sent.
- POLY_FT_OPTION_HEALTHCHECK_TIMEOUT
  This specifies the time-out period associated with each heartbeat message. Absence of a reply to a heartbeat message before this period has elapsed will result in client assuming the RTRDB has failed and instigating its fail-over procedure.
- POLY_FT_OPTION_KEEP_COPY
  This indicates whether the API should keep a copy of the data associated with an active query. Keeping a copy of the data allows deltas that occur as a result of fail-over to be calculated. A non-zero value for *optionValue* indicates a copy should be kept, zero that no copy should be kept.
- POLY_FT_OPTION_MAP_ROWIDS
  This indicates whether the API should maintain row ids for active queries across a fail-over. To do so the API must keep a copy of the primary key columns return by the query. A non-zero value for *optionValue* indicates that row ids should be maintained, zero that they should not be maintained. Note, setting POLY_FT_OPTION_KEEP_COPY implies POLY_FT_OPTION_MAP_ROWIDS.
- POLY_OPTION_CONNECTION_TIMEOUT
  If greater than zero, the value specifies the number of seconds to wait before timing out an attempt to connect to a Polyhedra server. A value of zero specifies no timeout.

The value of a particular option can also be obtained. The API function **ClientAPI::GetOption** is provided to obtain the value of client option. It has the prototype:

**static int GetOption(**

**ClientAPI *client,**

**int option);**

The equivalent C-callable function is:

**int ClientAPI_GetOption(**

**ClientAPI *client,**

**int option);**

The value of the *client* parameter refers to the specific client connection from which the option is to be obtained. If *client* is null, the option is obtained globally. The function returns the value of the specified option.

## 2.17 Fault Tolerance

The Call-back API provides support for implementing fault tolerant clients. That is, clients that provide automatic fail-over from alternative data services in a system configured to provide dual redundant databases. Details of how to configure such a system can be found in the *Real-Time Relational Database(RTRDB)* manual.
The fault tolerant features supported by the API are provided in a flexible manner. Memory usage by the API is an important consideration on many platforms and some features of fault tolerance require additional memory overhead. For this reason the fault tolerant features supported by the API are optional. The fault tolerant functionality supported is on a per (logical) client connection basis.
This section details the following aspects of implementing a fault tolerant client:

- Enabling a fault tolerant client
- Opening a fault tolerant client connection
- Fault tolerant transactions
- Obtaining the actual data service connection

### 2.17.1 Enabling a Fault Tolerant Client

Fault tolerance can be enabled in a client by setting the option POLY_FT_OPTION_ENABLE to TRUE with the function **ClientAPI::SetOption**. See section 2.16 for further details.

### 2.17.2 Opening a Fault Tolerant Client Connection

A fault tolerant client connection requires two data services to be specified when it is established. The ClientAPI function **ClientAPI::StartConnect** supports the specification of multiple data services by interpreting its first argument as a comma separated list of data service names. The following fragment of code attempts to establish a connection to a dual redundant system comprising two databases both listening on port 5000 on the hosts specified by the IP addresses 1.2.3.4 and 1.2.3.5,

```
#include "appapi.h"
#include "clntapi.h"

class MyApp
{
    ClientAPI *Client;
    static const void ConnectCallback(void *userdata);
public:
    void StartConnect();
```

```
};

void MyApp::StartConnect()
{
    // Enable FT functionality
    ClientAPI::SetOption(NULL, POLY_FT_OPTION_ENABLE, TRUE);

    // Initiate the connect operation
    Client = ClientAPI::StartConnect("1.2.3.4:5000,1.2.3.5:5000",
                                     &MyApp::ConnectCallback,
                                     this);
}

const void MyApp::ConnectCallback(void *userdata)
{
    MyApp *app = (MyApp *)userdata;

    if (ClientAPI::GetError(app->Client))
    {
        // The connection attempt failed.
    }
}
```

### 2.17.3 Fault Tolerant Transactions

In a Polyhedra system configured for fault tolerance it is still possible for transactions to fail due to the unexpected failure of one of the databases. The fail-over mechanism does not guarantee that no transactions will be lost. A transaction in process - dispatched to the server but not acknowledged - when the client detects the failure of the server and instigates a fail-over may or may not have succeeded. A special error is returned by the API function **TransAPI::GetError** to indicate this has occurred.

### 2.17.4 Obtaining the Actual Data Service Connection

It is possible for a client to determine which of the multiple data services specified at connection is actually being used at any particular point. The API function **ClientAPI::GetServiceName** is provided for this purpose. It has the prototype:

```
static void GetServiceName(
                ClientAPI *client,
                char *buffer,
                int length);
```

The equivalent C-callable function is:

```
void ClientAPI_GetServiceName(
                ClientAPI *client,
                char *buffer,
                int length);
```

The *client* parameter specifies the client connection from which the name is obtained. The name is copied into the buffer pointed to by the *buffer* parameter. The *length* parameter specifies the length of the buffer. Any name copied into the buffer will be truncated so as not to overwrite beyond the end of the buffer.

### 2.17.5 Forcing a Fail-Over

It is possible for a client to perform a controlled fail-over to an alternative database. The API function **ClientAPI::FailOver** is provided for this purpose. It has the prototype:

**static void FailOver(**

**ClientAPI *client);**

The equivalent C-callable function is:

**void ClientAPI_FailOver(**

**ClientAPI *client);**

The *client* parameter specifies the client connection from which the name is obtained. The current connection is closed and an attempt is made to re-establish to an alternative server. The list of services supplied to the **ClientAPI::StartConnect** call are cycled through, in accordance with the normal client fail-over strategy, until a connection attempt is successful.

### 2.17.6 Connecting to the Standby Database

It is possible for a client to connect to a standby database. A standby database only provides a 'read-only' service (except for data columns marked as *local*). It is not possible to perform any database updates on the standby database on non-local data and any attempt to do so will fail with a warning message.
A client can only connect to a standby database by supplying the *ro*, *standby*, or *any* option with the data service name when connecting (see the Resources section of the User Guide for details of data service options).

### 2.17.7 Monitoring Fault Tolerant Modes

The actual database to which a connection is established can be in one of three modes. It is either: running as a master database with a standby database also running; running as a master database with no standby database currently running; or running as a standby database. Note, that in the latter case a master database must also be running.
It is possible for a client to obtain the current fault tolerant mode of a connection. The API function **ClientAPI::GetFTMode** is provided for this purpose. It has the prototype:

**static int GetFTMode(**

**ClientAPI *client);**

The equivalent C-callable function is:

**int ClientAPI_GetFTMode(**

**ClientAPI *client);**

The possible return values and their meanings are as follows:

0 The connection is to a master database running without a standby database.

1 The connection is to a master database running with a standby database available.

2 The connection is to a standby database. Consequently, a master database must be available.

3 The connection is to a replica database.

It is possible for a client application to be notified whenever the fault tolerant mode of a connection changes. The optional user-defined callback *modeChangeCallBack* and associated user-data *modeChangeUserData* supplied to the **ClientAPI::StartConnect** function register a call-back function that is called every time the fault tolerant mode of the connection changes.

# 3. API Reference

This section specifies the public interface of the classes that constitute the Application API.

## WARNING:

**Unless otherwise stated, parameters that are pointers to functions specifying call-back functions cannot be null, as their behaviour is undefined.**

---

## AppAPI

This section specifies the public member functions of the **AppAPI** class.

---

### AppAPI::Init

**static int Init();**

**C-callable Equivalent**

**int AppAPI_Init(void);**

**Description**

This function performs all the necessary initialisation before any Polyhedra functions may be called. This must be the first thing done in any task that wishes to use Polyhedra.
The function returns zero to indicate the initialisation was successful, and some non-zero number to indicate a specific failure.

**Return Value**

Returns zero for success and non-zero for failure.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 1 | POLY_EALREADYINIT | The scheduler has already been initialised. |

## AppAPI::Create

**static AppAPI* Create ();**

**C-callable Equivalent**

**AppAPI*AppAPI_Create();**

**Description**

This function creates an instance of the AppAPI class from which client connections to the database are created. Each thread within the client application must create its own instance of the AppAPI.

**Return Value**

Returns a pointer to a newly created instance of the AppAPI class or null to indicate failure.

## AppAPI::Delete

**static int Delete(**

**AppAPI *app);**

**C-callable Equivalent**

**int AppAPI_Delete(**

**AppAPI *app);**

**Parameters**

app A pointer to an instance of **AppAPI**.

**Description**

This function deletes an instance of the AppAPI class. It is important to call this function on embedded platforms to ensure that all client processes are terminated.

**Return Value**

Returns zero for success and non-zero for failures.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 2 | POLY_ENOINIT | The scheduler has not been initialised. |

## AppAPI::Execute

**The OSE form of the function**

**static int AppAPI::Execute(**

**AppAPI *app,**

**union SIGNAL **sig = 0);**

**The C-callable equivalent is:**

**int AppAPI_Execute(**

**AppAPI *app,**

**union SIGNAL **sig);**

**The non-OSE form of the function**

**static int AppAPI::Execute(**

**AppAPI *app);**

**The C-callable equivalent is:**

**int AppAPI_Execute(**

**AppAPI *app);**

**Parameters**

| | |
|---|---|
| app | A pointer to an instance of **AppAPI**. |
| sig | A pointer to a variable containing a signal to process. |

**Description**

This function provides a way of running the Polyhedra scheduler if the **AppAPI::Start()** function is not being used. Please note that if using **AppAPI::Execute()**, it must be called regularly to ensure that internal timers and messages from Polyhedra servers are handled.

We first describe the OSE form of the function.
This function allows the Callback API to be used by an OSE process constructed around a main loop that receives and processes OSE signals.
If supplied with zero value for the *sig* parameter, the function will operate as the non-OSE form of the function operates.
If supplied with a non-zero value for the *sig* parameter, the function will run the Polyhedra scheduler until the signal supplied has been processed.
If the signal supplied is understood by Polyhedra, then the pointer pointed to by the *sig* parameter will be set to zero. If the signal supplied is not understood by Polyhedra, then the scheduler will not run and the pointer pointed to by the *sig* parameter will be unaltered.

It is important to note that single Polyhedra signals are not guaranteed to produce a single event when processed by the Polyhedra scheduler. Therefore code should not be written using **AppAPI::Execute()** that assumes this is so. A single signal may generate more than one event because Polyhedra attempts to optimise signal usage by combining small events into one signal. Similarly some events may require several signals. In these cases repeated calls to **AppAPI::Execute()** will cause the Polyhedra scheduler to finally process the event once all the relevant signals have been received.

We now describe the non-OSE form of the function.
If there is an idle function defined and the idle function returns a value indicating that it should be called again, the **AppAPI::Execute()** function will run the Polyhedra scheduler briefly. The **AppAPI::Execute()** function will then return. Otherwise the **AppAPI::Execute()** function will run the scheduler until a message has been received or the time for the next internal timer has been reached.

**Return Value**

Returns zero for success and non-zero for failures.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 2 | POLY_ENOINIT | The scheduler has not been initialised. |
| 3 | POLY_ENOSTART | The scheduler has not been started-up. |

The following fragment of C code shows how an OSE process is written using this function:

```
#include "ose.h"
#include "appapi.h"

OS_PROCESS(example)
{
      AppAPI *app;
      /* Initialise the Polyhedra scheduler */
      AppAPI_Init();
      /* Create an instance of the Polyhedra scheduler */
      app = AppAPI_Create();
      /* Main loop */
      for (;;)
      {
            static const SIGSELECT sigAny[] = { 0 };
            union SIGNAL *sig;
            /* Wait for a signal */
            sig = receive((SIGSELECT *)sigAny);
            /* Hand it to Polyhedra for processing */
            AppAPI_Execute(app,  &sig);
            /* If sig is now 0, Polyhedra handled and freed it */
            if (sig)
            {
                  /* Process non-Polyhedra signal */
                  …
                  free_buf(&sig);
            }
      }
}
```

## AppAPI::ExecuteWithTimeout

**static int AppAPI::ExecuteWithTimeout(**

**AppAPI *app,**

**int timeout_secs,**

**int timeout_microsecs);**

**The C-callable equivalent is:**

**int AppAPI_ExecuteWithTimeout(**

**AppAPI *app,**

**int timeout_secs,**

**int timeout_microsecs);**

**Parameters**

| | |
|---|---|
| app | A pointer to an instance of **AppAPI**. |
| timeout_secs | The number of seconds. |
| timeout_microsecs | The number of microseconds. |

**Description**

This operates in a similar fashion to the **AppAPI::Execute()** function. It will wait for a message. It will do this for the length of time specified by the parameters to the function (in combination). The second parameter specifies a number of seconds and third parameter a number of microseconds. If the wait times out, the return value reflects this. See the table of error codes below.

**Return Value**

Returns zero for success and non-zero for failures.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 2 | POLY_ENOINIT | The scheduler has not been initialised. |
| 3 | POLY_ENOSTART | The scheduler has not been started-up. |
| 25 | POLY_ETIMEDOUT | The operation timed out. |

## AppAPI::Start

**static int Start(**

**AppAPI* app,**

**const int (*userFun)(void *),**

**const void*userdata);**

**C-callable Equivalent**

**int AppAPI_Start(**

**AppAPI* app,**

**const int (*userFun)(void *),**

**const void*userdata);**

**Parameters**

| | |
|---|---|
| app | A pointer to an instance of **AppAPI**. |
| userFun | A pointer to a function that takes a void * as its only parameter and returns an int. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |

**Description**

This function starts up the Polyhedra scheduler, and informs the scheduler that the first application function to invoke is *userFun* with *userdata* as its parameter.
*UserFun* returns zero to the scheduler to tell it not to call this application again, or any non-zero value to tell the scheduler to call the application again as part of the normal scheduling operations.
The **Start** function does not return unless the Polyhedra scheduler is closed down. If this does happen, the function returns zero to indicate the scheduler was originally started correctly, or some non-zero value to indicate a specific failure.

**Return Value**

Returns zero for success and non-zero for failure.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 2 | POLY_ENOINIT | The scheduler has not been initialised. |

## AppAPI::SetFun

**static int SetFun(**

**AppAPI *app,**

**const int (*userFun)(void *),**

**const void *userdata);**

**C-callable Equivalent**

**int AppAPI_SetFun(**

**AppAPI *app,**

**const int (*userFun)(void *),**

**const void *userdata);**

**Parameters**

| | |
|---|---|
| app | A pointer to an instance of **AppAPI**. |
| userFun | A pointer to a function that takes a void * as its only parameter and returns an int. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |

**Description**

This function informs the Polyhedra scheduler that the next application function to invoke is to be *userFun* with the parameter *userdata*. Any previous application function known to the scheduler is overwritten.
*UserFun* returns zero to the scheduler to tell it not to call this application again, or any non-zero value to tell the scheduler to call the application again as part of the normal scheduling operations.
The **SetFun** function returns zero to indicate the new application function has successfully been registered with the scheduler, and some non-zero value to indicate a specific failure.

**Return Value**

Returns zero for success and non-zero for failure.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 2 | POLY_ENOINIT | The scheduler has not been initialised. |
| 3 | POLY_ENOSTART | The scheduler has not been started-up. |

## AppAPI::Stop

**static int Stop();**

**AppAPI *app);**

**C-callable Equivalent**

**int AppAPIStop(**

**AppAPI *app);**

**Parameters**

app A pointer to an instance of **AppAPI**.

**Description**

This function stops the Polyhedra scheduler. When the currently executing callback function returns control to the scheduler its main loop terminates when the scheduler has finished what it is currently doing. Control is then returned to the function that initiated the scheduler by calling the AppAPI**::Start** function.
Once the scheduler is stopped, it can be restarted by a call to **Start()**, in which case the scheduler continues with any work which it was doing when **Stop()** was called (e.g. timers). It is an error to call **Stop()** when the scheduler is not running.

**Return Value**

Returns zero for success and non-zero for failure.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 2 | POLY_ENOINIT | The scheduler has not been initialised. |
| 3 | POLY_ENOSTART | The scheduler has not been started-up. |

## AppAPI::Tidy

**int Tidy();**

**C-callable Equivalent**

**int AppAPI_Tidy (void);**

**Description**

This function tidies up the application task.

On embedded operating systems, it does the following as part of the tidy up:
- Calls the C++ static destructors.
- Frees the static data segment.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 2 | POLY_ENOINIT | The scheduler has not been initialised. |

# TimerAPI

This section specifies the public member functions of the **TimerAPI** class.

## TimerAPI::CreateOneShotTimer

**static TimerAPI *CreateOneShotTimer(**

**AppAPI *app,**

**long dsecs,**

**const void (*timerFun)(void *),**

**const void *userdata);**

**C-callable Equivalent**

**TimerAPI *TimerAPI_CreateOneShotTimer(**

**AppAPI *app,**

**long dsecs,**

**const void (*timerFun)(void *),**

**const void *userdata);**

**Parameters**

| | |
|---|---|
| app | A pointer to an instance of **AppAPI**. |
| dsecs | A long representing a time period in tenths of a second. |
| timerFun | A pointer to a function that takes a void * as its only parameter and returns an int. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |

**Description**

This function creates and initialises a one shot **TimerAPI** instance. If the timer was created successfully, then the function returns a pointer to the created instance, otherwise the function returns a NULL.
Assuming the timer was created successfully, when at least *dsecs* tenths of a second has elapsed. Then callback function *timerFun* is invoked with *userdata* as its parameter. When the callback function returns to the API, the API deletes the timer.

**Return Value**

Returns a pointer to an instance of **TimerAPI**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 2 | POLY_ENOINIT | The scheduler has not been initialised. |

| | | |
|---|---|---|
| 3 | POLY_ENOSTART | The scheduler has not been started-up. |

## TimerAPI::StopTimer

**static void StopTimer (**

**TimerAPI *timer);**

**C-callable Equivalent**

**void TimerAPI_StopTimer (**

**TimerAPI *timer);**

**Parameters**

timer A pointer to an instance of **TimerAPI**

**Description**

This function stops timers. The instance of **TimerAPI** pointed to by *timer* is stopped and deleted. This function can be called from with the callback function of the timer it is stopping. In which case the callback function will execute until completion and then the timer will be deleted.
It should be noted that this function is assumed to succeed, thus no error codes are recorded or returned by the function.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 2 | POLY_ENOINIT | The scheduler has not been initialised. |
| 3 | POLY_ENOSTART | The scheduler has not been started-up. |

## TimerAPI::ReadTimer

**static long ReadTimer(**

**const TimerAPI *timer);**

**C-callable Equivalent**

**long TimerAPI_ReadTimer(**

**const TimerAPI *timer);**

**Parameters**

timer A pointer to an instance of **TimerAPI**

**Description**

This function returns the number of tenths of a second that has to elapse before a timer's user function is invoked. The time left is read from the instance of **TimerAPI** pointed to by *timer*.

**Return Value**

Returns the number of tenths of a second remaining on the timer.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 2 | POLY_ENOINIT | The scheduler has not been initialised. |
| 3 | POLY_ENOSTART | The scheduler has not been started-up. |

# ClientAPI

This section specifies the public member functions of the **ClientAPI** class.

## ClientAPI::StartConnect

**static ClientAPI *StartConnect(**

**AppAPI *app,**

**const char *serverName,**

**const void (*connectedCallBack)(void *),**

**const void *userdata,**

**const void (*disconnectCallBack)(void *) = NULL,**

**const void *disconnectUserData = NULL,**

**const void (*modeChangeCallBack)(void *) = NULL,**

**const void *modeChangeUserData = NULL);**

**C-callable Equivalent**

**ClientAPI *ClientAPI_StartConnect(**

**AppAPI *app,**

**const char *serverName,**

**const void (*connectedCallBack)(void *),**

**const void *userdata,**

**const void (*disconnectCallBack)(void *),**

**const void *disconnectUserData,**

**const void (*modeChangeCallBack)(void *),**

**const void *modeChangeUserData);**

**Parameters**

| | |
|---|---|
| app | A pointer to an instance of **AppAPI.** |
| serverName | A pointer to a null terminated string. The format of the string is the same as the format of the data_service resource. See the User Guide for further details. |
| connectedCallBack | A pointer to a function that takes a void * as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |
| disconnectCallBack | This can be zero or a pointer to a function that takes a void * as its only parameter and returns nothing. |

| | |
|---|---|
| disconnectUserdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |
| modeChangeCallBack | This can be zero or a pointer to a function that takes a void * as its only parameter and returns nothing. |
| modeChangeUserData | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function |

**Description**

This function creates an instance of **ClientAPI** and attempts to connect to the server specified by the address *serverName*. A pointer to the created instance of **ClientAPI** is returned to the calling function.
When the operation is complete, the error code is written to a private member variable in the instance, and the callback function *connectedCallback* is invoked with *userdata* as its parameter.
An accessor function is provided to read the private error code, where zero indicates the connection was accepted successfully, and a non-zero number indicates a specific failure.
Once established, if the connection to the server is lost unexpectedly, the callback function *disconnectCallBack* is invoked with *disconnectUserdata* as its parameter.
If the fault tolerant mode of the connection to the server changes, the callback function *modeChangeCallBack* is invoked with *modeChangeUserData* as its parameter.
Once the connection to the server has been closed, the **ClientAPI** instance will be deleted and thus should not be used.

**Return Value**

Returns a pointer to an instance of **ClientAPI**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 4 | POLY_ECONNECT | Failed to connect to database. |

## ClientAPI::StartLogin

**static void StartLogin(**

**ClientAPI *client,**

**const char *username,**

**const char *password,**

**const void (*loginCallBack)(void *),**

**const void *userdata,**

**const char *env = 0}**

**C-callable Equivalent**

**void ClientAPI_StartLogin(**

**ClientAPI *client,**

**const char *username,**

**const char *password,**

**const void (*loginCallBack)(void *),**

**const void *userdata,**

**const char *env);**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI** |
| username | A pointer to a null terminated string containing the name of a user. |
| password | A pointer to a null terminated string containing a password. |
| loginCallBack | A pointer to a function that takes a void * as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |
| env | A pointer to a null terminated string. When data connection monitoring is enabled, this value appears in the *env* attribute of the *dataconnection* class (see *Utility Classes* manual). |

**Description**

This function attempts to log on to the server. This is only necessary when the server is an RTRDB that has security enabled.
The instance of **ClientAPI** pointed to by *client* must be connected. The name of a user and a password, specified by *username* and *password* respectively, are sent to the server for authentication. If security is disabled in the server, authentication will always succeed. If security is enabled in the server, it will check whether an entry exists in the *users* table with the identical name and password. If there is a record, authentication will succeeded; if not, it will fail. The server then replies with success or failure. On receiving this reply an error code is written to a private member variable in the

instance pointed to by the *client* and the callback function *loginCallback* is evoked with *userdata* as a parameter.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 18 | POLY_ELOGIN | Attempt to log in to server failed. |

## ClientAPI::StartLogout

**static void StartLogout (**

**ClientAPI *client,**

**void (*logoutCallback) (void *),**

**void *userdata) ;**

**C-callable Equivalent**

**void ClientAPI_StartLogout(**

**ClientAPI *client,**

**void (*logoutCallback) (void *)**

**void *userdata) ;**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI** |
| logoutCallback | A pointer to a function that takes callback as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |

**Description**

The function will issue a logout request. The callback function **logoutCallback** will be called, with **userdata** as its parameter, when the response is received by the client. The success or failure of the request can be determined in the call-back function by calling the Call-Back API function **ClientAPI::GetError.**

An accessor function is provided to read the private error code, where zero indicates the connection was accepted successfully, and a non-zero number indicates a specific failure.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 18 | POLY_ELOGIN | Attempt to log out from server failed. |

## ClientAPI::GetError

**static int GetError (**

**const ClientAPI *client,**

**char *buffer = 0,**

**int length = 0,**

**int *pNativeError = 0);**

**C-callable Equivalent**

**int ClientAPI_GetError (**

**const ClientAPI *client,**

**char *buffer,**

**int length,**

**int *pNativeError);**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI**. |
| buffer | A pointer to a buffer into which the error message will be copied. |
| length | An integer specifying the length of the buffer. |
| pNativeError | A pointer to an integer variable into which the error code will be copied. |

**Description**

This function returns the error code associated with the last operation performed on the client. It also allows the underlying error code and message generated by the database optionally to be obtained. If the operation was successful the value returned will be zero and the buffer, if any, supplied will not be modified.

Callback API error codes are listed in section 4.

If non-null, the *buffer* argument should point to a memory buffer with size in bytes specified by the *length* argument. The underlying database error message will then be copied into the buffer by the function.

If non-null the *pNativeError* argument must be the address of an *integer* variable into which the underlying database error code will be stored by the function.

Database error codes are listed in the Real-Time Relation Database user manual.

**Return Value**

The CallBack API error code associated with the last operation performed on the client.

## ClientAPI::DeleteClient

**static void DeleteClient (**

**ClientAPI *client,**

**const void (*deletedCallback)(void *),**

**const void *userdata);**

**C-callable Equivalent**

**void ClientAPI_DeleteClient (**

**ClientAPI *client,**

**const void (*deletedCallback)(void *),**

**const void *userdata);**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI** |
| deletedCallback | A pointer to a function that takes a void * as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |

**Description**

If the instance of **ClientAPI** pointed to by *client* is connected and there are no outstanding queries or transactions, then the client is disconnected.
If the instance pointed to by *client* is in the process of attempting to connect (via a **StartConnect** operation), then the **StartConnect** operation is aborted.
If the instance pointed to by *client* is connected and has any outstanding queries or transactions, then the client is not disconnected, and an error condition needs to be flagged.
When the required operations have been performed appropriate to the state of the instance pointed to by *client*, the error code is written to a private member variable in the instance pointed to by *client*, and the callback function *deletedCallback* is invoked with *userdata* as its parameter.
When the callback function finishes and returns control to the API, the **ClientAPI** destructor is invoked if the client is in a disconnected state.
The **ClientAPI** destructor is responsible for relinquishing all memory space allocated by the constructor.
An accessor function is provided to read the private error code, where zero indicates the connection was accepted successfully, and a non-zero number indicates a specific failure.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 14 | POLY_EINUSE | A client connection is in use and cannot be deleted. |

## ClientAPI::StartShutdown

**static void StartShutdown (**

**ClientAPI *client,**

**const void (*shutdownCallback)(void *),**

**const void *userdata);**

**C-callable Equivalent**

**void ClientAPI_StartShutdown (**

**ClientAPI *client,**

**const void (*shutdownCallback)(void *),**

**const void *userdata);**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI** |
| shutdownCallback | A pointer to a function that takes a void * as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |

**Description**

If the instance of **ClientAPI** pointed to by *client* is connected to a server then a shutdown operation is issued to the server which will then commence with the standard shutdown procedure.
When the required operations have been performed appropriate to the state of the instance pointed to by the *client*, an error code is written to a private member variable in the instance pointed to by the *client* and the callback function *shutdownCallback* is evoked with *userdata* as a parameter.
When the called-back function finishes and returns control to the API, the destructor is invoked if the database was shutdown successfully.
The **ClientAPI** destructor is responsible for relinquishing all memory space allocated by the constructor.
An accessor function is provided to read the private error code, where zero indicates the connection was accepted successfully, and a non-zero number indicates a specific failure.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 17 | POLY_ESHUTDOWN | The server could not be shutdown. |

## ClientAPI::SetOption

**static void SetOption (**

**ClientAPI *client,**

**int option,**

**int optionValue);**

**C-callable Equivalent**

**void ClientAPI_SetOption (**

**ClientAPI *client,**

**int option,**

**int optionValue);**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI** |
| option | An integer specifying the option to be set. |
| optionValue | An integer specifying the value to which the option is to be set. |

**Description**

This function sets the value of ClientAPI options. The *client* parameter refers to a specific connection for which the option is to be set. If *client* is null, the option is set globally for all client connections. Setting an option for a specific client connection overrides the global setting of that option. The following values for *option* are supported:

- POLY_FT_OPTION_RECONNECTION_RETRIES
  This specifies the number times to attempt to connect to each service during fail-over. The *optionValue* is interpreted as the number of retries.
- POLY_FT_OPTION_RECONNECTION_INTERVAL
  This specifies the interval between each attempt to connect to a service during fail-over. The *optionValue* is interpreted as the interval.
- POLY_FT_OPTION_RECONNECTION_TIMEOUT
  This specifies the time-out period, in milliseconds, associated with an attempt to connect to each service during fail-over. The *optionValue* is interpreted the connection time-out.
- POLY_FT_OPTION_HEALTHCHECK_INTERVAL
  This specifies the interval between heartbeat messages sent from the client to the RTRDB. The *optionValue* is interpreted as the number of milliseconds between each heartbeat message being sent.
- POLY_FT_OPTION_HEALTHCHECK_TIMEOUT
  This specifies the time-out period associated with each heartbeat message. Absence of a reply to a heartbeat message before this period has elapsed will result in client assuming the RTRDB has failed and instigating its fail-over procedure.
- POLY_FT_OPTION_KEEP_COPY
  This indicates whether the API should keep a copy of the data associated with an active query. Keeping a copy of the data allows deltas that occur as a result of fail-over to be calculated. A non-zero value for *optionValue* indicates a copy should be kept and zero that no copy should be kept.

- POLY_FT_OPTION_MAP_ROWIDS
  This indicates whether the API should maintain row ids for active queries across a fail-over. To do so the API must keep a copy of the primary key columns return by the query. A non-zero value for *optionValue* indicates that row ids should be maintained, zero that they should not be maintained. Note, setting POLY_FT_OPTION_KEEP_COPY implies POLY_FT_OPTION_MAP_ROWIDS.

## ClientAPI::GetOption

**static int GetOption(**

**ClientAPI *client,**

**int option);**

**C-callable Equivalent**

**int ClientAPI_GetOption(**

**ClientAPI *client,**

**int option);**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI** |
| option | An integer specifying the option to be set. |

**Description**

This function obtains the value of ClientAPI options. The *client* parameter refers to a specific connection for which the option is to be set. If *client* is null, the global setting of the option for all client connections is obtained. For a list of the possible options see the description of the function **ClientAPI::SetOption**.

**Return Value**

Returns the value of an option.

## ClientAPI::GetServiceName

**static void GetServiceName(**

**ClientAPI *client,**

**char *buffer,**

**int length);**

**C-callable Equivalent**

**void ClientAPI_GetServiceName(**

**ClientAPI *client,**

**char *buffer,**

**int length);**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI** |
| buffer | A pointer to a buffer into which the service name will be copied. |
| length | An integer specifying the length of the buffer. |

**Description**

This function obtains the name of the service to which the client connection is currently established. This is particularly useful when fault tolerance options are enabled on the connection and two alternative data services have been specified. The *client* parameter specifies the client connection from which the name is obtained. The name is copied into the buffer pointed to by the *buffer* parameter. The *length* parameter specifies the length of the buffer. Any name copied into the buffer will be truncated so as not to overwrite beyond the end of the buffer.

## ClientAPI::GetFTMode

**static int GetFTMode(**

**ClientAPI *client);**

**C-callable Equivalent**

**int ClientAPI_GetFTMode(**

**ClientAPI *client);**

**Parameters**

client A pointer to an instance of **ClientAPI**

**Description**

This function obtains the current fault tolerant mode of the service to which the client connection is currently established. This is used when fault tolerance options are enabled on the connection and two alternative data services have been specified. The *client* parameter specifies the client connection from which the name is obtained.

Possible return values and their meanings are:

| | |
|---|---|
| 0 | The server is running as the fault tolerant master without a standby server available |
| 1 | The server is running as the fault tolerant master with a standby server available. |
| 2 | The server is running as standby. |
| 3 | The server is running as a replica. |

## ClientAPI::GetCharacterSet

**static int GetCharacterSet(**

**ClientAPI *client,**

**int type);**

**C-callable Equivalent**

**int ClientAPI_GetCharacterSet(**

**ClientAPI *client,**

**int type);**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI** |
| type | An integer specifying the data type. |

**Description**

This function obtains the character set being used by the specified type. The type specified should be a character based data type. Currently the only valid value for *type* is POLY_TYPE_STRING. The *client* parameter refers to the specific connection for which the character set is configured.

**Return Value**

A code for the character set used by the specified type. Possible return values and their meanings are:

| | |
|---|---|
| 0 | The ASCII character set. |
| 1 | The UTF-8 character set. |

## ClientAPI::FailOver

**static void FailOver(**

**ClientAPI *client);**

**C-callable Equivalent**

**void ClientAPI_FailOver(**

**ClientAPI *client);**

**Parameters**

client A pointer to an instance of **ClientAPI**

**Description**

This function forces the connection to fail across to an alternative server. This is used when fault tolerance options are enabled on the connection and two alternative data services have been specified. The *client* parameter specifies the client connection from which the name is obtained.

## ClientAPI::EncryptPassword

**static int EncryptPassword(**

**const ClientAPI *client,**

**const char *password,**

**char *buffer,**

**int buffer_length,**

**int *buffer_length_ptr);**

**C-callable Equivalent**

**int ClientAPI_EncryptPassword(**

**const ClientAPI *client,**

**const char *password,**

**char *buffer,**

**int buffer_length,**

**int *buffer_length_ptr);**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI**. |
| password | A string containing the clear-text password to be encrypted. |
| buffer | A pointer to the buffer in which to return the encrypted password. |
| buffer_length | An integer specifying the length of the buffer. |
| buffer_length_ptr | A pointer to memory in which to return the total number of bytes (excluding the null-termination character) available to return in the buffer. |

**Description**

This function encrypts the *password* parameter using the encryption algorithm appropriate to the security settings of the specified connection. If successful, the encrypted version of the password is placed in the character buffer specified by the *buffer* parameter. The encrypted password can then be used to set the password for a new or existing user, by storing the value in the *password* column of the *users* table. The *Real-time Relation Database* manual explains password security.
If the client is not connected or if the encryption fails for some other reason the function returns non-zero.

**Return Value**

Returns zero for success and non-zero for failure.

# QueryAPI

This section specifies the public member functions of the **QueryAPI** class.

## QueryAPI::StartQuery

**static QueryAPI *StartQuery(**

**const ClientAPI *client,**

**const char *sqlText,**

**const void (*gotRow)(void *),**

**const void (*queryComplete)(void *),**

**const void *userdata,**

**const unsigned int maxRows = POLY_MAX_ROWS_UNLIMITED);**

**C-callable Equivalent**

**QueryAPI *QueryAPI_StartQuery(**

**const ClientAPI *client,**

**const char *sqlText,**

**const void (*gotRow)(void *),**

**const void (*queryComplete)(void *),**

**const void *userdata,**

**const unsigned int maxRows);**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI** which is connected to a server. |
| sqlText | A pointer to a null terminated string containing the SQL statement. |
| gotRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| queryComplete | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |
| maxRows | An optional unsigned integer specifying the maximum number of rows that may be returned by the query. |

**Description**

This function creates an instance of **QueryAPI** and attempts to execute the SQL statement in the string *sqlText* on a database that has a client identified by *client*. A pointer to the created **QueryAPI** instance is returned to the calling function.

For each row retrieved from the database by the query, subject to the optional limit *maxRows*, the callback function *gotRow* is invoked with *userdata* as its parameter. Before the callback function is called an error code is written to a private member variable in the instance pointed to by *client*.
An accessor function is provided to read the private error code, where zero indicates the row was got successfully, and a non-zero number indicates a specific failure.
When all rows have been retrieved by the query and the callback function for the last row has completed, the error code is written to the private member variable in the instance pointed to by *client* and the callback function *queryComplete* is invoked with the parameter *userdata*.
The error code written is zero to indicate that the query as completed successfully and some non-zero value to indicate a specific failure.
When the *queryComplete* callback function finishes the query is deleted.
The *maxRows* optional parameter can be used to limit the number of rows returned by the query. The default value is POLY_MAX_ROWS_UNLIMITED (0xFFFFFFFF or 4294967295), i.e. effectively no limitation.
The results are undefined if *sqlText* contains SQL DML or DDL rather than a query.

**Return Value**

Returns a pointer to an instance of **QueryAPI**.

**Error Code**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 9 | POLY_EQUERY | An SQL query has failed to execute. |

## QueryAPI::StartActiveQuery

```
static QueryAPI *StartActiveQuery(
                    const ClientAPI *client,
                    const char *sqlText,
                    const void (*insertedRow)(void *),
                    const void (*updatedRow)(void *),
                    const void (*deletedRow)(void *),
                    const void (*deltaComplete)(void *),
                    const void *userdata,
                    const unsigned int maxRows = POLY_MAX_ROWS_UNLIMITED,
                    const unsigned int secs = 0,
                    const unsigned int microsecs = 0);
```

**C-callable Equivalent**

```
QueryAPI *QueryAPI_StartActiveQuery(
                    const ClientAPI *client,
                    const char *sqlText,
                    const void (*insertedRow)(void *),
                    const void (*updatedRow)(void *),
                    const void (*deletedRow)(void *),
                    const void (*deltaComplete)(void *),
                    const void *userdata,
                    const unsigned int maxRows,
                    const unsigned int secs,
                    const unsigned int microsecs);
```

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI** which is connected to a server. |
| sqlText | A pointer to a null terminated string containing the SQL statement. |
| insertedRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| updatedRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| deletedRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| deltaComplete | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |
| maxRows | An optional unsigned integer specifying the maximum number of rows that may be returned by the query. |
| secs, *microsecs* | Optional unsigned integers specifying the minimum number of seconds and microseconds between deltas being sent by the server. The default values imply an ordinary active query. |

**Description**

This function creates an instance of **QueryAPI** as an active query, which monitors a database which has a client identified by client using the SQL statement specified by the string *sqlText*. A pointer to the created **QueryAPI** instance is returned to the calling function.
The first time the active query is issued the function invokes the callback function *insertedRow* with the parameter *userdata* to give the application the initial values for each row being monitored. When all rows have been presented using these callback functions, the callback function *deltaComplete* is invoked with *userdata* as its parameter.
Subsequently, the active query monitors the database for relevant changes. When a change is detected, the appropriate sets of callback functions are invoked. Before any callback function is invoked an error code is written to the private member variable in the instance of **QueryAPI**.
The callback function *insertedRow* is invoked with *userdata* when a row has been inserted.
The callback function *updatedRow* is invoked with *userdata* when a row has been changed. Only the changed columns are available to the application to read.
The callback function *deletedRow* is invoked with *userdata* when a row has been deleted.
When all the application has been notified of all changes, an error code is written to the private member variable in the **QueryAPI** instance and the *deltaComplete* callback function is invoked with *userdata* as its parameter.
The *maxRows* optional parameter can be used to limit the number of rows returned by the query. Using *maxRows* ensures the client will never be informed of more than *maxRows* rows. Furthermore, the client will always be informed of *maxRows* rows as long as sufficient rows are found to satisfy the query.
The optional *secs* and *microsecs* parameters are used to reduce the processing load on the server and the network traffic between server and client. Together they specify a minimum interval between deltas being sent by the server. See *The Real-Time Relational Database* for more details.
An accessor function is provided to read the private error code, where zero indicates the row was got successfully, and a non-zero number indicates a specific failure.
The error code written before any operation is zero indicate that the operation was completed successfully , and some non-zero value to indicate a specific failure.

**Return Value**

Returns a pointer to an instance of **QueryAPI**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 9 | POLY_EQUERY | An SQL query has failed to execute. |
| 12 | POLY_EDELTA | The delta for an active query has failed. |

## QueryAPI::StartObjectQuery

**static QueryAPI *StartObjectQuery(**

**const ClientAPI *client,**

**const char *tableName,**

**const void (*gotRow)(void *),**

**const void (*queryComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(QueryAPI *, void *));**

**C-callable Equivalent**

**QueryAPI *QueryAPI_StartObjectQuery(**

**const ClientAPI *client,**

**const char *tableName,**

**const void (*gotRow)(void *),**

**const void (*queryComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(QueryAPI *, void *));**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI** which is connected to a server. |
| tableName | A pointer to a null terminated string containing the (case sensitive) name of a database table or view. |
| gotRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| queryComplete | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |
| collectArgs | A pointer to a function that takes a pointer to an instance of **QueryAPI** and a pointer to a void as its parameters and returns nothing. |

**Description**

This function creates an instance of **QueryAPI** and attempts to execute an object query on the table *tableName* on a database that has a client identified by *client*. A pointer to the created **QueryAPI** instance is returned to the calling function.
Before the query is dispatched to the database the callback function *collectArgs* is invoked to collect any parameters and column names associated with the query. This is done by calling the **QueryAPI::AddName** and **QueryAPI::AddArg** functions. Note that the function *collectArgs* will also be called when a query is recovered after a fail-over.
For the row retrieved from the database by the query, the callback function *gotRow* is invoked with *userdata* as its parameter. Before the callback function is called an error code is written to a private member variable in the instance pointed to by *client*.
An accessor function is provided to read the private error code, where zero indicates the row was got successfully, and a non-zero number indicates a specific failure.
When the row has been retrieved by the query and the callback function for the row has completed, the error code is written to the private member variable in the instance pointed to by *client* and the callback function *queryComplete* is invoked with the parameter *userdata*.
The error code written is zero to indicate that the query as completed successfully and some non-zero value to indicate a specific failure.
When the *queryComplete* callback function finishes and returns control to the API, the destructor is called to delete this instance of **QueryAPI**. It is the responsibility of the destructor to relinquish all memory space allocated by the constructor of this class.

**Return Value**

Returns a pointer to an instance of **QueryAPI**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 9 | POLY_EQUERY | An SQL query has failed to execute. |

## QueryAPI::StartActiveObjectQuery

```
static QueryAPI *StartActiveObjectQuery(
                        const ClientAPI *client,
                        const char *tableName,
                        const void (*insertedRow)(void *),
                        const void (*updatedRow)(void *),
                        const void (*deletedRow)(void *),
                        const void (*deltaComplete)(void *),
                        const void *userdata,
                        const void (*collectArgs)(QueryAPI *, void *),
                        const unsigned int secs = 0,
                        const unsigned int microsecs = 0);
```

**C-callable Equivalent**

```
QueryAPI *QueryAPI_StartActiveObjectQuery(
                        const ClientAPI *client,
                        const char *tableName,
                        const void (*insertedRow)(void *),
                        const void (*updatedRow)(void *),
                        const void (*deletedRow)(void *),
                        const void (*deltaComplete)(void *),
                        const void *userdata,
                        const void (*collectArgs)(QueryAPI *, void *),
                        const unsigned int secs,
                        const unsigned int microsecs);
```

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI** which is connected to a server. |
| tableName | A pointer to a null terminated string containing the (case sensitive) name of a database table or view. |
| insertedRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| updatedRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| deletedRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| deltaComplete | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |
| collectArgs | A pointer to a function that takes a pointer to an instance of **QueryAPI** and a pointer to a void as its parameters and returns nothing. |
| secs, *microsecs* | Optional unsigned integers specifying the minimum number of seconds and microseconds between deltas being sent by the server. The default values imply an ordinary active query. |

**Description**

This function creates an instance of **QueryAPI** as an active object query which monitors a database which has a client identified by *client* using an object query on the table specified by the string *tableName*. A pointer to the created **QueryAPI** instance is returned to the calling function.
Before the query is dispatched to the database the callback function *collectArgs* is invoked to collect any parameters and column names associated with the query. This is done by calling the **QueryAPI::AddName** and **QueryAPI::AddArg** functions. Note that the function *collectArgs* will also be called when a query is recovered after a fail-over.
The first time the active query is issued the function invokes the callback function *insertedRow* with the parameter *userdata* to give the application the initial values for the row being monitored. When the row has been presented using these callback functions, the callback function *deltaComplete* is invoked with *userdata* as its parameter.
Subsequently, the active query monitors the database for relevant changes. When a change is detected, the appropriate set of callback functions is invoked. Before any callback function is invoked an error code is written to the private member variable in the instance of **QueryAPI**.
The callback function *insertedRow* is invoked with *userdata* when a row has been inserted.
The callback function *updatedRow* is invoked with *userdata* when a row has been changed. Only the changed columns are available to the application to read.
The callback function *deletedRow* is invoked with *userdata* when a row has been deleted.
When all the application has been notified of all changes, an error code is written to the private member variable in the **QueryAPI** instance and the *deltaComplete* callback function is invoked with *userdata* as its parameter.
The optional *secs* and *microsecs* parameters are used to reduce the processing load on the server and the network traffic between server and client. Together they specify a minimum interval between deltas being sent by the server. See *The Real-Time Relational Database* for more details.
An accessor function is provided to read the private error code, where zero indicates the row was got successfully, and a non-zero number indicates a specific failure.
The error code written before any operation is zero indicate that the operation was completed successfully , and some non-zero value to indicate a specific failure.

**Return Value**

Returns a pointer to an instance of **QueryAPI**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 9 | POLY_EQUERY | An SQL query has failed to execute. |
| 12 | POLY_EDELTA | The delta for an active query has failed. |

## QueryAPI::StartProcedureQuery

**static QueryAPI *StartProcedureQuery(**

**const ClientAPI *client,**

**const char *procedureName,**

**const void (*gotRow)(void *),**

**const void (*queryComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(QueryAPI *, void *),**

**const unsigned int maxRows = POLY_MAX_ROWS_UNLIMITED);**

**C-callable Equivalent**

**QueryAPI *QueryAPI_StartProcedureQuery(**

**const ClientAPI *client,**

**const char *procedureName,**

**const void (*gotRow)(void *),**

**const void (*queryComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(QueryAPI *, void *),**

**const unsigned int maxRows);**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI** which is connected to a server. |
| procedureName | A pointer to a null terminated string containing the (case sensitive) name of a database procedure. |
| gotRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| queryComplete | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |
| collectArgs | A pointer to a function that takes a pointer to an instance of **QueryAPI** and a pointer to a void as its parameters and returns nothing. |
| maxRows | An optional unsigned integer specifying the maximum number of rows that may be returned by the query. |

**Description**

This function creates an instance of **QueryAPI** and attempts to execute an SQL procedure query specified by *procedureName* on a database that has a client identified by *client*. A pointer to the created **QueryAPI** instance is returned to the calling function.
Before the query is dispatched to the database the callback function *collectArgs* is invoked to collect any parameters associated with the query. This is done by calling the **QueryAPI::AddArg** function. Note that the function *collectArgs* will also be called when a query is recovered after a fail-over.
For the row retrieved from the database by the query, the callback function *gotRow* is invoked with *userdata* as its parameter. Before the callback function is called an error code is written to a private member variable in the instance pointed to by *client*.
An accessor function is provided to read the private error code, where zero indicates the row was got successfully, and a non-zero number indicates a specific failure.
When the row has been retrieved by the query and the callback function for the row has completed, the error code is written to the private member variable in the instance pointed to by *client* and the callback function *queryComplete* is invoked with the parameter *userdata*.
The error code written is zero to indicate that the query as completed successfully and some non-zero value to indicate a specific failure.
When the *queryComplete* callback function finishes and returns control to the API, the destructor is called to delete this instance of **QueryAPI**. It is the responsibility of the destructor to relinquish all memory space allocated by the constructor of this class.
The *maxRows* optional parameter can be used to limit the number of rows returned by the query. The default value is POLY_MAX_ROWS_UNLIMITED (0xFFFFFFFF or 4294967295), i.e. effectively no limitation

**Return Value**

Returns a pointer to an instance of **QueryAPI**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 9 | POLY_EQUERY | An SQL query has failed to execute. |

## QueryAPI::StartActiveProcedureQuery

```
static QueryAPI *StartActiveProcedureQuery(
                        const ClientAPI *client,
                        const char *procedureName,
                        const void (*insertedRow)(void *),
                        const void (*updatedRow)(void *),
                        const void (*deletedRow)(void *),
                        const void (*deltaComplete)(void *),
                        const void *userdata,
                        const void (*collectArgs)(QueryAPI *, void *),
                        const unsigned int maxRows = POLY_MAX_ROWS_UNLIMITED,
                        const unsigned int secs = 0,
                        const unsigned int microsecs = 0);
```

**C-callable Equivalent**

```
QueryAPI *QueryAPI_StartActiveProcedureQuery(
                        const ClientAPI *client,
                        const char *procedureName,
                        const void (*insertedRow)(void *),
                        const void (*updatedRow)(void *),
                        const void (*deletedRow)(void *),
                        const void (*deltaComplete)(void *),
                        const void *userdata,
                        const void (*collectArgs)(QueryAPI *, void *),
                        const unsigned int maxRows,
                        const unsigned int secs,
                        const unsigned int microsecs);
```

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI** which is connected to a server. |
| procedureName | A pointer to a null terminated string containing the (case sensitive) name of a database procedure. |
| insertedRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| updatedRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| deletedRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| deltaComplete | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |
| collectArgs | A pointer to a function that takes a pointer to an instance of **QueryAPI** and a pointer to a void as its parameters and returns nothing. |
| microsecs | An unsigned integer specifying the minimum number of microseconds between deltas being sent by the server. |
| maxRows | An optional unsigned integer specifying the maximum number of rows that may be returned by the query. |
| secs, *microsecs* | Optional unsigned integers specifying the minimum number of seconds and microseconds between deltas being sent by the server. The default values imply an ordinary active query. |

**Description**

This function creates an instance of **QueryAPI** as an active SQL procedure query, which monitors a database which has a client identified by *client* using an SQL procedure query specified by the string *procedureName*. A pointer to the created **QueryAPI** instance is returned to the calling function.
Before the query is dispatched to the database the callback function *collectArgs* is invoked to collect any parameters associated with the query. This is done by calling the **QueryAPI::AddArg** function. Note that the function *collectArgs* will also be called when a query is recovered after a fail-over.
The first time the active query is issued the function invokes the callback function *insertedRow* with the parameter *userdata* to give the application the initial values for the row being monitored. When the row has been presented using these callback functions, the callback function *deltaComplete* is invoked with *userdata* as its parameter.
Subsequently, the active query monitors the database for relevant changes. When a change is detected, the appropriate sets of callback functions are invoked. Before any callback function is invoked an error code is written to the private member variable in the instance of **QueryAPI**.
The callback function *insertedRow* is invoked with *userdata* when a row has been inserted.
The callback function *updatedRow* is invoked with *userdata* when a row has been changed. Only the changed columns are available to the application to read.
The callback function *deletedRow* is invoked with *userdata* when a row has been deleted.
When all the application has been notified of all changes, an error code is written to the private member variable in the **QueryAPI** instance and the *deltaComplete* callback function is invoked with *userdata* as its parameter.
The *maxRows* optional parameter can be used to limit the number of rows returned by the query. Using *maxRows* ensures the client will never be informed of more than *maxRows* rows. Furthermore, the client will always be informed of *maxRows* rows as long as sufficient rows are found to satisfy the query.

The optional *secs* and *microsecs* parameters are used to reduce the processing load on the server and the network traffic between server and client. Together they specify a minimum interval between deltas being sent by the server. See *The Real-Time Relational Database* manual for more details.
An accessor function is provided to read the private error code, where zero indicates the row was got successfully, and a non-zero number indicates a specific failure.
The error code written before any operation is zero indicate that the operation was completed successfully , and some non-zero value to indicate a specific failure.

**Return Value**

Returns a pointer to an instance of **QueryAPI**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 9 | POLY_EQUERY | An SQL query has failed to execute. |
| 12 | POLY_EDELTA | The delta for an active query has failed. |

## QueryAPI::AddName

**static int AddName(**

**const QueryAPI *query,**

**const char *name);**

**C-callable Equivalent**

**int QueryAPI_AddName(**

**const QueryAPI *query,**

**const char *name);**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI**. |
| name | A pointer to a null terminated string specifying the (case sensitive) name of a column to be retrieved. |

**Description**

This function specifies the name of a column to be retrieved from the database by an object query. The name is specified by the *name* parameter.
This function can only be called inside the callback function for collecting arguments. It must be called for each column required in the output. All calls to it from inside a *CollectArgs* callback function must be made before any calls to the **QueryAPI::AddArg** function.
A return value of 0 indicates success and non-zero indicates failure. The function writes an error code to the private member variable of the **QueryAPI** instance passed to it.

**Return Value**

Returns zero for success and non-zero for failure.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |

## QueryAPI::AddArg

**static int AddArg (**

**const QueryAPI *query,**

**const char *name,**

**int type,**

**const void *buffer,**

**int bufferLength);**

**C-callable Equivalent**

**int QueryAPI_AddArg (**

**const QueryAPI *query,**

**const char *name,**

**int type,**

**const void *buffer,**

**int bufferLength);**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI**. |
| name | A pointer to a null terminated string specifying the (case sensitive) name of the argument. A null pointer specifies an unnamed positional parameter. |
| type | A integer specifying the type of the argument. |
| buffer | A pointer to a buffer containing the value of the argument. A null pointer means that a value of NULL is to be used. |
| bufferLength | A integer specifying the length of the buffer. The value is ignored if *buffer* is null. |

**Description**

This function specifies a query argument called *name*. The type of the argument is specified by the *type* parameter. The value of the argument is passed in the buffer pointed to by the *buffer* parameter. The length of the buffer is specified by the *bufferLength* parameter. If a non-null *name* is supplied, the query argument will be referenced by ":<type>name" in the associated SQL text of the query or command. The named parameter may be referenced any number of times. If a null *name* parameter is supplied, the query argument will be referenced by "?" in the associated SQL text of the query or command. The positional arguments specified by calls to *QueryAPI::AddArg* will be referenced sequentially with "?" markers in the text. e.g. "select id from customer where name = ? and status = ?" Positional parameters may not be used for object queries.
This function can only be called inside the callback function for collecting arguments. It must be called for each argument to the query. All calls to it from inside a *CollectArgs* callback function must be made after any calls to the **QueryAPI::AddName** function.
A return value of 0 indicates success and non-zero indicates failure. The function writes an error code to the private member variable of the **QueryAPI** instance passed to it.
Section 5 covers the possible values for the type parameter.

**Return Value**

Returns zero for success and non-zero for failure.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |

## QueryAPI::GetRowId

**static long GetRowId (**

**const QueryAPI *query);**

**C-callable Equivalent**

**long QueryAPI_GetRowId (**

**const QueryAPI *query);**

**Parameters**

query A pointer to an instance of **QueryAPI**

**Description**

This function returns an identification number for the current row of data.
It can only be validly called from within a *gotRow*, *insertedRow*, *updatedRow* or *deletedRow* callback function, as this is the only time when a row of data is available.
The function writes an error code to the private member variable of the **QueryAPI** instance passed to it.
An accessor function is provided to read the private error code, where zero indicates the row identification number is valid, and a non-zero number indicates a specific failure.

**Return Value**

Returns a row id.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |

## QueryAPI::GetClientRowId

**static long GetClientRowId (**

**const QueryAPI const*query);**

**C-callable Equivalent**

**long QueryAPI_GetClientRowId (**

**const QueryAPI const*query);**

**Parameters**

query A pointer to an instance of **QueryAPI**

**Description**

This functions returns the user supplied client row id for the current row of data or zero if the row is not one inserted through the active query. It can only be validly called from within a *gotRow*, *insertedRow*, *updatedRow* or *deletedRow* callback function, as this is the only time when a row of data is available. The client application can use this function to identify rows it has inserted and then obtain the server allocated row id for those rows.

**Return Value**

Returns a client row id.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |

## QueryAPI::CopyNColumn

**static int CopyNColumn(**

**const QueryAPI *query,**

**const int columnNumber,**

**void *variable,**

**int bytesToCopy);**

**C-callable Equivalent**

**int QueryAPI_CopyNColumn(**

**const QueryAPI *query,**

**const int columnNumber,**

**void *variable,**

**int bytesToCopy);**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI** |
| columnNumber | The ordinal value of the column number, between 1 and n, where n is the number of columns retrieved by the query. |
| variable | A pointer to a variable of any type. |
| bytesToCopy | An integer specifying the number of bytes to copy. |

**Description**

This function copies the data in the column indicated by *columnNumber* to the variable pointed to by *variable*. The function copies no more than *bytesToCopy* to the destination. If the number of bytes available is larger than *bytesToCopy* an error code is also set to indicate that the data returned has been truncated. In the case of a string this truncated data will not be null terminated.
If used in the context of an active query and the column is not available (because the column in question was not changed), then no bytes are copied.
This function can only be called inside the callback function for got row (for a fixed query) or the callback functions for insert row and update row (for an active query), as this is the only time when data is available.

The function **QueryAPI::GetError** is provided to read the error code, where zero indicates that the column was copied correctly, and some non-zero number indicates a specific failure.

**Return Value**

Returns the number of bytes copied (even when truncated) or zero if it fails.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 6 | POLY_ECOLUMNNO | Invalid column number. |
| 7 | POLY_ENOVALUE | A column value is unavailable. |
| 8 | POLY_ENULLVALUE | A column value is null. |
| 13 | POLY_ETRUNCATED | The data requested for a column has been truncated. |

## QueryAPI::CopyColumn

**static int CopyColumn(**

**const QueryAPI *query,**

**const char *columnName,**

**void *variable,**

**int bytesToCopy);**

**C-callable Equivalent**

**int QueryAPI_CopyColumn(**

**const QueryAPI *query,**

**const char *columnName,**

**void *variable,**

**int bytesToCopy);**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI.** |
| columnName | A pointer to a null terminated string containing the (case sensitive) name of a column retrieved by the query. |
| variable | A pointer to a variable of any type. |
| bytesToCopy | An integer specifying the number of bytes to copy. |

**Description**

This function copies the data in the column indicated by *columnName* to the variable pointed to by *variable*. The function copies no more than *bytesToCopy* to the destination. If the number of bytes available is larger than *bytesToCopy* an error code is also set to indicate that the data returned has been truncated. In the case of a string this truncated data will not be null terminated.
If used in the context of an active query and the column is not available (because the column in question was not changed), then no bytes are copied.

This function can only be called inside the callback function for got row (for a fixed query) or the callback functions for insert row and update row (for an active query), as this is the only time when data is available.
The function **QueryAPI::GetError** is provided to read the error code, where zero indicates that the column was copied correctly, and some non-zero number indicates a specific failure.

**Return Value**

Returns the number of bytes copied (even when truncated) or zero if it fails.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 5 | POLY_ECOLUMNNAME | Invalid column name. |
| 7 | POLY_ENOVALUE | A column value is unavailable. |
| 8 | POLY_ENULLVALUE | A column value is null. |
| 13 | POLY_ETRUNCATED | The data requested for a column has been truncated. |
| 16 | POLY_NOINFO | Information requested for a query before it has returned. |

## QueryAPI::CopyColumnToString

**static int CopyColumnToString(**

**const QueryAPI *query,**

**const int columnNumber,**

**char *buffer,**

**int bufferLength);**

**C-callable Equivalent**

**int QueryAPI_CopyColumnToString(**

**const QueryAPI *query,**

**const int columnNumber,**

**char *buffer,**

**int bufferLength);**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI** |
| columnNumber | The ordinal value of the column number, between 1 and n, where n is the number of columns retrieved by the query. |
| buffer | A pointer to a buffer to receive the value. |
| bufferLength | An integer specifying the number of bytes to copy. |

**Description**

This function copies a string representation of the data in the column indicated by *columnNumber* to the buffer pointed to by *buffer*. The function copies no more than *bufferLength* to the destination. If the number of bytes available is larger than *bufferLength* an error code is also set to indicate that the data returned has been truncated. If truncation occurs *buffer* will not be null terminated.
If used in the context of an active query and the column is not available (because the column in question was not changed), then no characters are copied.

This function can only be called inside the callback function for got row (for a fixed query) or the callback functions for insert row and update row (for an active query), as this is the only time when data is available.
The function **QueryAPI::GetError** is provided to read the error code, where zero indicates that the column was copied correctly, and some non-zero number indicates a specific failure.

**Return Value**

Returns the number of bytes copied (even when truncated) or zero if it fails.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 6 | POLY_ECOLUMNNO | Invalid column number. |
| 7 | POLY_ENOVALUE | A column value is unavailable. |
| 8 | POLY_ENULLVALUE | A column value is null. |
| 13 | POLY_ETRUNCATED | The data requested for a column has been truncated. |
| 16 | POLY_ENONINFO | Information requested for a query before it has returned. |

## QueryAPI::GetColNum

**static int GetColNum(**

**const QueryAPI *query,**

**const char *columnName);**

**C-callable Equivalent**

**int QueryAPIGetColNum(**

**const QueryAPI *query,**

**const char *columnName);**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI.** |
| columnName | A pointer to a null terminated string containing the (case sensitive) name of a column retrieved by the query. |

**Description**

This function takes the column name pointed to by *columnName* and returns the ordinal number of the column. The number returned is between **1** and **n**, where n is the number of columns returned by the query. If the named column could not be found, the function returns a zero.
This function can be successfully called at any time after the first callback function has been invoked.

**Return Value**

Returns a column number.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 5 | POLY_ECOLUMNNAME | Invalid column name. |
| 16 | POLY_ENOINFO | Information requested for a query before it has returned. |

## QueryAPI::GetColumnCount

**static int GetColumnCount(**

**const QueryAPI *query);**

**C-callable Equivalent**

**int QueryAPI_GetColumnCount(**

**const QueryAPI *query);**

**Parameters**

query A pointer to an instance of **QueryAPI.**

**Description**

This function returns the number of columns retrieved by the query. This function can be successfully called at any time after the first callback function has been invoked.

**Return Value**

Returns the number of columns

## QueryAPI::GetColumnLength

**static int GetColumnLength(**

**const QueryAPI *query,**

**const int columnNumber);**

**C-callable Equivalent**

**int QueryAPI_GetColumnLength(**

**const QueryAPI *query,**

**const int columnNumber);**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI.** |
| columnNumber | The ordinal value of the column number, between 1 and n, where n is the number of columns retrieved by the query. |

**Description**

This function obtains the length in bytes of the value of the column specified by the *columnNumber* parameter. This function can be successfully called at any time after the first callback function has been invoked.

**Return Value**

A non-negative value represents the length of the data. A return value of -1 indicates an error. A return value of 0 indicates that the column is null.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 6 | POLY_ECOLUMNNO | Invalid column number. |
| 7 | POLY_ENOVALUE | A column value is unavailable. |
| 8 | POLY_ENULLVALUE | A column value is null. |
| 16 | POLY_ENOINFO | Information requested for a query before it has returned. |

## QueryAPI::GetColumnName

**static const char *GetColumnName(**

**const QueryAPI *query,**

**const int columnNumber);**

**C-callable Equivalent**

**const char *QueryAPI_GetColumnName(**

**const QueryAPI *query,**

**const int columnNumber);**

### Parameters

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI.** |
| columnNumber | The ordinal value of the column number, between 1 and n, where n is the number of columns retrieved by the query. |

### Description

This function obtains the name of the column specified by the *columnNumber* parameter.
This function can be successfully called at any time after the first callback function has been invoked.

### Return Value

Returns a pointer to the column name

## QueryAPI::GetColumnType

**static int GetColumnType(**

**const QueryAPI *query,**

**const int columnNumber);**

**C-callable Equivalent**

**int QueryAPI_GetColumnType(**

**const QueryAPI *query,**

**const int columnNumber**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI.** |
| columnNumber | The ordinal value of the column number, between 1 and n, where n is the number of columns retrieved by the query. |

**Description**

This function obtains the type of the column specified by the *columnNumber* parameter.
This function can be successfully called at any time after the first callback function has been invoked.
The possible type values are given in section 5.

**Return Value**

Returns the column type.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 6 | POLY_ECOLUMNNO | Invalid column number. |
| 16 | POLY_ENOINFO | Information requested for a query before it has returned. |

## QueryAPI::GetColumnFlags

**static int GetColumnFlags(**

**const QueryAPI *query,**

**const int columnNumber);**

**C-callable Equivalent**

**int QueryAPI_GetColumnFlags(**

**const QueryAPI *query,**

**const int columnNumber**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI.** |
| columnNumber | The ordinal value of the column number, between 1 and n, where n is the number of columns retrieved by the query. |

**Description**

This function obtains the flags associated with the column specified by the *columnNumber* parameter. Flags indicate whether the column is a primary key, nullable and updatable.
This function can be successfully called at any time after the first callback function has been invoked.
The possible type values are given in section 6.

**Return Value**

Returns the column flags.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 6 | POLY_ECOLUMNNO | Invalid column number. |
| 16 | POLY_ENOINFO | Information requested for a query before it has returned. |

## QueryAPI::GetColumnChange

**static int GetColumnChange(**

**const QueryAPI *query,**

**const int columnNumber);**

**C-callable Equivalent**

**int QueryAPI_GetColumnChange(**

**const QueryAPI *query,**

**const int columnNumber );**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI.** |
| columnNumber | The ordinal value of the column number, between 1 and n, where n is the number of columns retrieved by the query. |

**Description**

This function indicates whether the value for a specified column has changed. This is independent of whether the value is available. The function returns a value of 1 if the value has changed and 0 if the value has remained unchanged.

This function can only be called inside the callback function for got row (for a fixed query) or the callback functions for insert row and update row (for an active query), as this is the only time when data may be available.

An accessor function is provided to read the error code, where zero indicates that the column has changed, and some non-zero number indicates a specific reason why the column is unchanged.

**Return Value**

Returns the number 1 if the value has changed, and 0 if the value remains unchanged.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 7 | POLY_ENOVALUE | A column value has not changed and is not available. |
| 16 | POLY_ENOINFO | Information requested for a query before it has returned. |
| 24 | POLY_ENOCHANGE | A column value has not changed but is available. |

## QueryAPI::InsertColumn

**static int InsertColumn(**

**const QueryAPI *query,**

**long clientRowId,**

**int columnNumber,**

**void *buffer,**

**int length);**

**C-callable Equivalent**

**int QueryAPI_InsertColumn(**

**const QueryAPI *query,**

**long clientRowId,**

**int columnNumber,**

**void *buffer,**

**int length);**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI** which this insert is to performed on. |
| clientRowId | The client's row id of the record to be inserted. |
| columnNumber | The column number of the record to be inserted (1 indicates the first field of the record). |
| buffer | A pointer to the data to be inserted in the *columnNumber* and client*RowId* defined above. A null pointer means that a value of NULL is to be used. |
| length | The length of *buffer*. The value is ignored if *buffer* is null. |

**Description**

This function may be called in the *UpdateCallback* of the **TransAPI::Commit** function. Used in groups, one call for each column of each row to be inserted, this function inserts a value to the given *clientRowId* and *columnNumber.*
The **QueryAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

**Return Value**

Returns 0 if no error occurred, non-zero otherwise. If there was an error, its code can be obtained by calling **QueryAPI::GetError**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 16 | POLY_ENOINFO | Query has not yet returned. |
| 6 | POLY_ECOLUMNNO | *columnNumber* provided is out of range. |
| 23 | POLY_ENOTINCOMMIT | **TransAPI::Commit** has not been called. |

## QueryAPI::UpdateColumn

**static int UpdateColumn(**

**const QueryAPI *query,**

**long rowId,**

**int columnNumber,**

**void *buffer,**

**int length);**

**C-callable Equivalent**

**int QueryAPI_UpdateColumn(**

**const QueryAPI *query,**

**long rowId,**

**int columnNumber,**

**void *buffer,**

**int length);**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI** which this update is to performed on. |
| rowId | The row id of the record to be updated. |
| columnNumber | The column number of the record to be updated (1 indicates the first field of the record). |
| buffer | A pointer to the data to be updated in the *columnNumber* and *rowId* defined above. A null pointer means that a value of NULL is to be used |
| length | The length of *buffer*. The value is ignored if *buffer* is null. |

**Description**

This function may be called in the *UpdateCallback* of the commit. This function updates a value to the given r*owId* and *columnNumber*.
When the operation is complete, the error code is written to a private member variable, and the function pointed to by *callback* is invoked on the *userdata* instance.
The **QueryAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

**Return Value**

Returns 0 if no error occurred, non-zero otherwise. If there was an error, its code can be obtained by calling **QueryAPI::GetError**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 16 | POLY_ENOINFO | Query has not yet returned. |
| 6 | POLY_ECOLUMNNO | *columnNumber* provided is out of range. |
| 23 | POLY_ENOTINCOMMIT | **TransAPI::Commit** has not been called. |

## QueryAPI::DeleteRow

**static int DeleteRow(**

**const QueryAPI *query,**

**long rowId);**

**C-callable Equivalent**

**int QueryAPI_DeleteRow(**

**const QueryAPI *query,**

**long rowId);**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI** which this delete is to performed on. |
| rowId | The row id of the record to be deleted. |

**Description**

This function may be called in the *UpdateCallback* of the commit. It causes the record specified by the *rowId* parameter to be deleted.
When the operation is complete, the error code is written to a private member variable, and the function pointed to by *callback* is invoked on the *userdata* instance.
The **QueryAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

**Return Value**

Returns 0 if no error occurred, non-zero otherwise. If there was an error, its code can be obtained by calling **QueryAPI::GetError**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 16 | POLY_ENOINFO | Query has not yet returned. |
| 23 | POLY_ENOTINCOMMIT | **TransAPI::Commit** has not been called. |

## QueryAPI::StartAbort

**static void StartAbort(**

**QueryAPI *query,**

**const void (*queryAborted)(void *),**

**const void *userdata);**

**C-callable Equivalent**

**void QueryAPI_StartAbort(**

**QueryAPI *query,**

**const void (*queryAborted)(void *),**

**const void *userdata);**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI.** |
| queryAborted | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |

**Description**

This function aborts the query pointed to by *query*. After the query is aborted (or if there is no query in progress), an error code is written to the private error member variable. Then, if *queryAborted* is not NULL, the callback function pointed to by *queryAborted* is invoked with *userdata*.

If the query is successfully aborted then after any *queryAborted* function finishes and providing the query is not an instance of **CommandAPI**, the query is deleted. If the query is a command, i.e. it is an instance of **CommandAPI**, it is not deleted. Instead the command is returned to its pre-execution state, either prepared or not.

An accessor function is provided to read the private error code, where zero indicates that the query was aborted successfully, and some non-zero number indicates a specific failure.

This function should only be used to abort an active query and only after the first delta has been successfully received.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 19 | POLY_EQUERYINUSE | Query is added to a transaction. |

## QueryAPI::GetError

**static int GetError (**

**const QueryAPI *query,**

**char *buffer = 0,**

**int length = 0,**

**int *pNativeError = 0);**

**C-callable Equivalent**

**int QueryAPI_GetError (**

**const QueryAPI *query,**

**char *buffer,**

**int length,**

**int *pNativeError);**

**Parameters**

| | |
|---|---|
| query | A pointer to an instance of **QueryAPI**. |
| buffer | A pointer to a buffer into which the error message will be copied. |
| length | An integer specifying the length of the buffer. |
| pNativeError | A pointer to an integer variable into which the error code will be copied. |

**Description**

This function returns the error code associated with the last operation performed on the query or command. It also allows the underlying error code and message generated by the database optionally to be obtained. If the operation was successful the value returned will be zero and the buffer, if any, supplied will not be modified.

Callback API error codes are listed in section 4.

If non-null, the *buffer* argument should point to a memory buffer with size in bytes specified by the *length* argument. The underlying database error message will then be copied into the buffer by the function.

If non-null the *pNativeError* argument must be the address of an *integer* variable into which the underlying database error code will be stored by the function.

Database error codes are listed in the Real-Time Relation Database user manual.

**Return Value**

The CallBack API error code associated with the last operation performed on the query or command.

# CommandAPI

This section specifies the public member functions of the **CommandAPI** class, which is derived from the **QueryAPI** class.

## CommandAPI::CreateCommand

**static CommandAPI *CreateCommand(**

**const ClientAPI *client,**

**const char *sqlText);**

**C-callable Equivalent**

**CommandAPI *CommandAPI_CreateCommand(**

**const ClientAPI *client,**

**const char *sqlText);**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI.** |
| sqlText | A pointer to a null terminated string containing the SQL statement. |

**Description**

This function creates an instance of **CommandAPI** on a database which has a client identified by *client* using the supplied SQL text. A pointer to the created **CommandAPI** instance is returned.

**Return Value**

A pointer to an instance of **CommandAPI**.

## CommandAPI::StartPrepare

**static int StartPrepare(**

**CommandAPI *command,**

**const void (*prepareCallback)(void *),**

**const void *userdata);**

**C-callable Equivalent**

**int CommandAPI_StartPrepare(**

**CommandAPI *command,**

**const void (*prepareCallback)(void *),**

**const void *userdata);**

**Parameters**

| | |
|---|---|
| command | A pointer to an instance of **CommandAPI** which is to be prepared. |
| prepareCallback | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the call-back function. The pointer is of type void *, which allows any type of data or object to be passed into the call-back function. |

**Description**

This function attempts to prepare a command that was previously created by calling **CommandAPI::CreateCommand**. Preparing a command involves compilation and optimisation of the SQL statement associated with the command. When a command has been prepared it is more efficient to execute.
When the operation is complete, the error code is written to a private member variable, and the function pointed to by *prepareCallback* is invoked on the *userdata* instance.
The **QueryAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

**Return Value**

Returns 0 if no error occurred, non-zero otherwise. If there was an error, its code can be obtained by calling **QueryAPI::GetError**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 27 | POLY_ECOMMANDINUSE | The command is already prepared, executing as a query or added to a transaction. |

## CommandAPI::StartQuery

**static int *StartQuery(**

**CommandAPI *command,**

**const void (*gotRow)(void *),**

**const void (*queryComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(QueryAPI *, void *));**

**const unsigned int maxRows = POLY_MAX_ROWS_UNLIMITED);**

**C-callable Equivalent**

**int CommandAPI_StartQuery(**

**CommandAPI *command,**

**const void (*gotRow)(void *),**

**const void (*queryComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(QueryAPI *, void *));**

**const unsigned int maxRows);**

**Parameters**

| | |
|---|---|
| command | A pointer to an instance of **CommandAPI** which is to be executed as a query. |
| gotRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| queryComplete | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |
| collectArgs | A pointer to a function that takes a pointer to an instance of **CommandAPI** and a pointer to a void as its parameters and returns nothing. |
| maxRows | An optional unsigned integer specifying the maximum number of rows that may be returned by the query. |

**Description**

This function attempts to execute a command, which may or may not have been prepared, as a query on the database associated with the command.
Before the query is dispatched to the database the callback function *collectArgs* is invoked to collect any parameters associated with the query. This is done by calling the **QueryAPI::AddArg** function. Note that the function *collectArgs* will also be called when a query is recovered after a fail-over.
For each row retrieved from the database by the query, subject to the optional limit *maxRows*, the callback function *gotRow* is invoked with *userdata* as its parameter. Before the callback function is called an error code is written to a private member variable in the instance pointed to by *client*.
An accessor function is provided to read the private error code, where zero indicates the row was got successfully, and a non-zero number indicates a specific failure.

When all rows have been retrieved by the query and the callback function for the last row has completed, the error code is written to the private member variable in the instance pointed to by *client* and the callback function *queryComplete* is invoked with the parameter *userdata*.
The error code written is zero to indicate that the query as completed successfully and some non-zero value to indicate a specific failure.
When the *queryComplete* callback function finishes the command is returned to its pre-execution state.
The *maxRows* optional parameter can be used to limit the number of rows returned by the query. The default value is POLY_MAX_ROWS_UNLIMITED (0xFFFFFFFF or 4294967295), i.e. effectively no limitation.

If the *sqlText* passed to the **StartQuery** contains SQL other than a query such as DDL or DML the results are undefined.

**Return Value**

Returns 0 if no error occurred, non-zero otherwise. If there was an error, its code can be obtained by calling **QueryAPI::GetError**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 9 | POLY_EQUERY | An SQL query has failed to execute. |
| 26 | POLY_ECOMMAND | The command has been prepared and its SQL is not suitable for executing as a query. |
| 27 | POLY_ECOMMANDINUSE | The command is being prepared, already executing as a query or added to a transaction. |

## CommandAPI::StartActiveQuery

**static int *StartActiveQuery(**

**CommandAPI *command,**

**const void (*insertedRow)(void *),**

**const void (*updatedRow)(void *),**

**const void (*deletedRow)(void *),**

**const void (*deltaComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(QueryAPI *, void *));**

**const unsigned int maxRows = POLY_MAX_ROWS_UNLIMITED,**

**const unsigned int secs = 0,**

**const unsigned int microsecs = 0);**

**C-callable Equivalent**

**int CommandAPI_StartActiveQuery(**

**CommandAPI *command,**

**const void (*insertedRow)(void *),**

**const void (*updatedRow)(void *),**

**const void (*deletedRow)(void *),**

**const void (*deltaComplete)(void *),**

**const void *userdata,**

**const void (*collectArgs)(QueryAPI *, void *));**

**const unsigned int maxRows = POLY_MAX_ROWS_UNLIMITED,**

**const unsigned int secs = 0,**

**const unsigned int microsecs = 0);**

**Parameters**

| | |
|---|---|
| command | A pointer to an instance of **CommandAPI** which is to be executed as an active query. |
| insertedRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| updatedRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| deletedRow | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| deltaComplete | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |
| collectArgs | A pointer to a function that takes a pointer to an instance of **CommndAPI** and a pointer to a void as its parameters and returns nothing. |
| maxRows | An optional unsigned integer specifying the maximum number of rows that may be returned by the query. |
| secs, *microsecs* | Optional unsigned integers specifying the minimum number of seconds and microseconds between deltas being sent by the server. The default values imply an ordinary active query. |

**Description**

This function attempts to execute a command, which may or may not have been prepared, as an active query on the database associated with the command.
Before the query is dispatched to the database the callback function *collectArgs* is invoked to collect any parameters associated with the query. This is done by calling the **QueryAPI::AddArg** function. Note that the function *collectArgs* will also be called when a query is recovered after a fail-over.
The first time the active query is issued the function invokes the callback function *insertedRow* with the parameter *userdata* to give the application the initial values for each row being monitored. When all rows have been presented using these callback functions, the callback function *deltaComplete* is invoked with *userdata* as its parameter.
Subsequently, the active query monitors the database for relevant changes. When a change is detected, the appropriate sets of callback functions are invoked. Before any callback function is invoked an error code is written to the private member variable in the instance of **CommandAPI**.
The callback function *insertedRow* is invoked with *userdata* when a row has been inserted.
The callback function *updatedRow* is invoked with *userdata* when a row has been changed. Only the changed columns are available to the application to read.
The callback function *deletedRow* is invoked with *userdata* when a row has been deleted.
When all the application has been notified of all changes, an error code is written to the private member variable in the **CommandAPI** instance and the *deltaComplete* callback function is invoked with *userdata* as its parameter.
The *maxRows* optional parameter can be used to limit the number of rows returned by the query. Using *maxRows* ensures the client will never be informed of more than *maxRows* rows. Furthermore, the client will always be informed of *maxRows* rows as long as sufficient rows are found to satisfy the query.
The optional *secs* and *microsecs* parameters are used to reduce the processing load on the server and the network traffic between server and client. Together they specify a minimum interval between deltas being sent by the server. See *The Real-Time Relational Database* for more details.
An accessor function is provided to read the private error code, where zero indicates the row was got successfully, and a non-zero number indicates a specific failure.

The error code written before any operation is zero indicate that the operation was completed successfully, and some non-zero value to indicate a specific failure.

**Return Value**

Returns 0 if no error occurred, non-zero otherwise. If there was an error, its code can be obtained by calling **QueryAPI::GetError**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 9 | POLY_EQUERY | An SQL query has failed to execute. |
| 12 | POLY_EDELTA | The delta for an active query has failed. |
| 26 | POLY_ECOMMAND | The command has been prepared and its SQL is not suitable for executing as a query. |
| 27 | POLY_ECOMMANDINUSE | The command is being prepared, already executing as a query or added to a transaction. |

## CommandAPI::DeleteCommand

**static int DeleteCommand(**

**CommandAPI *command,**

**const void (*deleteCallback)(void *),**

**const void *userdata);**

**C-callable Equivalent**

**int CommandAPI_DeleteCommand (**

**CommandAPI *command,**

**const void (*deleteCallback)(void *),**

**const void *userdata);**

**Parameters**

| | |
|---|---|
| command | A pointer to an instance of **CommandAPI** which is to be deleted. |
| deleteCallback | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing |
| userdata | A pointer to user-defined data with which to invoke the call-back function. The pointer is of type void *, which allows any type of data or object to be passed into the call-back function. |

**Description**

Function called to delete an instance of **CommandAPI** pointed to by *command*. If the command is executing as an active query, the query will also be aborted.
When the operation is complete, the function pointed to by *deleteCallback* is invoked on the *userdata* instance.
The **QueryAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

**Return Value**

Returns 0 if no error occurred, non-zero otherwise. If there was an error, its code can be obtained by calling **QueryAPI::GetError**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 27 | POLY_ECOMMANDINUSE | The command is added to a transaction. |

# TransAPI

This section specifies the public member functions of the **TransAPI** class.

## TransAPI::CreateTrans

**static TransAPI *CreateTrans(**

**const ClientAPI *client);**

**C-callable Equivalent**

**TransAPI *TransAPI_CreateTrans(**

**const ClientAPI *client);**

**Parameters**

client A pointer to an instance of **ClientAPI.**

**Description**

This function creates an instance of **TransAPI** on a database which has a client identified by *client*. A pointer to the created **TransAPI** instance is returned.

**Return Value**

A pointer to an instance of **TransAPI**.

## TransAPI::StartTrans

**static TransAPI *StartTrans(**

**const ClientAPI *client,**

**const char *sqlText,**

**const void (*transactionComplete)(void *),**

**const void *userdata,**

**int safeCommitMode = false);**

**C-callable Equivalent**

**TransAPI *TransAPI_StartSQLTrans(**

**const ClientAPI *client,**

**const char *sqlText,**

**const void (*transactionComplete)(void *),**

**const void *userdata,**

**int safeCommitMode);**

**Parameters**

| | |
|---|---|
| client | A pointer to an instance of **ClientAPI.** |
| sqlText | A pointer to a null terminated string containing the SQL statement. |
| transactionComplete | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |
| safeCommitMode | A flag indicating whether safe-commit mode should be used for the transaction. |

**Description**

This function creates an instance of **TransAPI**, and attempts to execute the SQL statement in the string *sqlText* on a database, which has a client, identified by *client*. A pointer to the created **TransAPI** instance is returned to the calling function.
When the operation is complete, the error code is written to a private member variable, and the function pointed to by *transactionComplete* is invoked using *userdata* as its parameter.
When the callback function finishes and returns control to the API, the instance of **TransAPI** is deleted.
If *safeCommitMode* is specified as true, the database only notifies the client of completion of the transaction when the changes made are safe. The exact meaning of 'safe' depends on the configuration of the system. If the system is running as fault tolerant pair of databases, then a transaction is considered to be safe then the standby database has acknowledged that it has also applied the change. If the system is configured to use Journaling, the change is considered to be safe when it has been successfully written to the journal file.
An accessor function is provided to read the private error code, where zero indicates that the query was aborted successfully, and some non-zero number indicates a specific failure.

**Return Value**

Returns a pointer to an instance of **TransAPI**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 10 | POLY_ESTARTTRANS | A transaction has failed to start. |
| 11 | POLY_ECOMMIT | A transaction has failed to commit. |

## TransAPI::StartTrans

**static int StartTrans(**

**const TransAPI *trans,**

**const void (*callback) (void *),**

**void *userdata);**

**C-callable Equivalent**

**int TransAPI_StartTrans(**

**const TransAPI *trans,**

**const void (*callback) (void *),**

**void *userdata);**

**Parameters**

| | |
|---|---|
| trans | A pointer to an instance of **TransAPI** which is to be started. |
| callback | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the call-back function. The pointer is of type void *, which allows any type of data or object to be passed into the call-back function. |

**Description**

This function starts a transaction, already created by calling **TransAPI::CreateTrans**.
When the operation is complete, the error code is written to a private member variable, and the function pointed to by *callback* is invoked on the *userdata* instance.
The **TransAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

**Return Value**

Returns 0 if no error occurred, non-zero otherwise. If there was an error, its code can be obtained by calling **TransAPI::GetError**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 21 | POLY_ETRANSINUSE | This function has already been called or a command has been added. |
| 22 | POLY_ENOQUERY | No queries have been added to the transaction using **TransAPI::AddQuery**. |

## TransAPI::AddCommand

**static int AddCommand(**

**TransAPI *trans,**

**CommandAPI *command,**

**const void (*collectArgs)(CommandAPI *, void *),**

**const void *userdata);**

**C-callable Equivalent**

**int TransAPI_AddCommand(**

**TransAPI *trans,**

**CommandAPI *command,**

**const void (*collectArgs)(CommandAPI *, void *),**

**const void *userdata);**

**Parameters**

| | |
|---|---|
| trans | A pointer to an instance of **TransAPI**. |
| command | A pointer to an instance of **CommandAPI** to be added to *trans*. |
| collectArgs | A pointer to a function that takes a pointer to an instance of **CommndAPI** and a pointer to a void as its parameters and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the callback function. The pointer is of type void *, which allows any type of data or object to be passed into the callback function. |

**Description**

This function adds the command instance, *command*, to the transaction *trans*. The command may or may not have been prepared. The command will be executed as part of the transaction when it is committed, using **TransAPI::Commit**. A command can be added to a same transaction multiple times, but it cannot be added to another transaction until the first has been committed. When a transaction is committed, all the commands that have been added to it are removed.
When **TransAPI::AddCommand** is called the callback function *collectArgs* is immediately invoked to collect any parameters associated with the command. This is done by calling the **QueryAPI::AddArg** function. Note the collection of arguments occurs when **TransAPI::AddCommand** is called rather than when the transaction is committed.
When the operation is complete, the error code is written to a private member variable.
The **TransAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

**Return Value**

Returns 0 if no error occurred, non-zero otherwise. If there was an error, its code can be obtained by calling **TransAPI::GetError**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|

| | | |
|---|---|---|
| 0 | POLY_OK | No error. |
| 20 | POLY_EWRONGCLIENT | The client of *trans* is not the same as *command*. |
| 21 | POLY_ETRANSINUSE | **TransAPI::Commit** has already been called. |
| 26 | POLY_ECOMMAND | The command has been prepared and its SQL is not suitable for use in a transaction. |
| 27 | POLY_ECOMMANDINUSE | The command is being prepared, executing as a query or added to a different transaction. |

## TransAPI::AddQuery

**static int AddQuery(**

**TransAPI *trans,**

**const QueryAPI *query);**

**C-callable Equivalent**

**int TransAPI_AddQuery(**

**TransAPI *trans,**

**const QueryAPI *query);**

**Parameters**

| | |
|---|---|
| trans | A pointer to an instance of **TransAPI**. |
| query | A pointer to an instance of **QueryAPI** to be added to *trans*. |

**Description**

This function adds the query instance, *query*, to the transaction *trans*.
When the operation is complete, the error code is written to a private member variable.
The **TransAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

**Return Value**

Returns 0 if no error occurred, non-zero otherwise. If there was an error, its code can be obtained by calling **TransAPI::GetError**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 19 | POLY_EQUERYINUSE | Query has already been added to a transaction. |
| 20 | POLY_EWRONGCLIENT | The client of *trans* is not the same as *query*. |
| 21 | POLY_ETRANSINUSE | **TransAPI::StartTrans** has already been called. |
| 27 | POLY_ECOMMANDINUSE | The query is a command that is not executing as a query. |

## TransAPI::Commit

**static int Commit(**

**const TransAPI *trans,**

**const void (*updateCallback) (QueryAPI *, void *),**

**const void (*commitCallback) (void *),**

**void *userdata,**

**bool safeCommitMode = false);**

**C-callable Equivalent**

**int TransAPI_Commit(**

**const TransAPI *trans,**

**const void (*updateCallback) (QueryAPI *, void *),**

**const void (*commitCallback) (void *),**

**void *userdata,**

**int safeCommitMode);**

**Parameters**

| | |
|---|---|
| trans | A pointer to an instance of **TransAPI** which is to be committed. |
| updateCallback | A pointer to a function that takes a pointers to a **QueryAPI** and a void as its parameters and returns nothing. |
| commitCallback | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the call-back function. The pointer is of type void *, which allows any type of data or object to be passed into the call-back function. |
| safeCommitMode | A flag indicating whether safe-commit mode should be used for the transaction. |

**Description**

This function is used to initiate the commit process on a transaction, passed as *trans*. The function takes two callback functions, the first called when the commit has been started, and is waiting for data modification on the queries added to the transaction. This is called once for each query included in the transaction, passing the pointer to the query as the first parameter to the callback. When the commit is initiated, the error code is written to a private member variable. This calls the second of the two call back functions which is pointed to by *updateCallback* and is invoked on the *userdata* instance.
When the commit operation is complete, the error code is written to a private member variable, and the function pointed to by *commitCallback* is invoked on the *userdata* instance.
If *safeCommitMode* is specified as true, the database only notifies the client of completion of the transaction when the changes made are safe. The exact meaning of ‘safe’ depends on the configuration of the system. If the system is running as fault tolerant pair of databases, then a transaction is considered to be safe then the standby database has acknowledged that it has also applied the change. If the system is configured to use journaling, the change is considered to be safe when it has been successfully written to the journal file.

The **TransAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

**Return Value**

Returns 0 if no error occurred, non-zero otherwise. If there was an error, its code can be obtained by calling **TransAPI::GetError**.

**Error Codes**

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 21 | POLY_ETRANSINUSE | **TransAPI::StartTrans** has already been called. |
| 22 | POLY_ENOQUERY | No queries have been added to the transaction using **TransAPI::AddQuery**. |

## TransAPI::DeleteTrans

**static void DeleteTrans(**

**const TransAPI *trans,**

**const void (*deleteCallback) (void *),**

**void *userdata);**

**C-callable Equivalent**

**void TransAPI_DeleteTrans(**

**const TransAPI *trans,**

**const void (*deleteCallback) (void *),**

**void *userdata);**

**Parameters**

| | |
|---|---|
| trans | A pointer to an instance of **TransAPI** which is to be deleted. |
| deleteCallback | A pointer to a function that takes a pointer to a void as its only parameter and returns nothing. |
| userdata | A pointer to user-defined data with which to invoke the call-back function. The pointer is of type void *, which allows any type of data or object to be passed into the call-back function. |

**Description**

Function called to delete the transaction pointed to by *trans*.
When the operation is complete, the function pointed to by *deleteCallback* is invoked on the *userdata* instance.
The **TransAPI::GetError** function is provided to read the private error code, where zero indicates the operation was a success and some non-zero number indicates a specific failure.

## TransAPI::GetError

**static int GetError(**

**const TransAPI *trans,**

**char *buffer = 0,**

**int length = 0,**

**int *pNativeError = 0);**

**C-callable Equivalent**

**int TransAPI_GetError(**

**const TransAPI *trans,**

**char *buffer,**

**int length,**

**int *pNativeError);**

**Parameters**

| | |
|---|---|
| trans | A pointer to an instance of **TransAPI**. |
| Buffer | A pointer to a buffer into which the error message will be copied. |
| Length | An integer specifying the length of the buffer. |
| pNativeError | A pointer to an integer variable into which the error code will be copied. |

**Description**

This function returns the error code associated with the last operation performed on the transaction. It also allows the underlying error code and message generated by the database optionally to be obtained. If the operation was successful the value returned will be zero and the buffer, if any, supplied will not be modified.

Callback API error codes are listed in section 4.

If non-null, the *buffer* argument should point to a memory buffer with size in bytes specified by the *length* argument. The underlying database error message will then be copied into the buffer by the function.

If non-null the *pNativeError* argument must be the address of an *integer* variable into which the underlying database error code will be stored by the function.

Database error codes are listed in the Real-Time Relation Database user manual.

**Return Value**

The CallBack API error code associated with the last operation performed on the transaction.

## TransAPI::GetRowCount

**static int GetRowCount(**

**TransAPI *trans);**

**C-callable Equivalent**

**int TransAPI_GetRowCount(**

**TransAPI *trans);**

**Parameters**

trans A pointer to an instance of **TransAPI**.

**Description**

This function returns the number of records in the database that were modified by the last transaction performed using the transaction object. The count includes all records inserted, updated and deleted by the transaction, including those modified indirectly by sub-systems such as CL.

**Return Value**

The number of records modified in the last transaction performed using the transaction object.

## TransAPI::GetSQLRowCount

**static int GetSQLRowCount(**

**TransAPI *trans);**

**C-callable Equivalent**

**int TransAPI_GetSQLRowCount(**

**TransAPI *trans);**

### Parameters

| | |
|---|---|
| trans | A pointer to an instance of **TransAPI**. |

### Description

This function returns the number of records in the database that were directly modified by SQL statements executed in the last transaction performed using the transaction object. The count includes all records directly inserted, updated and deleted by SQL statements executed in the transaction. It does not include any records modified indirectly by sub-systems such as CL.

### Return Value

The number of records directly modified by SQL statements executed in the last transaction performed using the transaction object.

# 4. Error Codes

The following table lists all the error codes that can be returned by **GetError**.

| Value | Mnemonic | Description |
|---|---|---|
| 0 | POLY_OK | No error. |
| 1 | POLY_EALREADYINIT | The scheduler has already been initialised. |
| 2 | POLY_ENOINIT | The scheduler has not been initialised. |
| 3 | POLY_ENOSTART | The scheduler has not been started-up. |
| 4 | POLY_ECONNECT | Failed to connect to database. |
| 5 | POLY_ECOLUMNNAME | Invalid column name. |
| 6 | POLY_ECOLUMNNO | Invalid column number. |
| 7 | POLY_ENOVALUE | A column value is unavailable. |
| 8 | POLY_ENULLVALUE | A column value is null. |
| 9 | POLY_EQUERY | An SQL query has failed to execute. |
| 10 | POLY_ESTARTTRANS | A transaction has failed to start. |
| 11 | POLY_ECOMMIT | A transaction has failed to commit. |
| 12 | POLY_EDELTA | The delta for an active query has failed. |
| 13 | POLY_ETRUNCATED | The data requested for a column has been truncated. |
| 14 | POLY_EINUSE | A client connection is in use and cannot be deleted. |
| 16 | POLY_ENOINFO | Information requested for a query before it has returned. |
| 17 | POLY_ESHUTDOWN | Failed to shutdown a server. |
| 18 | POLY_ELOGIN | Attempt to log onto server failed. |
| 19 | POLY_EQUERYINUSE | The query has already been added to a transaction. |
| 20 | POLY_EWRONGCLIENT | Attempt to add a query or command to a transaction on a different client |
| 21 | POLY_ETRANSINUSE | **TransAPI::StartTrans** or **TransAPI::Commit** has already been called |
| 22 | POLY_ENOQUERY | No queries have been added to the transaction using **TransAPI::AddQuery** |
| 23 | POLY_ENOTINCOMMIT | **TransAPI::Commit** has not been called |
| 24 | POLY_ENOCHANGE | A column value has not changed. |
| 25 | POLY_ETIMEDOUT | **AppAPI::ExecuteWithTimeout** has timed out. |
| 26 | POLY_ECOMMAND | The command has been prepared and its SQL is not a suitable for the requested operation. |
| 27 | POLY_ECOMMANDINUSE | The command is in use. The precise reason depends on the |

| | | |
|---|---|---|
| | | operation requested, but it can be because the command is being prepared, already prepared, executing as a query or added to a transaction. |

# 5. Data Types

The following table lists all data types:

| Value | Mnemonic | Database Type |
|---|---|---|
| 1 | POLY_TYPE _INTEGER | Integer, Integer8, Integer16 |
| 2 | POLY_TYPE_STRING | Varchar |
| 3 | POLY_TYPE_FLOAT | Float, Float32 |
| 6 | POLY_TYPE_BOOLEAN | Bool |
| 7 | POLY_TYPE_BINARY | Binary |
| 8 | POLY_TYPE_DATETIME | DateTime |
| 29 | POLY_TYPE_INTEGER64 | Integer64 |

# 6. Column Flags

The following table lists all column flags:

| Value | Mnemonic | Description |
|---|---|---|
| 1 | POLY_FLAG_PRIMARY_KEY | The column is a primary key column. |
| 2 | POLY_FLAG_NULLABLE | The column may contain null values. |
| 4 | POLY_FLAG_UPDATABLE | The column may be updated. |

# 7. Client Options

The following table lists all client option values:

| Value | Mnemonic | Description |
|---|---|---|
| 1 | POLY_FT_OPTION_RECONNECTION_RETRIES | Number of times reconnection is attempted to each data service when an established connection has been lost. |
| 2 | POLY_FT_OPTION_RECONNECTION_INTERVAL | Interval, in milliseconds, between making each reconnection attempt. |
| 3 | POLY_FT_OPTION_RECONNECTION_TIMEOUT | Time-out period, in milliseconds, associated with each reconnection attempt. |
| 4 | POLY_FT_OPTION_HEALTHCHECK_INTERVAL | Interval, in milliseconds, between each heartbeat message being sent to the server. |
| 5 | POLY_FT_OPTION_HEALTHCHECK_TIMEOUT | Time-out, in milliseconds, of each heartbeat sent to the server. If a reply is not received from the server within this time-out period, the server is considered to be in error and the connection is closed and fail-over is initiated. |
| 6 | POLY_FT_OPTION_ENABLE | Enable FT options. |
| 7 | POLY_FT_OPTION_KEEP_COPY | Maintain a copy of all data returned by an active query to allow calculation of data changes after a fail-over |
| 8 | POLY_FT_OPTION_MAP_ROWIDS | Maintain a mapping of row ids across fail-over to facilitate the calculation of data changes. |

# Index