



Product:	CDP2SQL
Product version:	v1.0
Document ID:	PM-CDP2SQL
Doc revision:	A
Written/Appr.:	KD / NPE
Date:	12. Feb. 2009

Industrial Control Design AS



CDP2SQL v1.0

Programmer Manual

The content of this document is confidential information not to be published without the consent of Industrial Control Design AS.

Industrial Control Design AS, www.icd.no, support@icd.no, forum.icd.no

Contents

1. INTRODUCTION.....	3	5. CDP2SQL HIGH-LEVEL API.....	14
1.1. About.....	3	5.1. Class DatabaseHelper.....	14
1.2. Terms and Definitions.....	3	5.1.1. Description.....	14
2. INSTALLATION.....	4	5.1.2. Methods.....	14
2.1. Prerequisites:.....	4	5.1.3. Example.....	14
2.2. Linking with the CDP2SQL library.....	4	5.2. Class CDPDatabase.....	14
2.3. How to add the CDP2SQL library in a new or existing project.....	5	5.2.1. Description.....	14
2.4. Modify the project XML files.....	8	5.2.2. Methods.....	14
3. FUNCTIONAL OVERVIEW.....	9	5.2.3. Example.....	15
3.1. SQLLogger Component.....	9	5.3. Class StatementPrepare.....	15
3.2. CDP2SQL High-Level API.....	9	5.3.1. Description.....	15
3.3. Generic Database Interface (DBI).....	9	5.3.2. Methods.....	15
3.4. SQL Database.....	9	5.3.3. Example.....	15
4. SQLLOGGER COMPONENT.....	10	6. GENERIC DATABASE INTERFACE (DBI).....	16
4.1. About.....	10	6.1. Overview.....	16
4.2. Overview.....	10	6.1.1. Description.....	16
4.2.1. Component Architecture.....	10	6.1.2. Class overview.....	16
4.2.2. Component States.....	10	6.2. Class Database.....	16
4.3. Sending event messages to the SQLLogger.....	11	6.2.1. Description.....	16
4.3.1. About.....	11	6.2.2. Methods.....	16
4.3.2. Event data structures.....	11	6.2.3. Example.....	17
4.3.3. How to send event data.....	11	6.3. Class Query.....	17
4.4. Sending query messages to the SQLLogger.....	12	6.3.1. Description.....	17
4.4.1. About.....	12	6.3.2. Methods.....	17
4.4.2. Query data structures.....	12	6.3.3. Example.....	17
4.4.3. How to send query data.....	12	6.4. Class Statement.....	18
4.5. Sending SQL command messages to the SQLLogger.....	13	6.4.1. Description.....	18
4.5.1. About.....	13	6.4.2. Methods.....	18
4.5.2. SQL command data structures.....	13	6.4.3. Example.....	18
4.5.3. How to send SQL command data.....	13	6.5. Class Transaction.....	18
		6.5.1. Description.....	18
		6.5.2. Methods.....	18
		6.5.3. Example.....	18
		7. SQL DATABASE ENGINE.....	19
		7.1. Concurrency and multi-threading.....	19
		7.2. Database location.....	19
		8. APPENDIX.....	20
		8.1. Example SQL.....	20
		8.1.1. Creating tables.....	20
		8.1.2. Inserting data rows.....	20
		8.1.3. Requesting data rows.....	20
		8.2. References.....	20

1. Introduction

1.1. About

This document describes how the CDP2SQL CDP component works, and how to set it up and use it with the CDP system. The CDP2SQL CDP component has the following features:

- Facilitates logging of signals and event messages to SQL databases.
- Provides high-level abstraction API that requires no SQL knowledge.
- Provides generic database interface to SQL database.
- Allows changing the underlying database type with minimal changes to existing code.

1.2. Terms and Definitions

CDP

Control Design Platform.

CDP Controller

Computer (Usually an industrial computer) running CDP application, usually on a true real-time operating system.

CDPUI

The CDP graphical user interface.

Component

Object with strict interface specification.

DBI

Database Interface

ODBC

Open Database Connectivity

SQL

Structured Query Language

2. Installation

The CDP2SQL can be delivered both as source code and as a separate library which is linked into the application. If delivered as separate library, the CDP2SQLLib Setup will by default install all files in sub-folders of “CDP Developer”.

2.1. Prerequisites:

- A valid CDP license
- Familiar with CDP
- Familiar with SQL databases
- CDP version 2.3.1.6

2.2. Linking with the CDP2SQL library

Copy the example SQLLogger.xml component file that came with the distribution into your project's Components folder, and the SQLLogger.xml model XML file into the Models folder.

Copy the CDP2SQLLib folder either into your project, to a location for standard libraries, or copy the CDP2SQLLib_<config>.lib and header-files to a location where your linker can find it.

Include the CDP2SQLLib.h header file in your application's Libraries.h file:

```
#include <CDP2SQLLib.h>
```

If necessary, update the include paths by adding the path to where to find CDP2SQLLib header-files (and source code if available):

- CDP_Application -> Properties
 - C/C++ -> General -> Additional Include Directories

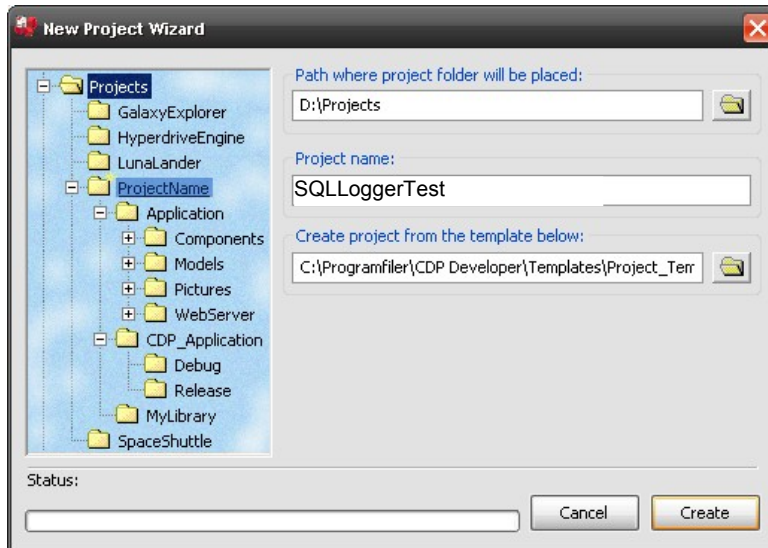
Link the CDP2SQL library into your application by putting CDP2SQL_<config>.lib in a folder included in your linker path. For the CDP_Application project:

- CDP_Application->Properties
 - Linker -> Input -> Additional dependencies

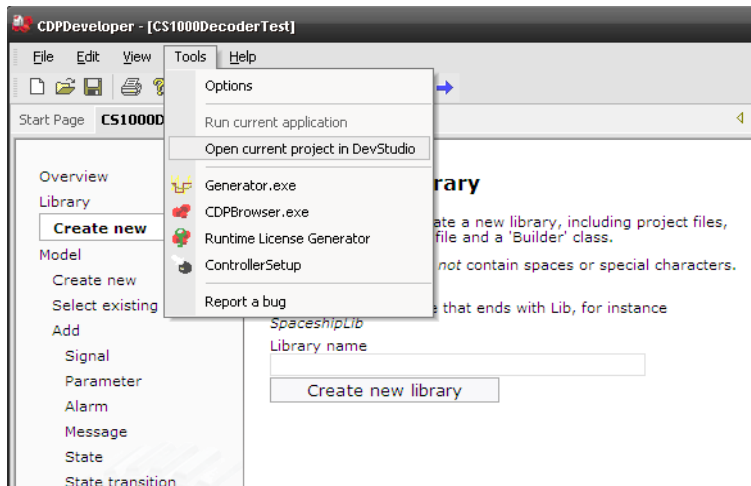
Finally, link to the SQL database implementation used, SQLite3. Same procedure as for linking with the CDP2SQL library above.

2.3. How to add the CDP2SQL library in a new or existing project

- Start CDPDeveloper located at Start Menu > Programs > CDP > CDPDeveloper
- Make a new project by Selecting File->New, type in project name and click Create (or skip four steps forward and open an already existing project in Visual Studio)

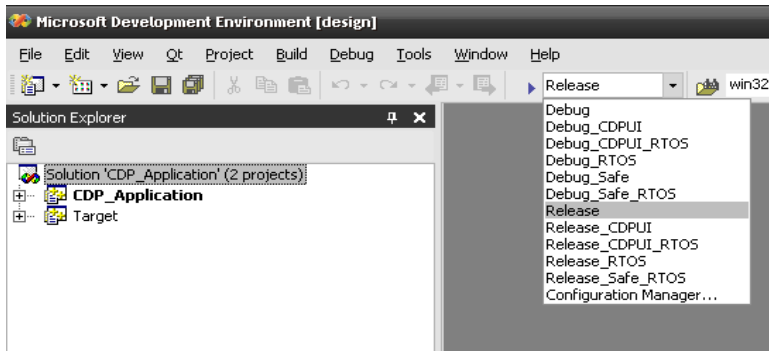


- Choose 'Open current project in DevStudio' from the Tools menu:

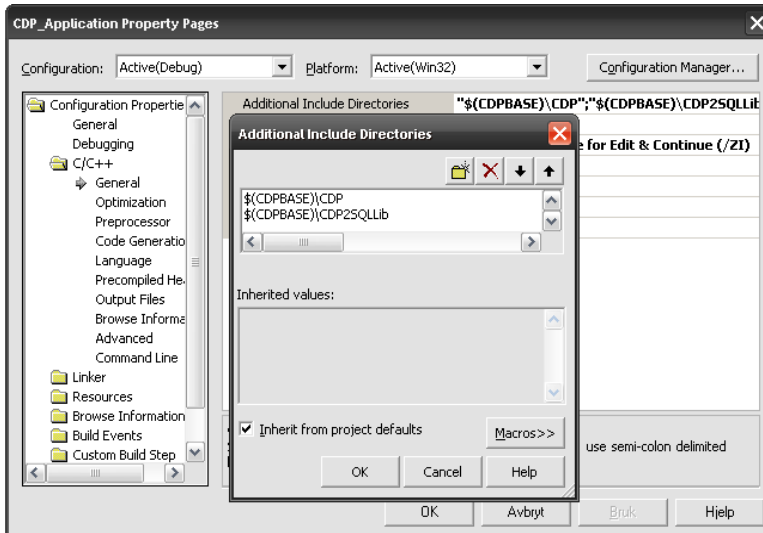


- If Visual Studio asks to convert the project, accept this by selecting 'Next'/'Finish'/'Close' until done. Close the conversion report.

- In Visual Studio, select the configuration that is right for your target platform.

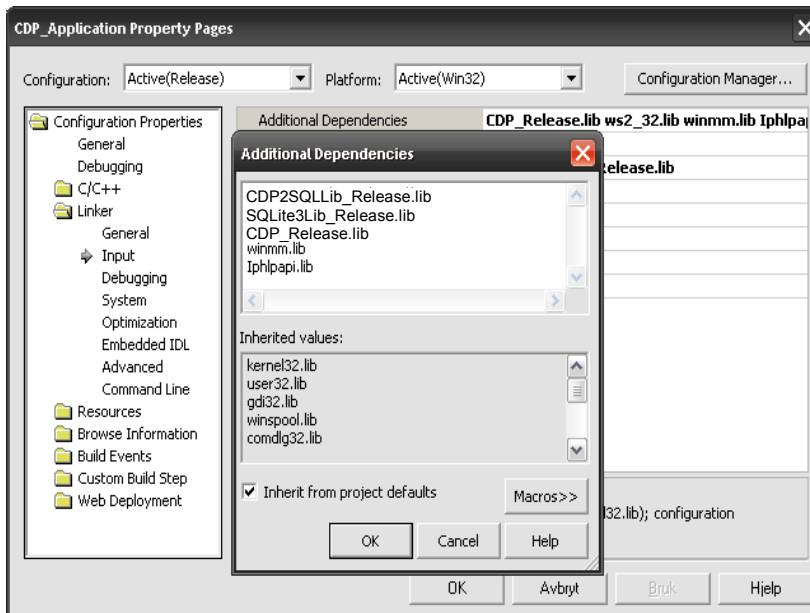


- Select 'CDP_Application' from the 'Solution Explorer', right-click and select Properties.
- In C++/Additional include Directories, make sure it says: “\$(CDBASE)\CDP2SQLLib”; “\$(CDBASE)\CDP”. When compiling for On Time RTOS, the following include is also required: “;”\$(RTTarget)\Include”.

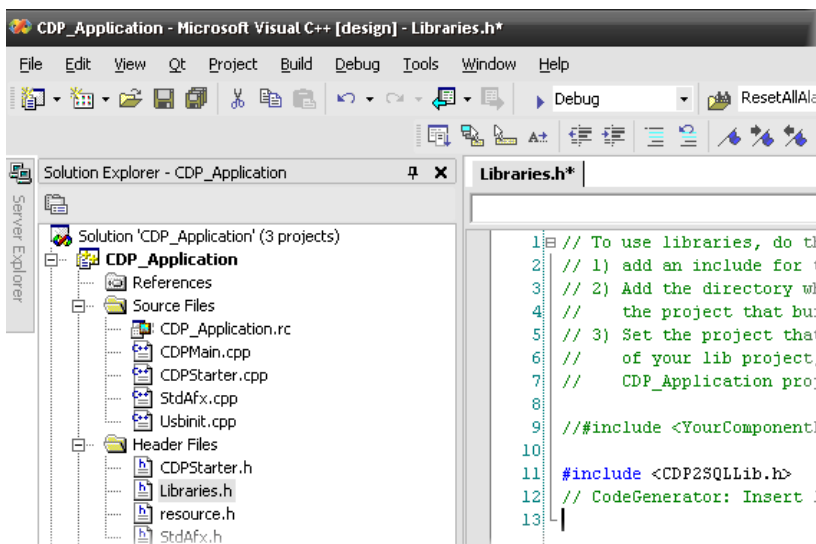


The compiler will look in the directories specified in 'Additional Include Directories' for files that you #include in your .cpp and .h files. If you get an error 'Can not open include file ...' then it is most likely caused by a missing include directory, or that the file you #include does not exist.

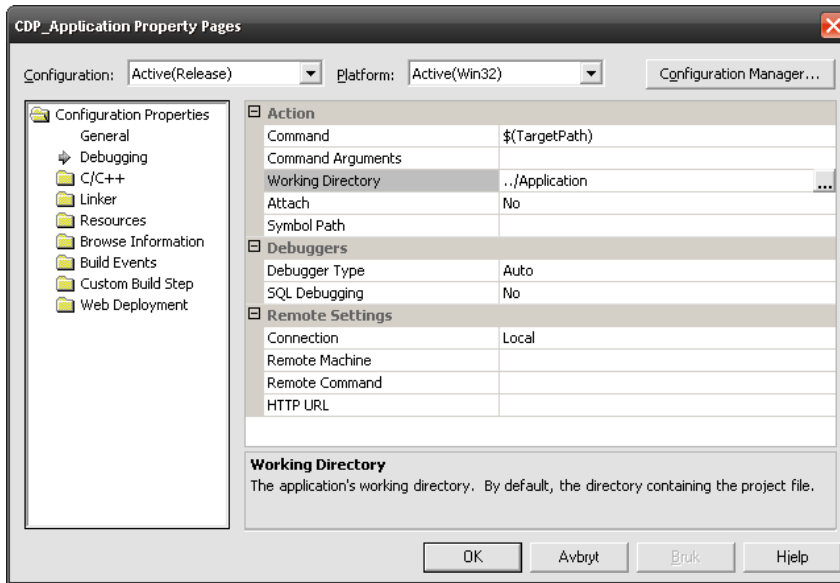
- In Linker->Input, make sure you list CDP2SQLLib_Release.lib, SQLite3Lib_Release.lib and CDP_Release.lib in addition to all the win32 libraries you need. This will ensure that the linker finds all the functions that is referenced in the code. Note that when compiling for other Operating Systems you will need to replace these libraries with the ones for the target OS.



- In the 'Solution Explorer', inside the 'CDP_Application' project, locate the Header file Libraries.h, and add the line `#include <CDP2SQLLib.h>`:



- Finally, make sure that the working directory of the project is set to ../Application. This will ensure that the executable is started where the configuration files are located.



2.4. Modify the project XML files

Add the following to your project's Application.xml:

Inside the <Components> element, add an instance of a SQLLogger component, for instance:

```
<Component Name="SQLLogger" src="Components/SQLLogger.xml"></Component>
```

Or, inside the <Subcomponents> element, add:

```
<Subcomponent Name="SQLLogger" Model="SQLLogger" src="Components/SQLLogger.xml" />
```

This will tell CDP to initialize a component named "SQLLogger" from a component file located at "Components/SQLLogger.xml". Make sure that your Models\ folder contains an SQLLogger.xml model file, or the component will not be initialized correctly.

3. Functional Overview

3.1. SQLLogger Component

The SQLLogger CDP component is a ready-to-use component for logging groups of signals to SQL database at a given frequency. The component is configured in XML. The SQLLogger component can also be sub-classed for extending the functionality.

3.2. CDP2SQL High-Level API

The high-level abstraction API provides a collection of application-specific APIs for logging to and manipulating of SQL databases without having to deal with the SQL language. It includes helper methods and classes for creating tables, inserting rows etc.

3.3. Generic Database Interface (DBI)

The CDP2SQL generic database interface (DBI) aims to insulate the programmer from as much of the underlying SQL implementation as possible. It is comprised of a small collection of holder classes that the developer uses to access and manipulate the SQL database. Using this API requires knowledge of the SQL language.

3.4. SQL Database

At the bottom of the CDP2SQL application and library, lies one or more SQL database implementations. The reference implementation is based on the SQLite3 database, which is a simple, file-based SQL engine.

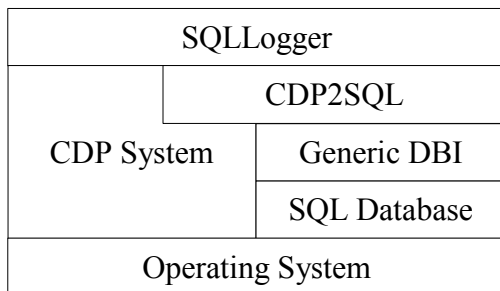
4. SQLLogger Component

4.1. About

The SQLLogger CDP component is a ready-to-use component for logging groups of signals to SQL database at a given frequency. The component is configured in XML as described in the CDP2SQL User Manual.

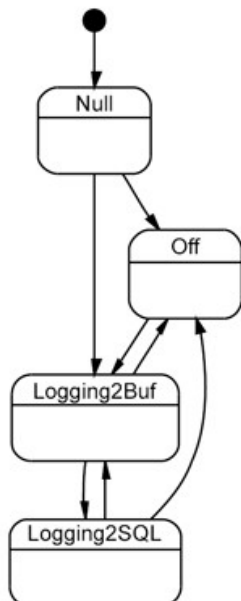
4.2. Overview

4.2.1. Component Architecture



4.2.2. Component States

The component includes the following states. See the CDP2SQL User Manual for more information.



4.3. Sending event messages to the SQLLogger

4.3.1. About

The SQLLogger does not include CDPEventManager functionality as this would require dependencies to the CDPEventManager library. Instead, the SQLLogger can receive messages with event data. The following sections explain what data can be received and how to send it.

4.3.2. Event data structures

The following data can be received by the SQLLogger:

```

/**
 * Struct for data samples of type EventData
 */
struct EventData
{
    double logTimestamp;    // The time when the event was added to buffer
    int eventId;           // EventID - Unique identifier (for this alarmNode)
    int originalEventId;   // Set for reprise events
    int handle;            // CDP Handle of object that generated the event
    int status;            // New status of object after the event
    int eventCode;         // Code describing the alarm event
    char timestamp[22];    // Time stamp when the event was reported (local time in reporting node)
    char eventName[32];    // A name identifying the event type e.g. GroupChanged, TextChanged, StateChanged
    char eventDescription[400]; // Human readable description of the event
    char objectName[256];  // The full name of the event object
};

/**
 * Message struct for sending EventData
 */
struct MessageEventData
{
    Message header;
    EventData data;
};

#define CM_EVENTDATA 0x0000088
  
```

4.3.3. How to send event data

Sending event data to the SQLLogger is simple. First, create a CDPCConnector and connect it to the SQLLogger. Then, update a MessageEventData structure with the event data to send and send it using the CDPCConnector. In the following example, the CDPCConnector is named logConnectorOne.

```

#include <CDP2SQL/CDPDatabaseDefines.h>
using namespace CDP2SQL;

/**
 * Function for sending event data
 */
void LogManager::SendEventMessage()
{
    MessageEventData msg;
    msg.data.logTimestamp = htod(0);
    msg.data.eventId = htonl(1);
    msg.data.originalEventId = htonl(2);
    msg.data.handle = htonl(3);
    msg.data.status = htonl(4);
    msg.data.eventCode = htonl(5);
    strcpy(msg.data.timestamp, CDPTIME::GetCurrentDateTimeMsString(true).c_str());
    strcpy(msg.data.eventName, "TestEvent");
    strcpy(msg.data.eventDescription, description);
    strcpy(msg.data.objectName, "");

    MESSAGE_FILL(&msg, 0, Handle(), CM_EVENTDATA);
    msg.header.parmaSize = htonl(sizeof(msg.data));

    else if (logConnectorOne.Connected())
        logConnectorOne.SendMessage((Message*) &msg, true);
    else
        CDPMESSAGE("SendEventMessage: Unable to send event.\n");
}
  
```

4.4. Sending query messages to the SQLLogger

4.4.1. About

The following sections explain how to send a query to a logger component.

4.4.2. Query data structures

```

/**
 Struct for data samples of type EventData
 */
struct QueryData
{
  enum{ OUTFILE_MAX = 128, DELIMITER_MAX = 4 };

  char outfile[OUTFILE_MAX];
  char delimiter[DELIMITER_MAX];
  char query[MESSENGER_UDPMESSEGEDATABUFFERSIZE-OUTFILE_MAX-DELIMITER_MAX-sizeof(MessageTransportPacket)];
};

typedef std::vector<QueryData> QueryDataList;

/**
 Message struct for sending QueryData
 */
struct MessageQueryData
{
  Message header;
  QueryData data;
};

#define CM_QUERYDATA 0x0000089
  
```

4.4.3. How to send query data

Sending query data to the SQLLogger is simple. First, create a CDPConnector and connect it to the SQLLogger. Then, update a MessageQueryData structure with the query data to send and send it using the CDPConnector. In the following example, the CDPConnector is named logConnectorOne.

```

#include <CDP2SQL/CDPDatabaseDefines.h>
using namespace CDP2SQL;

/**
 Function for sending query data
 */
void LogManager::SendQueryMessage()
{
  MessageQueryData msg;
  strcpy(msg.data.outfile, "query2.txt");
  strcpy(msg.data.delimiter, ",");
  strcpy(msg.data.query, "select * from LoggerOne");

  MESSAGE_FILL(&msg, 0, Handle(), CM_QUERYDATA);
  msg.header.parmaSize = htonl(sizeof(msg.data));

  if(logConnectorOne.Connected())
    logConnectorOne.SendMessage((Message*)&msg, true);
  else
    CDPMessage("SendQueryMessage: Unable to send query message.\n");

  return 1;
}
  
```

4.5. Sending SQL command messages to the SQLLogger

4.5.1. About

The following sections explain how to send a SQL command message to a logger component.

4.5.2. SQL command data structures

```
/**
 * Message struct for sending SQL command
 */
struct MessageSQLData
{
    Message header;
    char data[MESSENGER_UDPMESSAGEATABUFFERSIZE-sizeof(MessageTransportPacket)];
};

#define CM_SQLCMD 0x0000090
```

4.5.3. How to send SQL command data

Sending a SQL command to the SQLLogger is simple. First, create a CDPConnector and connect it to the SQLLogger. Then, update a MessageSQLData structure with the command data to send and send it using the CDPConnector. In the following example, the CDPConnector is named logConnectorOne. The function will send a create table command the first time it is run and insert data commands in following runs.

```
#include <CDP2SQL/CDPDatabaseDefines.h>
using namespace CDP2SQL;

/**
 * Function for sending SQL commands
 */
void LogManager::SendSQLCommandMessage ()
{
    MessageSQLData msg;
    static int count = 0;

    if(count++ == 0)
        strcpy(msg.data, "CREATE TABLE logManagerTable(id INTEGER PRIMARY KEY, data VARCHAR(100))");
    else
        strcpy(msg.data, "INSERT INTO logManagerTable values(null, 'Test')");

    MESSAGE_FILL(&msg, 0, Handle(), CM_SQLCMD);
    msg.header.parmaSize = htonl(strlen(msg.data)+1);

    if(logConnectorOne.Connected())
        logConnectorOne.SendMessage((Message*)&msg, true);
    else
        CDPMessage("SendSQLCommandMessage: Unable to send query message.\n");

    return 1;
}
```

5. CDP2SQL High-Level API

The high-level abstraction API provides a collection of application-specific APIs for logging to and manipulating of SQL databases without having to deal with the SQL language. It includes helper methods and classes for creating tables, inserting rows etc.

5.1. Class DatabaseHelper

5.1.1. Description

The DatabaseHelper class is a collection of database utility methods without any dependencies to CDP itself.

5.1.2. Methods

- Open – open a database file or connection.
- Close – close the database connection.
- CreateTable – create a table with fields in the database.
- DeleteTable – delete a table from the database.
- CreateStatement – create SQL statement for logging CDP signals.
- InsertRow – insert a row of values into a table.
- IdenticalTable – checks if the fields of two tables are identical.
- GetNumberOfRows – check how many entries are in a table.
- ConvertType – converts CDP signal types to valid SQL types.
- GetDatabase – access the lower-level database object for more direct SQL manipulation.
- QueryToFile – execute a query and output the result in a file.

5.1.3. Example

Example on how to create a table with three fields.

```
using namespace CDP2SQL;
CDP2SQL::DatabaseHelper db("mysqlite3databasefile.db", "sqlite3");
CDP2SQL::ColumnInfo colElements;
colElements.Add("id", "INTEGER PRIMARY KEY");
colElements.Add("Name", "varchar(100)");
colElements.Add("Value", "varchar(300)");
db.CreateTable("mykeymap", colElements);
db.Close();
```

5.2. Class CDPDatabase

5.2.1. Description

The CDPDatabase is used by SQLLogger to manipulate the database. It inherits from the more generic DatabaseHelper class.

5.2.2. Methods

- Open – open a database file or connection.
- Close – close the database connection.
- CreateSignalTable – create a table with fields for logging signals in the database.
- CreateEventTable – create a table with fields for logging events in the database.

- LogSignals – log CDP signals with values and timestamp.
- LogEvents – log CDP events.
- IdenticalSignalTable – checks if the fields of two signal tables are identical.
- QueryToFile - execute a query and write query output to file with a CDP header.
- QueryToFile - get data from query buffer, execute query and return the result to file.
- ValidateQueryData - validate query data
- ExecuteSQL - get SQL Commands from SQL buffer and execute

5.2.3. Example

Example on how to create a table with three fields.

```
using namespace CDP2SQL;
CDP2SQL::CDPDatabase db("mysqlite3databasefile.db", "sqlite3");
CDP2SQL::ColumnInfo colElements;
colElements.Add("id", "INTEGER PRIMARY KEY");
colElements.Add("Name", "varchar(100)");
colElements.Add("Value", "varchar(300)");
db.CreateTable("mykeymap", colElements);
db.Close();
```

5.3. Class StatementPrepare

5.3.1. Description

An extension of the Statement class that can create SQL statements dynamically.

5.3.2. Methods

- CreateStatement – create statement for inserting rows into a given table, optionally specifying a list of which columns to insert values into.
- CreateUpdateStatement – create statement for updating specific existing rows in a given table.

5.3.3. Example

Example on how to add rows to table using Statement prepare:

```
using namespace CDP2SQL;
CDPDatabase db("employment", "sqlite3");
StatementPrepare st(db, "employee");
st.Bind(1, 123); st.Bind(2, "John Doe"); st.Execute(); st.Reset();
```

6. Generic Database Interface (DBI)

6.1. Overview

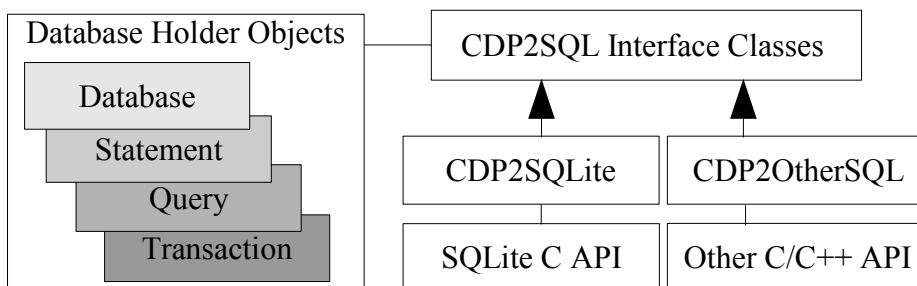
6.1.1. Description

The CDP2SQL generic database interface (DBI) aims to insulate the programmer from most of the underlying SQL implementation as possible. As such, it is comprised of a small collection of holder classes that the developer uses to access and manipulate the SQL database. Using this API requires knowledge of the SQL language.

6.1.2. Class overview

The end-programmer will mostly work with the following classes:

- Database - connecting to a database.
- Statement - used for inserting into and updating the database.
- Query - used for reading data from the database.
- Transaction - simple transaction support handling.
- SQLException - thrown for most errors.



6.2. Class Database

6.2.1. Description

The main database class used to connect a database and make simple queries. (For more complicated queries, you'll want some way of iterating over the results. See the Query class.)

6.2.2. Methods

- ConnectTo - connects to a database.
- Disconnect - disconnects from the database.
- Execute - run a database command.
- ExecuteScalar - run a simple query with one integer result.

6.2.3. Example

Short example usage of the API to insert a row into a table and then count the rows in that table. Note that the database is disconnected automatically when the Database object goes out of scope. Any failure will raise an SQLException.

```
using namespace CDP2SQL;
try {
    Database db("sqlite3", "mysqlite3databasefile.db");
    db.Execute("INSERT INTO employee VALUES (99, 'John Doe');" );
    db.ExecuteScalar("SELECT count(*) FROM employee;");
} catch (SQLException& e) {
    printf("Error occurred: %s\n", e.Message());
}
```

6.3. Class Query

6.3.1. Description

Class that is used for making queries to the database. It allows you to iterate over the results, using a "cursor".

6.3.2. Methods

- Constructor - create a Query connected to a database.
- ExecuteQuery - run an SQL query or Statement against the database.
- NumFields - how many fields (columns) are in the result.
- FieldName - get table field name of a given column.
- FieldType - get table field type of a given column.
- FieldValueInt - get data on current row for a given column as integer.
- FieldValueStr - get data on current row for a given column as string.
- FieldValueDouble - get data on current row for a given column as floating point.
- FieldValueBool - get data on current row for a given column as boolean.
- FieldValueTime - get data on current row for a given column as time.
- NextRow - move cursor to the next row in the result.
- IsEof - have we reach the end of the query results?

6.3.3. Example

Simple example showing how to make a query and iterate through the result.

```
Database db("sqlite3", "mytestingdatabase.db");
Query q(db, "select * from employee;"); // Calls ExecuteQuery immediately.

// Get meta-information about the results. Field names and types.
for (int index = 0; index < q.NumFields(); index++)
    printf("Index: %d Name: %s Type: %s \n", index, q.FieldName(index), q.FieldType(index));

// Iterate over all the rows in the query result
while (!q.IsEof())
{
    int workerid = q.FieldValueInt(0);
    const char *fullname = q.FieldValueStr(1);
    q.NextRow();
}
```

6.4. Class Statement

6.4.1. Description

While you can modify a database by creating custom SQL strings and calling Database::Execute directly, it is not a recommended approach due to potential for memory-overhead, escaping-mishaps and SQL injection attacks. Instead you can use a Statement object to create a precompiled statement which will handling data escaping/quoting and can be reused if you are inserting multiple rows.

6.4.2. Methods

- Compile – compile a Statement with an SQL insertion/update string.
- Bind - replace a placeholder in the insertion string with a data value (can be integer, string etc);
- Execute - execute the statement. All placeholders must be binded first.
- Reset - resets the bindings to placeholders. Statement can now be reused.

6.4.3. Example

Example using a prepared statement with placeholders ten employees with the same name.

```
Database db("sqlite3", "mytestingdatabase.db");
Statement st(db, "INSERT INTO employee VALUES (?, ?);");
for (int i=0; i<10; i++) // Insert ten unknown employees...
{
    st.Bind(1, i); // First parameter is the workerid
    st.Bind(2, "John Doe"); // Second parameter the name.
    st.Execute(); // Execute the statement, writes to DB.
    st.Reset(); // Reset the statement bindings for next loop.
}
```

6.5. Class Transaction

6.5.1. Description

Class for executing database transactions. Especially useful in case of exceptions, as it will by default rollback changes when the object goes out of scope. (Thus no need for a lot of nested try-catch loops.)

6.5.2. Methods

- Constructor – create a transaction. Can be configured to auto-begin a transaction and/or auto-commit on destruction instead of the default auto-rollback.
- Begin - start a database transaction.
- Commit - commit the current transaction.
- Rollback - abort the on-going transaction and undo changes.

6.5.3. Example

Transaction example where we are adding two people to the database. If either one fails, the transaction is automatically rolled back. Result is that either both or none is added.

```
using namespace CDP2SQL;
try {
    Database db("sqlite3", "mysqlite3databasefile.db");
    Transaction t(db);
    t.Begin(); // Starts the transaction.
    Statement st(db, "insert into employee values (?, ?, ?);");
    st.Bind(1, 123); st.Bind(2, "John Doe"); st.Bind(3, 64000.99); st.Execute(); st.Reset();
    st.Bind(1, 456); st.Bind(2, "Jane Doe"); st.Bind(3, 63999.99); st.Execute(); st.Reset();
    t.Commit(); // Commit the changes to the database.
} catch (SQLException& e) { // Auto-rollback if exception
    printf("Error occurred: %s\n", e.Message()); // None was added if failure.
}
```

7. SQL Database Engine

At the bottom of the CDP2SQL application and library, lies one or more SQL database engine. The reference implementation is based on the SQLite3 database, which is a simple, file-based SQL engine.

7.1. Concurrency and multi-threading

The issue of multi-thread support is very much dependent on the underlying SQL database implementation and how it is configured.

As a rule, database objects and connections created in one thread should not be used in another thread. Instead, each thread needing database connectivity should have its own database connection, leaving the locking and transaction handling to the SQL engine itself.

7.2. Database location

SQLite and other small embedded databases tend to be accessed locally as one or more files on the local file system. This means file I/O will often be one of the limiting factors regarding performance. It is also possible to configure some of these databases to run everything in-memory

Other SQL databases are usually accessed over the network, primarily using TCP. (For example MySQL, PostgreSQL, etc). This will often be too heavy-handed for frequent logging of data, due to the network I/O overhead, but can be useful for in-frequent events that one wishes to push logging centrally.

8. Appendix

8.1. Example SQL

8.1.1. Creating tables

```
CREATE TABLE SQLLogger_Info(id INTEGER PRIMARY KEY, Name varchar(100), Value varchar(300));
CREATE TABLE SQLLogger(id INTEGER PRIMARY KEY, timestamp DATETIME, FirstSig double, SecondSig double);
```

8.1.2. Inserting data rows

```
INSERT INTO SQLLogger_Info VALUES (NULL, 'Logging', 'MANUAL');
INSERT INTO SQLLogger(FirstSig, SecondSig) VALUES (2.78, 3.14);
```

8.1.3. Requesting data rows

```
SELECT * FROM SQLLogger_Info;
SELECT FirstSig, SecondSig FROM SQLLogger WHERE id < 100;
```

8.2. References

SQLite - “self-contained, serverless, zero-configuration, transactional SQL database engine”

<http://www.sqlite.org/>

CppSQLite - “C++ Wrapper for SQLite”

<http://www.codeproject.com/KB/database/CppSQLite.aspx>

Comparison of different SQL implementations

<http://troels.arvin.dk/db/rdbms/>